THEORETICAL AND PRACTICAL ASPECTS OF TYPESTATE

IAIN MCGINNISS

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

AUGUST 2013

Abstract

The modelling and enforcement of typestate constraints in object oriented languages has the potential to eliminate a variety of common and difficult to diagnose errors. While the theoretical foundations of typestate are well established in the literature, less attention has been paid to the practical aspects: is the additional complexity justifiable? Can typestate be reasoned about effectively by "real" programmers? To what extent can typestate constraints be inferred, to reduce the burden of large type annotations? This thesis aims to answer these questions and provide a holistic treatment of the subject, with original contributions to both the theorical and practical aspects of typestate.

Acknowledgements

Scientists are made, not born, and I am grateful for the contribution of many to my forging.

I would like to thank Dr. Simon Gay and Prof. Joe Sventek of the University of Glasgow, and Dr. David Aspinall of the University of Edinburgh for their supervision, support and patience throughout my time as a PhD student. Without their guidance and encouragement, it is unlikely that this document would exist.

Dr. Aspinall and Prof. Donald Sanella provided an excellent opportunity for me to apply my early research efforts in a practical setting for their business, Contemplate Ltd. The prototyping of a static and dynamic analysis for the Hanoi language (defined in Chapter 3) at Contemplate proved to be very influential on my subsequent research. Their patronage is deeply appreciated.

Dr. Jonathan Aldrich's invitation to join his research group at Carnegie Mellon University over the summer of 2011 proved essential to the development of the ideas contained in this thesis. I am grateful for the interest he took in my research, and the experiences of visiting a prestigious academic institution which he facilitated. The excursion was financially supported by the College of Science and Engineering at the University of Glasgow, who have generally provided an excellent environment in which to learn and conduct research.

The funding for this PhD has been provided by the Scottish Informatics and Computing Science Alliance. SICSA provided much more than just financial support; their yearly student conferences and summer schools have provided additional perspective on my work, and the opportunity to network with students and established academics throughout Scotland. Without SICSA, my life as a student would have been much less interesting.

Finally, the unconditional support of my family and friends provided me the strength to complete what has become the greatest intellectual challenge of my life. I would like to thank my parents, John and Edraine McGinniss, for instilling a burning desire to question everything in me. My partner Sarah Brozio deserves recognition for tolerating my frequent emotional outbursts while writing; without her stabilising influence, I would not have been capable of providing the best possible account of my work.

The journey is the reward. — *Chinese Proverb*

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

The primary goal of a type system in a programming language is to detect and prevent errors in code. A well designed type system should detect the most common and critical errors, based upon the expected usage of the programming language. This requires that the designer of the language be aware of the intended use cases for the language, the knowledge and skill set of the programmers who will be using it, and whether the restrictions imposed by the type system provide sufficient value to be justifiable.

The programming language research community pursue and devise ever more complex type systems with the goal of preventing a wide variety of errors. Many of these type systems are built without maintaining awareness of the practical aspects of the language: the type systems becomes harder to understand and interact with as they increase in complexity. A successful type system must balance the desire to statically detect common and critical errors with the ability of programmers to comprehend and fix the errors reported. For instance, statically detecting one of the most common forms of programmer error, undeclared or misspelled variable usage, can be presented in a manner that is easy to understand and can save a programmer a significant amount of time.

The cost of a program failing is an implicit part of the decision making process in selecting the language to be used for a project. For many web applications, if the application fails, refreshing the page is often enough to fix it. While it is still desirable to reduce the frequency at which such transient errors occur, these failures are rarely critical and often easy to mitigate — restarting the process and repeating the request can often be done without the end user ever realising a failure occurred.

As such, a complex, strongly typed language such as Agda [23] is unlikely to be chosen for building a typical web application. With careful use of such a language much greater confidence in a program can be derived. However the cost of failure does not justify the additional time, effort and therefore expense involved in satisfying the type checker. Agda may, however, be an excellent choice for building a safety critical system, where the cost of failure is significantly higher. Mitigating failure through process restarts in the control systems of an aircraft or robotic surgical equipment is less likely to be a viable option compared to a web content delivery system.

Between throw-away scripts and safety critical systems is a spectrum of software with an increasing and non-trivial cost of failure, where using a language with a stronger type system is justified. In this middle ground, engineers with a formal education in computer science build systems where failure is highly undesirable if not catastrophic. Such engineers desire tool support which can provide confidence in (but not necessarily proof of) the correctness of their software, particularly in “core” components.

1.1 Save us from ourselves

In object oriented languages such as Java and C#, there are many common errors that are not dealt with adequately by the static component of the type system. Perhaps the most common source of error in such languages is dereferencing `null` — Hoare described the introduction of the `null` value for references in the ALGOL W language and the consequent replication of this feature in the languages it inspired as his “billion dollar mistake” [68]. The runtime systems of both Java and C# will halt a program immediately when a null value is dereferenced, rather than allow a program to continue with undefined behaviour (as may occur in C), but at the expense of additional runtime checks.

The situation is avoidable through the use of an *option type* in place of `null`, requiring that all references be to a valid location in memory. This eliminates the risk of runtime failure, but programmers are forced to explicitly handle the null equivalent. Interacting with option types is straightforward in a functional programming language due to pattern matching, higher order functions and closures, as demonstrated by utility functions for the `Maybe` monad in Haskell [114, Chapter 14] or the `Option` type in OCaml [98, Chapter 7]. Java 7 currently lacks these features, making interaction with option-like types cumbersome. A more sophisticated language such as Scala [111] makes handling option types easier [151]; as such it uses option types pervasively instead of `null`. Scala’s additional complexity makes this possible, but this additional complexity is a barrier to wider adoption.

Another very common pattern of errors in object oriented programs arises from the misuse of mutable private state. Types such as the `Iterator` interface in Java place stateful restrictions on their usage — the method `next()` may only be called on an iterator when another element is available, otherwise an unchecked `NoSuchElementException` will be thrown. This exception is an indication of programmer error rather than some unexpected, potentially resolvable error in the system like a network connection being severed. It is the obligation of the client of an `Iterator` to call `hasNext()` and ensure it returns `true` before calling `next()`.

While an alternative design for the `Iterator` interface may avoid this problem by returning an option type on `next`, this simply moves the fundamental problem — the iterator is depleted, and the client code must respond to this situation appropriately.

When combined with reference aliasing, the problem is exacerbated. State changes that occur in one context may not be observable in another, resulting in one alias depleting an iterator while another still believes the iterator has a value through a previous call to `hasNext()`. Such bugs are particularly insidious because the cause of the failure is separated from the point of detection by the runtime system in both space and time — the stack trace emitted for the failure provides no information on other calls that have been made recently on the object, or why the local assumptions about the state of the object are incorrect.

Bugs of this type are so common that a backlash against mutable state is occurring in object oriented programming communities. Programmers are recommended to write and use immutable object types and persistent data structures wherever possible [18, Chapter 4], specifically to avoid the issues of interacting with objects that have private mutable state in a shared environment.

Immutability runs contrary to what objects were intended to be — objects in their most fundamental form can be viewed as entities with private state and a *protocol*, that specifies the messages an entity will respond to. An immutable object, meaning an object with constant private state and a stateless protocol, has much more in common with a type constructor in a functional language.

A mutable object may have a very simple protocol, where the set of messages (or equivalently, method calls) it is willing to accept is constant throughout its lifecycle — this is what the type system of programming languages such as Java can express. This is insufficient to correctly model an object's behaviour in many cases, as the set of messages to which that object can correctly respond is dependent upon its private state. State change is typically synchronous, in response to message receipt, and safety requires that any message sent to an object is within the set that can be received for the current state. The private state can often be abstracted such that restrictions on sequences of messages can be expressed as a deterministic finite state machine.

Describing the allowed sequence of messages which a class of objects can receive, their *typestate* [140], produces significantly more complex type signatures than in an object oriented language without typestate constraints. Functions which take objects as parameters must describe the type of object they accept, in which state, and what state transformation can occur. This type annotation burden is particularly heavy in a language with function literals, which must be succinct in order to be practical.

1.2 Thesis outline

The theory behind defining and checking typestate constraints is now well-established for object oriented languages with nominal subtyping such as Java, however little work has been done to investigate whether such type system extensions fulfil the following important practical criteria: Can typestate be understood by "real" programmers? How should typestate constraints be expressed in the source code and documentation of a program? Can the annotation burden be reduced, or eliminated, through type inference? How should typestate errors be presented to the user, to facilitate diagnosis and resolution?

This thesis attempts to answer all of these questions, considering both the theoretical and practical aspects of this promising branch of object oriented type theory. The work presented herein specifically attempts to answer the following questions:

1. What essential features of a typestate constrained object must be modelled in order to accurately capture its protocol? What options exists for expressing typestate constraints? What are the strengths and weaknesses of the options for expressing typestate constraints? These questions are studied in more detail in Chapter 3, where a new method of expressing typestate constraints for Java is presented and compared to existing typestate constraint modelling methods.

2. Can a lightweight dynamic checker for typestate constraints be provided for Java-like languages, without significant modification of the language? This question is studied in detail in Chapter 4, which evaluates multiple options for providing dynamic enforcement of typestate constraints in Java, and provides a proof-of-concept implementation of the two most promising options.

3. Can programmers reason effectively about typestate-constrained objects? Does the method of expressing typestate constraints influence the ability for programmers to reason effectively about typestate? These questions are studied in detail in Chapter 5, where a user study comparing two methods of expressing typestate constraints is analysed.

4. Is type inference feasible for code which interacts with typestate-constrained objects? This question is explored in detail in Chapter 6, where the TS language is defined and studied with the express purpose of determining what type information can and cannot be reasonably inferred.

5. What theoretical challenges still remain for future typestate oriented languages? Chapter 7 outlines a hypothetical language based upon Scala, and describes the potential utility of such a language in addition to the many unanswered questions that the interactions between closures, alias control and effects present.

Overall, this thesis attempts to examine typestate in a holistic manner, and determine whether typestate is practical, useful and worthy of further study by the research community.

1.3 Thesis statement

Typestate is a practical and useful extension to the semantics of modern object oriented languages. Typestate constraints can be understood by real software engineers, and dynamic checking of typestate for existing languages provides useful safeguards and diagnostic information in both production and test environments. The primary barrier to typestate adoption, cumbersome type annotations, can be overcome through the use of type inference.

Chapter 2

Literature Review

Strom and Yemini first defined the term *typestate* [140] to capture the concept of a reference having both a *type* and an associated *state* that changes during the execution of a program. The original presentation was concerned with the state of references in a simple imperative language: a reference was either *uninitialised* or *initialised*. Reference state transitions would occur in response to explicit allocation and free operations, and dereferencing in the *null* state is undefined. As such, the state of a reference could be represented as a Moore automaton [101].

The fundamental goal of this formalism is to prevent the execution of undefined operations. This requires that the state of a reference can be accurately tracked, which was achieved through a data flow analysis [61]. Critically, references could not be copied:

> *Languages that allow unrestricted pointer assignment do not support tracking typestate at compile-time because the mapping between variable names and execution-time objects is not one-to-one. As a result, a typestate change resulting from applying an operation to an object under one name will not be reflected in the typestate of the other variable names referring to the same object.*

It was also required that procedures declare the *effect* they have on their parameters, in order to allow for a *modular* analysis. Strom and Yemini had therefore identified the core technical challenges associated with typestate, which persist to this day.

Typestate has many synonyms in the literature. Some refer to the restrictions on operations in relation to the state as *temporal constraints*, often referring to the need to do an operation *before* or *after* another.

The typestate concept generalises readily to more complex finite automata, which can be used to model state transitions and usage restrictions of objects. The type of a value would then define the full set of capabilities of the referenced value, while the state places restrictions on which of those capabilities may be used. This idea was independently described by Nierstrasz [110], using the term *active objects* to describe objects which have an interface which could be described as a regular language. Typestate restrictions are common in object oriented APIs [13], and typestate violations are a common source of difficult-to-diagnose issues [78]. It would seem, therefore, that programmers would derive benefit from typestate modelling and enforcement, if this can be provided in a manner which does not overly burden the programmer or runtime system.

Literature review outline

Systems which attempt to provide typestate modelling and enforcement for popular object oriented languages are reviewed in Section 2.1.

A related area to typestate is in the enforcement of communication protocols in distributed systems, through the use of *session types*, which are reviewed in Section 2.2.

Typestate in an object oriented language essentially externalises an abstraction of the private state of an object, of which a client must be aware in order to decide whether an operation is safe or not. This is somewhat related to the concept of *dependent types*, which are reviewed in Section 2.3.

In a typical object oriented language such as Java or C#, aliasing is both common and essential. In order for typestate enforcement to be practical in such languages, a mechanism is required by which aliasing information can be tracked and interpreted. The theory of alias control is reviewed in Section 2.4.

The ability to monitor and react to sequences of operations, an essential component of tracking state changes in typestate, can be generally useful. *Tracematches* provide a mechanism to do this in aspect oriented programming, and are covered in Section 2.5.

The expression and enforcement of *contracts* between components in a software system is also somewhat related, and some contract languages can be used to encode typestate constraints. These are covered in Section 2.6.

The specification of both contracts and typestate models for software can be a time consuming and error prone task. As such, investigations into the possibility of the inference of such contracts and models to make this process easier is an active area of research. This is covered in Section 2.7.

Early attempts to study the human factors of programming languages and the psychology of programming are covered in Section 2.8.

2.1 Typestate systems

The integration of typestate into existing type systems has been widely studied for imperative and object oriented languages, where the problems of mutable state are most obvious and a system to help uncover bugs related to typestate violation is potentially valuable. The three most common issues which these systems have attemped to address are:

1. Invalid operations. Functions and methods often carry a set of informally documented assumptions on the state of the entities they interact with that the programmer must satisfy in order to avoid error. Very common errors involve invoking an operation too many times, or out of order.

2. Consistent view of state. Where mutability and aliasing are permitted in a language, inconsistent views of an entity's state is a common problem.

3. Resource disposal. In a language without garbage collection and dynamic memory allocation, memory leaks are common and difficult to diagnose. Less seriously, in a system with garbage collection if a resource (such as a GUI window handle) is not explicitly disposed of this may tie up other low-level or scarce resources until the garbage collector is invoked after an indeterminate period of time.

Systems described in the literature either attempt to extend an existing language in a manner which is compatible with legacy code in the host language, extend a language in an incompatible way, or create an entirely new language in which to express typestate. The SLAM system [8] attempts to provide typestate verification for legacy C code, which places restrictions on what the analysis can conclude without the additional programmer support that a system such as Vault [36] relies on. Vault does not consider legacy code , allowing for sound reasoning about code at the cost of requiring significantly more guidance from the programmer. Plaid [2] takes a more radical approach in designing an entirely new language in which to explore deep integration of typestate into a language, where host language semantic choices are not an issue.

In this section, Plural and Fugue shall be explored in detail as they are recent examples of typestate systems which have usable implementations for popular languages. Each provides subtly different semantics for their typestate models and alias control strategy. Plural primarily was used as a vehicle to explore flexible typestate modelling and alias control, and so provides a rich environment in which to explore these topics. Fugue is notionally a successor to Vault and attempts to tackle resource disposal directly, with a simpler typestate model and alias control technique. The similarity of the semantics of the host languages for these systems (Java and C#, respectively) allows for direct comparison between the chosen typestate semantics and alias control strategies of each.

Plaid is also considered in more detail due to its novel presentation as a gradually typed programming language designed with deeply integrated support for typestate modelling and enforcement. Its hybrid object-functional nature also presents some interesting possibilities and challenges which do not exist for Plural and Fugue.

2.1.1 Plural

Aldrich *et al.* investigated typestate verification for Java using data flow analysis in their system known as Plural [3, 12, 16]. The work of Aldrich's group is interesting for a number of reasons:

- The data flow analysis is modular, meaning it should scale to large, component-oriented software systems, unlike whole program analyses.
- Sophisticated alias control techniques based on Boyland's fractional permissions [24] are employed, allowing stateful objects to be shared in a controlled manner.
- States within the machine can be "refined" — that is, the state machine includes a form of state subtyping. This can be useful in reducing the number of declarations required to define a model, by grouping together states based on common properties and restrictions.
- Java annotations are used to define the state machine associated with an object. This is necessary to facilitate the modular analysis and declare the fractional permission requirements of a particular method.

Plural covers many important aspects of defining state-based preconditions in real systems, but the method of expression in the form of annotations is not ideal — annotations are convenient for attaching arbitrary extra information to types that can be consumed by tools, but the syntactic restrictions make the expression of complex concepts difficult.

Plural's semantics are complex, and no contextual information is given in typestate violation warnings produced by its static analysis that would help the programmer understand what is wrong and how to fix it. With even the simplest code, many typestate violations will be found that are subtly related and often fixed as a group by small changes to the code or typestate annotations.

As an example, consider a simple class which allows alternate calls to methods `a()` and `b()`. A skeleton implementation of such a class without any Plural annotations is shown in Listing 2.1. There are 4 lines of overhead to enforce the state restrictions at runtime (all pertaining to the maintenance and checking of the variable `canCallA`). There is an implicit

```
import static java.lang.System.*;

public class AlternateMethods {

  private boolean canCallA = true;

  public void a() {
    assert canCallA;
    out.println("a called");
    canCallA = !canCallA;
  }

  public void b() {
    assert !canCallA;
    out.println("b called");
    canCallA = !canCallA;
  }
}
```

Listing 2.1: A class which allows alternate calls to a() and b()

assumption here that there can only be one reference to an instance of this class, as there is no way of determining whether one can call `a()` or `b()` dynamically. If the reference is shared, one client may call `a()` without the knowledge of other clients, leaving them with the false assumption that `a()` can still be called.

Plural can capture this information with its fractional permission system, but the resulting declarations can be difficult to understand. The Plural annotated form of Listing 2.1 is shown in Listing 2.2.

As Plural will statically verify that all calls to `a()` or `b()` are safe, there is no need to check that the client is calling the methods correctly at runtime which makes the implementation simpler. However, the introduction of an empty constructor and its annotation is necessary for Plural to function. The annotation declares that the reference generated by the constructor is unique, and that the object starts in state `CAN_CALL_A`. This constraint would be violated if the constructor were to store a reference to `this` in some other object, as a tree data structure where nodes have access to their parents may do. The annotations on the `a()` and `b()` method make explicit the required state of the object before the call can be made, and the state in which the object will be left via the "ensures" property. The `use` property states what capabilities the method itself will have, which can be one of four values:

- FIELDS — the method is allowed to access and modify fields of the object, but cannot make any method calls using `this` (i.e. `a()` cannot call `b()`).
- DISPATCH — the method is allowed to call other methods using `this`, but cannot change any fields directly.

```java
import static java.lang.System.*;
import edu.cmu.cs.plural.annot.*;

@States({"CAN_CALL_A", "CAN_CALL_B"})
public class AlternateMethods {

  @Perm(ensures="unique(this!fr) in CAN_CALL_A")
  public AlternateMethods() {}

  @Unique(use=Use.FIELDS,
          requires="CAN_CALL_A",
          ensures="CAN_CALL_B")
  public void a() {
    out.println("a called");
  }

  @Unique(use=Use.FIELDS,
          requires="CAN_CALL_B",
          ensures="CAN_CALL_A")
  public void b() {
    out.println("b called");
  }
}
```

Listing 2.2: Plural annotated form of Listing 2.1

- DISP_FIELDS — the method may both modify fields and call other methods using `this`.
- NONE — the method cannot access fields or call other methods. This is the default, if no value is provided for the `use` property.

Specifying the least `use` requirement possible makes Plural's analysis more precise, particularly when the type in question has subtypes. It can be challenging to reason effectively about such requirements, especially when the tool can only verify that a choice is sound without providing any guidance as to whether a weaker `use` requirement is possible. Plural's choice of safe defaults, such as `NONE` for the *use* property, result in subtle and initially counter-intuitive warnings. In Listing 2.2, if the *use* properties are not set to `FIELDS`, warnings are raised that the "ensures" clause of the methods cannot be honoured, as the methods cannot change the state of the object. This is counter-intuitive as the methods do not in fact access or modify any fields explicitly. However, Plural may be reasoning that with a `use` value of `NONE`, the methods cannot reasonably have side-effects and therefore no state transition can take place. The state in this example is purely virtual, but in a more realistic class the state of an object will be an abstraction of the actual private state of the object. A state transition in this case would involve changing a private field, which would require a `use` value of `FIELDS` at a minimum. Plural appears to be enforcing this even when it is not required, which is counter-intuitive when the state is virtual.

The Plural system does not alter the runtime representation of classes in any way — as a result, code which retrieves an object of type `AlternateMethods` and interacts with it through runtime reflection will be able to violate the constraints of the object. Static analysis of the form provided by Plural is useful for demonstrating the absence of a class of bugs in well-behaved code, but dynamic checking is often still required for systems that include components that are not entirely trusted. Generation of the additional code found in Listing 2.1 from the Plural annotations is possible, and desirable in a language which supports both typestate and reflection.

2.1.2 Plaid

To date, the only comprehensive attempt at designing and implementing a full general purpose programming language from scratch which draws together the state of the art in the theory of typestate is the Plaid language [2, 141]. I participated in the development of this language while working under the supervision of Jonathan Aldrich at Carnegie Mellon University in 2011.

Plaid is a *gradually typed* [136, 137, 153] language, meaning that it is a dynamically typed language in which type requirements and guarantees can be specified and statically checked in a selective manner. This technique largely mitigates the cost to the programmer of large type annotations, as they are optional, with the trade-off that method availability must be checked on every method call which adds a significant runtime overhead.

Plaid defines objects as a collection of related *state declarations* that define a hierarchical, parallel finite state machine. The state hierarchy is defined explicitly by declaring one state to be a sub-state of another; Listing 2.3 demonstrates this through the definition of an option type. Explicitly defined constructors are not necessary in Plaid — instead, a default constructor exists for each defined state that initialises all fields and methods that are defined with an initial value. Other fields can be initialised through the use of an associated code block as shown on Line 12 of Listing 2.3, where an object is instantiated in state `Some` with the `value` field initialised to the result of `f(value)`.

Plaid does not support parametric polymorphism for states, meaning that we cannot annotate the `Option` type with a type parameter and insist that the field `value` be of this type statically. As the language is dynamically typed this is of little practical consequence, but is a major limitation for the static analysis.

Plaid obviates the need for dynamic state test methods by allowing pattern matching on an object's type, as shown on Line 22. The `hasValue` method is defined for option values, but exists purely for programmer convenience rather than as a necessary part of the object's protocol.

```
state Option {

  method hasValue();
  method map(f);
}

state Some case of Option {
  val value;

  method getValue() { value }
  method hasValue() { true }
  method map(f) { new Some { val value = f(value); } }
}

state None case of Option {
  method hasValue() { false }
  method map(f) { this }
}

val o = makeSome(10)

match(o) {
  case Some { print(o.getValue()) }
  case None { print("none") }
}


val y = 2
val o2 = o.map(fn (x) => x + y)
```

Listing 2.3: Definition of an option type in Plaid

The example also demonstrates that Plaid supports *higher order functions*, and *function literals* with *implicit binding*, as shown on Line 29. This powerful feature is made practical by Plaid's dynamic typing — if the language required the full type of each function literal to be specified, they would be significantly more verbose and ruin readability. Dynamic typing allows the literals to be kept compact, as the expense of additional runtime checks and weaker static guarantees of safety. If type inference of the requirements and effect of the function literals were possible, then additional static guarantees could be provided, but Plaid does not presently explore this option.

The example in Listing 2.4 shows some types and methods with fully-specified typing information that can be statically checked. The method `callDoX` on Line 21 shows an *effect type* for the parameter a, specifying that the method requires a unique reference to a value of type A and will transform this reference to a unique reference of type B. Plaid utilises the same alias control annotations as Plural for references.

The method declaration on Line 2 specifies that the method requires a unique reference to be invoked, and will transform that reference (known as `this` in the scope of the method) to a unique reference of type B. State changes are enacted imperatively using the `<-` operator.

Effects are interpreted literally in Plaid, allowing a method to wilfully discard type information if desired, as shown by the method `discard` on Line 25. Given any input which satisfies the *precondition* specified by the effect type, the type system will assume the value is exactly the type specified after the call.

In this example, a value of type $D <: A$ is passed as a parameter to the method `callDoX`, which has effect $A \gg B$, which is compatible with its body which simply calls doX. After the call x will be treated as though it is of type B. One may have hoped the type system would have been able to derive that x must in fact be of type C, as `doX` is the only possible method that `callDoX` could invoke to change the type of the parameter from A to B, and that in state D this results in a transition to C.

Even though the static checker cannot be satisfied that the call to `onlyC` is safe, this can be tested at runtime. This idea will be revisited in Chapter 6, where such re-interpretation of effects for subtypes is a key part of the TS language defined in that chapter.

Plaid's semantics and static type system have not yet been formally defined or proven to be sound, though its runtime semantics have been formalised in the work of Sunshine [141]. The static type system is likely to be built upon the theoretical foundations of Plural.

2.1.3 Fugue

Fugue [37, 38] is a system for annotating code in .NET based languages with typestate restrictions, and ensures that resources are correctly disposed of. This is achieved through type

```
state A {
  method doX() [unique A >> unique B] {
    this <- B
  }
}

state B case of A { /* ... */ }

state C case of B {
  method onlyC() [unique C >> unique B] {
    this <- B
  }
}

state D case of A {
  method doX() [unique D >> unique C] {
    this <- C
  }
}

method callDoX(unique A >> unique B a) {
  a.doX();
}

method discard(unique A >> none A a) {}

method main() {
  val unique D x = new D;
  callDoX(x);
  // not statically safe
  x.onlyC();
}
```

Listing 2.4: Loss of type information in Plaid

```
[WithProtocol("raw", "bound", "connected", "down")]
class Socket {
  [Creates("raw")]
  public Socket(...);

  [ChangesState("raw", "bound")]
  public void Bind (EndPoint localEP);

  [
    ChangesState("raw", "connected"),
    ChangesState("bound", "connected")
  ]
  public void Connect (EndPoint remoteEP);

  [InState("connected")]
  public int Send(...);

  [InState("connected")]
  public int Receive(...);

  [ChangesState("connected", "down")]
  public void Shutdown (SocketShutdown how);

  [Disposes(State.Any)]
  public void Close ();
}
```

Listing 2.5: Fugue specification for a network socket

---

and method annotations that declare the set of states for an object, state transitions enacted by a method, and restrictions on method calls based on the current state. Fugue calls the state machine on an object a “protocol”, drawing on the similarities with stateful communication in distributed systems (which is also the basis of session types, as described in Section 2.2). An example of a Fugue specification for a network socket in C#, taken from [38], is given in Listing 2.5.

The `WithProtocol` annotation declares the set of states on a type. `Creates`, `ChangesState` and `InState` all declare the pre- and post-conditions for the methods in a type. Methods may have multiple annotations to cover their usage under different circumstances, as shown by the `Connect()` method, which can either be used after the socket is bound to a known local port, or directly from its raw state in which case an ephemeral local port will be assigned.

Fugue’s most interesting feature compared to other typestate systems is that it ensures an object reaches the end state (through a call to a method with the `Disposes` annotation) before it is dereferenced and eligible for garbage collection. This is a subtle and common cause of bugs that garbage collection only partially alleviates — while a `Socket` object may no longer be referenced, it is unpredictable how long it will survive before it is garbage collected and the connection is closed by a finalizer, allowing the connection to continue to waste network resources (data can still be received) and CPU time (processing of incoming data may still occur despite it not being consumed in any meaningful way). Thus, Fugue can ensure that programmers correctly dispose of objects which are tied to other resources which must be explicitly disposed of.

Fugue deals with the aliasing problem by explicitly annotating parameters and fields with `NotAliased`, `MayBeAliased`, and `Escaping`. If a field or parameter is marked with `MayBeAliased` and `Escaping`, the reference can be duplicated and passed as a parameter to another method freely. If a parameter is marked as `MayBeAliased` but not `Escaping`, then it can be used and passed as a parameter in a way that the alias will not be permanently duplicated — it can be "borrowed" for the duration of a method call, but not stored in another field. Finally, if a field is marked as `NotAliased`, it can be used locally on the containing object and passed as a parameter to other methods marked as `NotAliased`, or `MayBeAliased` and not `Escaping`, allowing it to be borrowed. `NotAliased` is essentially a form of linear typing, with relaxed semantics with regards to parameter passing. If an annotation is not specified on a field, then it is assumed to be `MayBeAliased`, and unannotated parameters are assumed to be `MayBeAliased` and `Escaping`.

Objects can only change state when they are `NotAliased`, which is quite a strong restriction and more restrictive than Plural's fractional permissions system.

Fugue implements the checking of typestate and alias control as a data flow analysis over the compiled .NET CLI assemblies. Due to the annotation based approach, similar to Plural, the analysis is modular and fast for large code bases — `mscorlib.dll`, containing 13385 methods, can be checked in under one minute on a Pentium 4 processor in 2004 according to DeLine [38].

2.2 Session types

Session types model communication protocols between processes in distributed systems [71, 143]. A session type defines the flow of typed messages on a bi-directional *channel* between two processes, and enables static checking of channel usage to ensure that a peer sends and receives messages in the order dictated by the session type.

Session types are formalised as a type system for Milner's π-*calculus* [97], which provides a foundation for the description of parallel computation. The π-calculus, in its simplest form, is untyped and Turing complete, and has provided an excellent context in which to formalise process equivalence (defined in terms of a *bi-simulation* relation) and study the variety of possible failure modes in distributed systems.

One fundamental class of communication error is for one process to send an unexpected message to another process. A simple type may be associated with a communication channel in order to prevent this, but then a higher-level problem becomes apparent — communication between processes is typically structured, with an expected *order* to the flow of messages. Messages typically contain different content at different times, and therefore cannot really be considered to be of one type. Session types provide the mechanism to enforce both the order of communication, and the types of the messages that may be sent. Each end of the channel has a session type which is the "mirror image" or *dual* of the other, which is to say that if the session type for one process dictates that it may only send an integer, then the other party may only receive an integer. A simple example of session types adapted from Honda's work [71] is shown in Listing 2.6 (it is modified for clarity), which demonstrates the definition of an ATM process which communicates with a user process and a bank process.

In this example, a user process initiates a session with an ATM, sends its user id, and is then presented with four options: it may deposit cash, withdraw cash, check its balance or quit the session. When attempting to withdraw, if sufficient funds are available then the user is informed that the requested amount has been successfully withdrawn. If, however, the requested amount would exceed the available balance, then the user will be informed that the requested amount would result in their account being overdrawn. After completing any transaction other than quit, the user may request any of these four options again.

`ATM(user2atm, atm2bank)` defines the behaviour of the ATM process in terms of two channel sources, `user2atm` and `atm2bank`, which are shared with the user process(es) and the bank process(es) respectively.

A user process would initiate communication with an ATM process through an expression of form `request user2atm(c) in P`, which creates a channel to a receiving ATM process which is bound to the variable c in the successor process P. The ATM accepts a process though an expression of form `accept user2atm(c) in P`. The roles of the parties are determined from this point such that if one wishes to send, the other must be prepared to receive, and this *duality* is enforced by the session type of the channel on each end.

Messages may be sent or received on channels through the expressions `c![a, b, ..., n]; P` and `c?(a, b, ..., n) in P` respectively. In the case of receiving, the names specified in parentheses are bound in the successor process P. A process may offer a choice in behaviour through the channel with an expression of form `c -> { a : Pa, ..., n : Pn }`, where a through n are the *labels* of the options. A process may select an option with an expression of form `c <- label; P`.

The channel sources `user2atm` and `atm2bank` are typed with a pair of session types — the type of `user2atm` is shown where X is the dual of X'. The initiator of the channel will be given a channel of type X while the receiver will be given a channel of type X'.

```
ATM(user2atm, atm2bank) =
  accept user2atm(uchan) in
  uchan?(userId);
  Actions[user2atm, atm2bank, userId, uchan]

Actions(user2atm, atm2bank, userId, uchan) =
  uchan -> {
    deposit  : request atm2bank(bchan) in
               uchan?(amount) in
               bchan <- deposit;
               bchan![userId, amount];
               Actions[user2atm, atm2bank]

    withdraw : request atm2bank(bchan) in
               uchan?(amount) in
               bchan <- withdraw;
               bchan![userId, amount];
               bchan -> {
                   success: uchan <- dispense;
                            uchan![amount];
                            Actions[user2atm, atm2bank]

                   failure: uchan <- overdraft;
                            Actions[user2atm,atm2bank]
               }

    balance : request atm2bank(bchan) in
              bchan <- balance;
              bchan?(amount) in
              uchan![amount];
              Actions[user2atm, atm2bank]

    quit : ATM(user2atm, atm2bank)
  }

user2atm : <X, X'> where
  X' = ?nat.Y
  Y = & {
    deposit : ?nat.Y ;
    withdraw : ?nat.+ { dispense : ?nat.Y ; overdraft : Y }
    balance : !nat.Y
    quit : exit
  }
```

Listing 2.6: The definition of an ATM process, which communicates with a user process and bank process through sessions, written in the session language of Honda *et al.* , defined in [71]

Therefore, the type of `uchan` is X'.

Channel endpoints such as `uchan` are *linear* — they must be used to exhaustion and cannot be duplicated. *Channel delegation* is possible, where a channel endpoint can be sent through a channel to another process which is responsible for exhausting the channel from that point.

Session types ensure that a well-typed program cannot send (or receive) a message of the wrong type to (or from) a channel, and allow messages of different types to be sent at different points in the program, unlike in simpler π-calculus based type systems, which must use a chain of separate channels and results in a much more fragmented description of the same fundamental communication between two parties.

The expression of session types in functional programming languages has been studied [57], and two implementations for the Haskell language exist [107, 124]. Object oriented languages have been designed with support for communication channels between threads or processes with statically enforced session types [29, 40–42, 103], though perhaps the most notable is the Session Java (SJ) language [73, 74] which extends Java and provides both synchronous and event-driven support for session types, and allows for efficient and type-safe implementations of common distributed algorithms [108].

Session types have also been extended to model communication between multiple parties [15, 22, 39, 72, 159], which allows for the modelling and enforcement of communication protocols between groups of processes. The Scribble modelling language [70] provides a practical method of specifying such protocols, where processes are assigned *roles*. An implementation of a suite of tools to check Scribble protocol definitions for deadlock and live-lock exists [69], with future plans to allow for the static analysis of code for conformance to such protocols.

2.2.1 Sessions for objects

In a *pure* object oriented language, all entities are objects and communication between entities takes the form of *messages*, or equivalently *method calls*. Unlike the π-calculus, control flow is typically sequential, carried with the message and back again when a response is generated. However, there are fundamental similarities between offering a choice in a session type and the specified set of methods on the interface of an object: the labels of session types are effectively method names, and any sequence of sends after choosing a label could be regarded as the parameters of that method, with the following sequence of responses as the return values. Callbacks are typically offered by providing the receiver of a method call with a handle to the initiator, or a proxy to it with a more appropriate interface.

When Java interfaces are viewed as session types, they offer a uniform choice after each completed transaction, though as already discussed this is often an over-simplification of the true behaviour of the object. By treating interfaces as session types, a more dynamic interface is possible.

Session Java does not attempt to extend its support for the tracking and modification of a channel's type to object interfaces in general. This was attempted by Gay and Vasconcelos [58], where objects with interfaces that are governed by session types are known as *non-uniform objects*. This term is derived from the work of Ravara [129, 130], which studies typestate-like restrictions in an actor-model extension of the π-calculus known as *TyCO*. A related set of work based on Nierstrasz' *active objects* was undertaken by Puntigam [125, 126].

An example of a non-uniform object specification for an `Iterator` adapted from Gay and Vasconcelos' work is shown in Listing 2.7. The session type declares four states which enforce the standard set of rules for an iterator:

- S: we do not know if there is another item, and must call `hasNext()` to determine if we do.
- T: we have a new item, and can call `next()`.
- U: we have taken an item from the iterator, and can either call `remove()` to remove it, or `hasNext()` to see if there is another item.
- V: we have taken an item from the iterator, and we know there is another item available. So, at this point we can call both `remove()` and `next()`.

In short, these rules state that `hasNext()` can always be called, `next()` can be called after `hasNext()` returns true, and `remove()` can be called once for each call to `next()`.

One of the primary contributions in Gay and Vasconcelos' work is the *link* type, which associates the return value of a method which represents an *internal choice*, such as the method `hasNext` makes. When `hasNext` is called on `iter`, the returned value is of type `boolean link iter`, while `iter`'s type becomes a *tagged union* `<false : Iterator[S] ; true : Iterator[T]`. When the return value is tested, the *link* to `iter` allows us to resolve the actual type of `iter`. Until this is done `iter` cannot be used, though testing the return value need not be performed immediately.

Another major contribution of this work is that an object with a session need not be wholly consumed within a method — it can be passed as a parameter to other methods, and stored as a field. The reference cannot be copied or shared however, to ensure that the state of the object can be accurately tracked.

```
interface Iterator {
  session S where
    S = { hasNext : <false: S, true: T> },
    T = { hasNext : <false: S, true: T>,
          next : U
        },
    U = { hasNext : <false: S, true: V>,
          remove : S
    },
    V = { hasNext : <false: S, true: V>,
        next: U,
          remove: S
    };
  boolean hasNext();
  Object next();
  void remove();
}

Iterator[S] iter = ... ;
boolean moreLeft = iter.hasNext();

// iter : <false: Iterator[S], true: Iterator[T]>
// moreLeft : boolean link iter

while(moreLeft == true) {
  // iter : Iterator[T]
  Object o = iter.next();
  // iter : Iterator[U]

  System.out.println(o);

  iter.remove();
  // iter : Iterator[S]

  moreLeft = iter.hasNext();
}

// iter : Iterator[S]
```

Listing 2.7: Session typing and link types as found in Bica, a Java language variant defined in the work of Gay *et al.* [58]

2.3 Dependent types

Dependent types are types which are parameterised by terms [119, Chapter 2], and represent one axis of Barendregt's Lambda Cube [9] which enumerates the means through which terms and types can be related. Dependent types allow for types such as $List(4)$, which can be chosen to mean a list with exactly 4 elements in it. Here, the type `List` is dependent on a term of type `nat`, meaning it is of *kind* $nat \rightarrow *$.

Singleton types can also be constructed in dependent types, for instance $SNat(5)$, which can be chosen to mean a type with only one value, which is constructed from the nat value 5.

Functions can be defined in a dependently typed language such that they make explicit the actions of the code they contain in a way that is not otherwise possible. For instance, the type of the `prepend` function for a list could be:

$$prepend : \Pi(n{:}nat).\, item \rightarrow List(n) \rightarrow List(n+1)$$

This means that the function takes an $item$ and a list of size n, and returns a list of size $n+1$. Π is used to indicate a term dependency for a type in the same way that λ indicates a term dependency for a term: $\lambda x.(x+1)$ is a term which adds the value 1 to a dependent term x, for instance.

Dependently typed lists and arrays which declare their size eliminate the need for runtime bounds checking [157], by declaring the `get` function for an array to only accept indices which are less than the size of the array:

$$get : \ \Pi(size{:}nat).\, \Pi i{:}\{i{:}int \mid 0 \leq i \wedge i < size\}.Int(i) \rightarrow List(size) \rightarrow item$$

In general, the term parameters in types can be anything, but as the complexity of the terms allowed by a language increases it becomes increasingly difficult to distinguish between compilation and execution of the program. However, by constraining the allowable structure of terms to simple forms, such as values of an enumeration or natural numbers with simple addition, we can reason about the changes to these terms in a static way and so retain the separation between compilation and execution.

While the practical utility of dependent types has been explored in functional programming languages [6, 23, 92, 156], less work has been done in an imperative space [106, 155], and none in object oriented languages. This is due to the danger that updating a value may inadvertently change the *type* of another value, in ways which are difficult to reason about. Mixing subtyping, a core feature of object orientation, with dependent types is likely to produce a very complex or overly restrictive type system.

```
\\ x : $\Sigma(x{:}int).List(x)$
if(x.left > 0) {
        y = x.right; // y : $List(x)$, where $x > 0$
        (item, y) = y.remove();
        // y : $List(z)$ where $z > -1$
} else {
        y = x.right; // y : $List(x)$, where $x \le 0$
        result = y.add(new item("hello"));
        // result : $List(z)$ where $z \le 1$
}
```

Listing 2.8: Dependently typed manipulation of a list in Java-like pseudo-code

---

However, dependent types have potential for usage within object oriented languages as a way of making an abstraction of an object's private state explicit and part of its interface, which is the essence of typestate. In the example of a list, it is clear that removing an item from a list should not be legal if the list is empty. This can be expressed with predicates on the dependent term:

$$remove : \Pi(n{:}\{n{:}nat \mid n > 0\}).\, List(n) \to (item, List(n-1))$$

This means that the `remove` function can be given a list with at least one element, and will return a pair containing the removed element and a list of size $n - 1$. The function is undefined for lists with an integer term $n \le 0$, and so it is the job of the type checking algorithm to ensure that no attempts are made to call `remove` when this is the case.

Dependent type theory also defines a generalization of *variant types* (also known as tagged unions) for types parameterised by terms, known as *sigma types*. Sigma types represent a dependent type with an unknown (but perhaps constrained) term. For instance, $\Sigma(x{:}nat).List(x)$ represents a list of unknown size, and values of this type are conceptually pairs of form $(x{:}nat, y{:}List(x))$. The first item in the pair can be considered to be a "label" that disambiguates the type of the second term, but unlike variant types the range of values on the first term need not be finite. By constraining the value x, we can have a sigma type to represent a non-empty list:

$$\Sigma x{:}\{x{:}int|x > 0\}.List(x)$$

Similar to selection of a label on a variant type, we can write code such as that contained in Listing 2.8. The type checker can determine that the code in each part of the if statement is type safe due to the check on the value of `x.left`.

From this example one can see that dependent types can be used as a form of precondition on a function: by stating that we are only willing to accept a certain range of terms in the

```
myList = createList();
// myList : $List(0)$

myList = myList.add(new item("hello")); // : $List(1)$
myList = myList.add(new item("world")); // : $List(2)$
(item1, myList) = myList.remove(); // $List(1)$
(item2, myList) = myList.remove(); // $List(0)$
```

Listing 2.9: Safe manipulation of a list without checks in Java-like pseudo-code

```
IterState = $\{S, T, U, V\}$
Iterator :: IterState $\rightarrow *$
hasNext : $\Pi st{:}\{S, U\}.\,Iterator(st) \rightarrow \Sigma newSt{:}\{S, T, V\}.Iterator(newSt)$
next : $\Pi st{:}\{T, V\}.\,Iterator(st) \rightarrow \Sigma newSt{:}\{S, U\}.Iterator(newSt)$
remove : $\Pi st{:}\{U, V\}.\,Iterator(st) \rightarrow \Sigma newSt{:}\{S, U\}.Iterator(newSt)$
```

Listing 2.10: Dependently typed version of an iterator

dependent type, we can control the legal values that can be provided to the function as a parameter. We can also provide more information about the return value than previously possible, giving guarantees that the sequence of calls in Listing 2.9 is safe. This can be viewed as a form of postcondition.

2.3.1 Encoding typestate in dependent types

The list example illustrates one simple case where dependent types can give us static guarantees where a simpler type system cannot. Typestate-like constraints can also be encoded into dependent types by defining the states we desire as an enumeration, and then restriction function availability to certain states.

Given an enumeration `IterState` composed of the state labels in Listing 2.7, a set of functions for the `Iterator` type as shown in Listing 2.10. The acceptable set of input states can be explicitly declared, and by returning sigma types we can indicate uncertainty in the return type that must be disambiguated by a switch statement. One important capability we have lost compared to session types is that we cannot explicitly declare the transitions between states in this form. However, if instead we use the form shown in Listing 2.11 which uses two booleans to represent whether `next()` and `remove()` can be called, the sigma types can be removed with the exception of the return type of `hasNext()`, which is where we would want a sigma type to exist. The example code contained in Listing 2.11 demonstrates that we can program in a familiar way, with type safety guarantees that we cannot call `next()` or `remove()` at invalid points in the program. This alternative approach is particularly interesting, as the boolean dependent terms of the type are like capability bits, representing the available methods.

```
Iterator :: $bool \to bool \to *$
hasNext : $\Pi a{:}bool.\Pi b{:}bool.Iterator(a,b) \to \Sigma x{:}bool.Iterator(x,b)$
next : $\Pi a{:}true.\Pi b{:}bool.Iterator(a,b) \to (item, Iterator(false,true))$
remove : $\Pi a{:}bool.\Pi b{:}true.Iterator(a,b) \to Iterator(a,false)$

// i : Iterator(false,false)
var hasNextResult = hasNext(i);
while(hasNextResult.left == true) {
        i = hasNextResult.right;

        // i : Iterator(true,false)
        var nextResult = i.next();
        print(nextResult.left);

        // nextResult.right : Iterator(false,true)
        i = nextResult.right;

        hasNextResult = hasNext(i);
        canRead = hasNextResult(i);
}

// i : Iterator(false,false) or Iterator(false,true)
```

Listing 2.11: Better form of the iterator type with sample usage, in Java-like pseudo-code

---

2.3.2 Effects in Idris

The Idris language [25] is a dependently typed functional programming language with support for *embedded domain specific languages* (eDSLs). Very recent work in Idris has explored the possibility of providing a domain specific language for the composition of effectful operations using *algebraic effects* [26]. The goal of this DSL was to provide easier composition than is possible with monads, which are the primary mechanism of controlling side-effects in pure functional languages such as Haskell.

Idris' `Effect` DSL leverages dependent types to control and track effects on fine grained mutable resources. The simplest effect, analogous to the `State` monad in Haskell, allows for a mutable reference to be retrieved and modified through the course of an imperative program. The canonical example provided in Brady's work [26] is to store an integer counter, and use this to provide unique identifiers to each node in a tree, and is shown in Listing 2.12.

In this example, `tag` is a function which takes a tree and returns an *effect program* which requires access to a `STATE Int` resource which is labelled `Tag`, and should exist in the *computation context* `m`. The value in `Tag` is extracted using the function `get` on Line 8, incremented by one and injected back into `Tag` using `put` on Line 9.

The effect program generated by `tag` can be converted into a pure function where the tags inserted into the tree nodes start at i using the `runPure` function to which a (pure) environment is provided along with the effect program. Environments constructed from monads

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

tag : Tree a -> Eff m [Tag ::: STATE Int] (Tree (Int, a))
tag Leaf = return Leaf
tag (Node l x r)
  = do l' <- tag l
       lbl <- Tag :- get
       Tag :- put (lbl + 1)
       r' <- tag r
       return (Node l' (lbl, x) r')

tagPure : Int -> Tree a -> Tree (Int, a)
tagPure i t = runPure [ Tag := i ] (tag t)
```

Listing 2.12: Idris effect DSL code for annotating a tree with unique labels

representing I/O or other impure entities can be used via a different function, `runWith`.

Generally, an environment could contain a labelled entity which provides operations conditionally, based upon its current state. An example from Brady's work [26] demonstrates this for file handles, which are either readable or writeable, and thus functions which manipulate file handles are conditionally available based upon this parameter of the file type.

The primary drawback of this work is that effects are not inferred, and must be fully specified. Additionally, the type errors generated when a constraint is violated are difficult to understand, due to the translations that occur between the DSL and the base Idris language. The system shows promise and is a significant step forward for the practical application of dependent types, but the inherent complexity of dependent types may present a significant barrier to uptake.

2.3.3 Dependent typestate

Dependent types, through the use of natural numbers as dependent terms, can model certain typestate restrictions in ways that finite state machines cannot. The relationship between the `push` and `pop` methods of an unbounded stack data structure (where we cannot call `pop` if the stack is empty) cannot be encoded directly into a finite state machine without the help of a dynamic state test method such as `isEmpty`. However with a natural number to represent the size of the stack as part of the type, we can express the stack protocol without `isEmpty`. The dependent values could be thought of as being attached to the *state* rather than the type, allowing different sets of labelled dependent values to be used as necessary. This is shown in Figure 2.1 for a stack, where the `EMPTY` state has no dependent values whereas the `NOT_EMPTY` state has a natural number representing the number of elements in the stack. More complex types could be imagined with more state labels and a wider variety

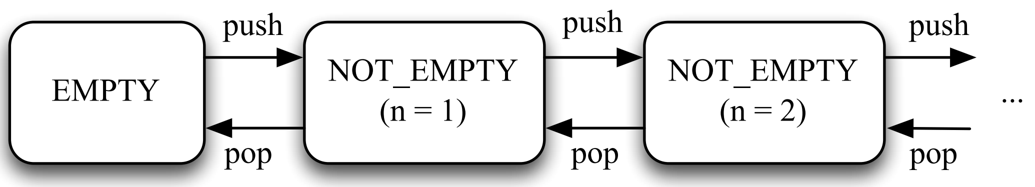


Figure 2.1: An infinite state machine for a stack data structure

of dependent values.

Generally, dependent types make it possible to support protocols which are not regular grammars. This is alluded to in DeLine's work [37] which calls this concept *dependent typestate*. This comes at the cost of a vastly more complex type system, but opens new possibilities for directly specifying some very natural protocols, such as an integer prefix indicating the number of messages of a certain type that are expected to follow.

2.4 Alias control

If multiple references exist to a typestate-constrained value, then care must be taken to ensure that the references have a consistent view of the state of that value. In general, state changes which do not monotonically increase the capabilities of an object can only be permitted on a unique reference. A method of ensuring that a reference is unique is therefore essential to typestate enforcement.

The simplest option is to prevent aliasing entirely, as in the original Strom and Yemini paper on typestate. In a language with procedures and call-by-reference semantics, this requires that a variable cannot be used as a parameter twice, as in Reynold's *syntactic control of interference* [131, 132].

2.4.1 Linear and uniqueness types

Linear types, an adaptation of the rules of Girard's linear logic [59], can provide the same *uniqueness* guarantee that is necessary to allow safe state changes. Linear logic is a variant of classical logic in which *contraction* and *weakening* are not permitted. In linear type systems this means that a linear value must be used *exactly* once; it cannot be discarded or duplicated.

In Wadler's seminal work [148], mutable array references are permitted without interference by using a linear type — updating or reading from the array notionally produces a new array reference, with the old reference consumed. Array references cannot be duplicated, and must be explicitly discarded. An example of this adapted from Wadler's work [148] in an ML-like language is shown in Listing 2.13. Atomic types such as `Int` which are prefixed with '¡'

```
alloc : ¡Array
lookup : Int → ¡Arr → Int ∘ ¡Array
update : Int → Int → ¡Array → Int ∘ ¡Array
dealloc : Int ∘ ¡Array → Int

let arr = alloc in
let (x, arr) = lookup 0 arr in
let (y, arr) = lookup 1 arr in
let arr = update 2 (x + y) arr in
dealloc (2, arr)
```

Listing 2.13: Wadler's linear types in an ml-like language

are linear, $T \circ U$ is a linear pair type, and $T \multimap U$ is a linear function. Any data structure which contains a linear reference must itself be linear, and any closure which captures a linear reference must be linear.

The primary criticism of this approach is that the "threading" of the `arr` reference is cumbersome. Additionally, for non-destructive operations such as `lookup`, linearity is overly restrictive. Wadler resolves this second point by introducing a variant of let-binding which allows a linear reference to be duplicated temporarily, but it may only be used in a read-only manner. Systems which relax linear typing in this fashion are highly sought after for object oriented systems where aliasing is required to express many common patterns.

Linear types have been exploited to support session types in both functional [57] and object oriented languages [58, 147].

Morrisett's L^3 language [102] exploits linearly typed references to support *strong update*, which allows a memory cell to hold values of unrelated types at different points in the execution of a program. The ability for a reference to change type is essential to the idea of typestate.

The Clean language offers *uniqueness types* [122], which offer essentially the same characteristics as linear types without the need for explicit "threading".

The Vault system [36] uses linear *resource keys* to control access to *tracked types* such as files and explicitly managed blocks of memory in an imperative language. The distinguishing feature of Vault is that the key to access a resource is separate from the resource; this allows the references to the resource to be duplicated, while the key is tracked separately in a *key set* which is carried around with the logical thread of control.

Where disposal is not of concern, such as when a garbage collector is in use, *affine* types may be used — affine logic permits contraction but not weakening. Affine types have been used in the Alms language [145, 146] for typestate, providing the uniqueness guarantee required for safe state change, without the need to also explicitly release a value.

2.4.2 Fractional permissions

Plural and Plaid implement an alias control scheme which is based on Boyland's *fractional permissions* [24]. Each reference has an associated *fraction*, which is conceptually a rational number between 0 and 1. A *whole* permission, with value 1, allows full read and write access to the memory cell to which the reference points, while a fraction $f \in (0, 1)$ only permits read-only access. Permissions may be split into pieces; the relative size of the pieces is unimportant, as long as nothing is lost in the process:

$$\frac{0 < g \leq g' \quad g + g' = 1}{f = f.g + f.g'}$$

$f.g$ and $f.g'$ are *complementary* fractions. Knowledge of whether two permissions f_1 and f_2 can be recombined requires proof that they are complementary. If the fractions were reified in the runtime system in some manner, this may be checked with no assumptions, and would allow temporary joining of permissions, but would greatly increase the cost of references in the runtime system. When checking statically, it must be possible to derive that two fractions are complements, or that this is an assumption that can be satisfied elsewhere in the program (i.e. as a precondition of a function for two parameters, placing the burden of proof on the invoker).

Boyland used this system to statically ensure the absence of interference in parallel execution in a simple, first-order language. A term "captures" the fractions for references it requires to execute safely, and all terms that are run in parallel (through the parallel execution term `t = t' | t''`) must use complementary fractions. A term such as `y := *x + 1 | z := *z - *x` is safe, as neither term writes to a variable that the other reads. The term `x := *x + 1 | y := *x` is not safe, as the left term writes to x while the right reads from it.

Plural's alias control system refines this concept by creating *permission types*, which are formalised independently from typestate in the recent work of Naden *et al.* [104]. While the same fraction split / join rules are still present, complementary fractions are tagged with a type that grants different capabilities through the reference:

- Unique — The reference is the only reference to the object, and so no restrictions on the manipulation of the object are necessary.
- Full — The reference has exclusive control over changing the object's state. There may be other read-only references, which are all *pure*.
- Share — The reference has the non-exclusive ability to change the object's state. This requires careful cooperation between the references, and often means very few as-

$$\begin{array}{lcl}
\texttt{Unique}(1) & = & \texttt{Full}(f) + \texttt{Pure}(f') \\
\texttt{Unique}(1) & = & \texttt{Share}(f) + \texttt{Share}(f') \\
\texttt{Unique}(1) & = & \texttt{Immutable}(f) + \texttt{Immutable}(f') \\
\texttt{Full}(f) & = & \texttt{Pure}(f.g) + \texttt{Pure}(f.g') \\
\texttt{Share}(f) & = & \texttt{Share}(f.g) + \texttt{Share}(f.g') \\
\texttt{Pure}(f) & = & \texttt{Pure}(f.g) + \texttt{Pure}(f.g') \\
\texttt{Immutable}(f) & = & \texttt{Immutable}(f.g) + \texttt{Immutable}(f.g')
\end{array}$$

Figure 2.2: Fractional permission splitting rules on Plural

sumptions can be made about the state of an object before it is used. The state will often have to be dynamically determined every time the object is used.

- Pure — The reference has non-exclusive access to the object, and cannot change its state. As another reference will exist that has a *full* permission, care must be taken to determine what state the object is in before it is used, similar to a share permission.

- Immutable — The reference has non-exclusive access to the object, however all references to the object are also *immutable*. This form of permission is useful for shared data structures like lookup tables that are set up in advance and then widely shared throughout the program.

The split / join rules for the fractions are defined in Figure 2.2.

A unique permission can be split into a full and pure pair, or into two share references, or into two immutable references. A pure permission can be split into as many additional pure permissions as required, as can share and immutable references.

Permissions may be temporarily split, for instance to pass an immutable reference to a method with the guarantee that the permission will be released upon return from the method. A unique permission can be split into two immutable references, and one passed to a method that wishes temporary access. Upon return from the method we are guaranteed that the borrowed reference is gone, and so the original reference can safely return to unique status. In Plural “borrowing” permissions for parameters is the default behaviour.

Borrowing permissions can also be applied to values returned from methods. This is useful for types such as data structures, where we may wish to fetch a value from a set, do something with it and have it returned to the set with the guarantee that it has not changed in any way that would affect the set’s assumptions about the object. For instance, it would not be possible to change the state of the returned object such that the type parameter of the data structure would be invalid.

It is also necessary to specify that a permission can be captured by a method. For instance, for the storage of object references in a data structure, it is necessary to capture the reference in order to allow that reference to be useful to the program at a later point.

2.4.3 Reference roles

Plural also contains an interesting feature known as "dimensions", where an object can have more than one orthogonal state. This allows for behaviour in the object to be partitioned into these dimensions, and different permission types to be given and split in those dimensions. A use case for dimensions would be in a queue data structure: the structure notionally supports two separate forms of interaction from a "producer", which inserts elements into the queue, and a "consumer" which removes elements from the queue.

Militão provides a similar mechanism which provides a more flexible user-defined splitting mechanism than Plural's fractions, named *views* [96]. An example of a pair with views taken from Militão's work [96] is shown in Listing 2.14. A `Pair` type is declared to be composed of two views, `Left` and `Right`, while an `EmptyPair` is composed of a `EmptyLeft` and `EmptyRight`. The field sets of each are disjoint, ensuring that the views cannot interfere, though overlapping field sets are possible where those in the intersection are read-only. When an `EmptyPair` is instantiated it is also implicitly of type `EmptyLeft * EmptyRight` by the defined equivalence.

The declaration of `setLeft` on Line 11 indicates that the method may only be called on a `EmptyLeft` reference, and that it will change this reference to a `Left` reference. The value x must be of type `L` and will be consumed by the method — the original reference becomes of type `none`, on which only effectively static methods such as `auto_init` could be invoked.

We may invoke `setLeft` with a reference of type `EmptyPair`, as the `EmptyLeft` view can be extracted, modified and re-inserted into the type. The `init` method on Line 19 demonstrates that a reference can be safely aliased where it is split along distinct view lines. The `this` reference is split into three pieces, one of type `EmptyLeft`, one of type `EmptyRight`, and one of type `none` used for the invocation of `auto_init`.

Unbounded splitting of views is permitted if this is explicitly declared for a view by providing an exclamation mark suffix to a view name: for instance `Integer!`, where Integer is effectively an immutable type.

Militão's views provide a less rigid system of permission splitting than in Plural or Plaid, allowing the programmer to specify the intended usage of a type more clearly. The system relies heavily on annotations just as Plural does, which may still present a barrier to acceptance.

```
class EmptyPair {
  view EmptyLeft { none l; }
  view EmptyRight { none r; }
  view Pair { L l; R r; }
  view Left { L l; }
  view Right { R r; }

  EmptyPair = EmptyLeft * EmptyRight;
  Pair = Left * Right;

  none setLeft(L>>none x) [EmptyLeft >> Left] {
    this.l = x;
  }

  none setRight(R>>none x) [EmptyRight >> Right] {
    this.r = x;
  }

  none init() [EmptyPair >> Pair] {
    this.auto_init(this, this);
  }

  none auto_init(EmptyLeft >> Left l, EmptyRight >> Right r)
                [none >> none] {
      l.setLeft(new L());
      r.setRight(new R());
  }
}

var p = new EmptyPair();    // p : EmptyLeft * EmptyRight
val l = new L();            // l : L
p.setLeft(new L());         // p : Left * EmptyRight, l : none
```

Listing 2.14: A pair type with views

2.5 Tracematches

Aspect oriented programming allows for the injection of code into a program as a secondary compile pass or at runtime through bytecode manipulation, at specially identified points called “join points” [82]. Two common and useful join points are entry to and exit from a method. This allows one to do various useful tasks, such as dynamically checking method pre- and postconditions declared as annotations on the method. An extension of this idea related to state based preconditions is known as a “tracematch”. Tracematches define a sequence of methods on one or more related objects that, when the sequence occurs, the aspect is to be executed. While this can be generally useful, it has been identified as a means of capturing illegal sequences of method calls and throwing an exception in the aspect to prevent the illegal pattern, or to log its occurrence.

Naïve implementation of tracematches can be devastating to performance [7]. Every method invocation associated with an object referenced by a tracematch is considered a “potential point of failure” (PPF), and must be wrapped with additional dynamic checks. Static analysis can be employed to reduce the list of such PPFs where the tracematch may execute. By doing this, runtime checking overhead can be reduced to acceptable levels, but not entirely eliminated [19].

Tracematches, via their aspect oriented roots, have the distinct advantage of not requiring any changes to the host language. However, this forces any static analysis of tracematches to deal with aliasing without any annotation support; as a result, tracematch analyses are typically whole-program and imprecise. The analyses are sound, in that they will not eliminate a PPF unless absolutely certain it cannot fail, but unnecessary dynamic checks will still potentially be performed.

Due to the computational cost of whole program analysis and processing of aliasing, many analyses are incremental, starting with cheap context-insensitive and flow-insensitive analyses to eliminate as many potential points of failure as possible before proceeding with a more expensive context-sensitive and flow-sensitive analysis [19]. This greatly increases the precision of the analysis, in the case of Fink’s work [48] eliminating 93% of PPFs on average in the code they surveyed.

One disadvantage of current tracematch definitions is that they are insensitive to method return values. A typical tracematch looks much like a regular expression on methods only, so a pattern like `(hasNext next)* next` against the Iterator interface in Java would express illegally calling `next()` without calling `hasNext()` first. This pattern does not capture the fact that if `hasNext()` returns `false`, then the call to next is also illegal, and so will potentially miss illegal sequences of method calls as defined in the interface’s documentation.

By expressing illegal behaviour in the form of a regular expression, tracematches are not an ideal means of documenting the intended behaviour and behavioural restrictions of a stateful interface. However, it should be possible to generate the set of tracematches required to ensure the safe usage of a stateful interface from an alternative notation, meaning that the approach is still useful for verification.

2.5.1 Clara

The Clara framework [21] provides the ability to define tracematch-like runtime monitoring which can be partially checked through static analysis, and inject runtime checks where the analysis fails. Clara uses a flat finite state machine DSL to declare the available methods in each state, and the effect the method call has. An example of this adapted from Bodden's work [21] is shown in Listing 2.15, which declares the state machine for a `Connection` object as a *dependent state machine* between lines Line 2 and Line 10. The rest of the code declares the behaviour of a runtime monitor which maintains a list of closed connections which have not been garbage collected, and raises an error if an attempt is made to write to a closed connection. The *final* state of the declared state machine is where a meaningful action (in this case, the error) would occur — the static analysis attempts to find all places where a transition to `s2` would occur.

Tracematches can be converted into Clara dependent state machines, if desired, though I believe Clara dependent state machines are clearer than tracematches for all but the simplest of models. Clara lacks the ability to specify conditional transitions in its state machines, therefore cannot provide any additional modelling precision over tracematches.

2.5.2 Protocols involving multiple objects

Tracematches are able to track sequences of method calls that occur in *groups* of related objects [60,105], something which a system such as Plural cannot. One example of the utility of this feature is in monitoring the relationship between an `Iterator` and the collection it is iterating over. In Java, an iterator over a collection c cannot be used if c is modified — any operation which modifies c effectively invalidates the iterator. Any calls to an iterator after the modification occurs may throw a `ConcurrentModificationException`. Similarly, one cannot change an object that has been added to a `HashSet` in a way that would change its hash code without first removing it from the collection. Such relationships can be quite complex, and Bodden's PhD thesis [19] provides many examples.

Such multi-object constraints are similar to multi-party session types, and present a significant challenge to typestate models which focus on individual objects. Static analysis of

```
aspect ConnectionClosed {
dependency {
        close, write, reconnect;

        initial s0: write -> s0,
                reconnect -> s0,
                close -> s1;
            s1: close -> s1,
                write -> s2;
      final s2: write -> s2;
    }

    Set closed = new WeakIdentityHashSet();

    dependent after close(Connection c) returning:
            call(* Connection.close()) && target(c) {
        closed.add(c);
    }

    dependent after reconnect(Connection c) returning:
            call(* Connection.reconnect()) && target(c) {
        closed.remove(c);
    }

    dependent after write(Connection c) returning:
            call(* Connection.write(..)) && target(c) {
        if(closed.contains(c))
            error("May not write to " + c + "as it is closed");
    }
}
```

Listing 2.15: A Clara aspect that monitors for attempts to write to closed connections

inter-object dependencies in the context of object oriented frameworks such as Spring have been studied by Jaspan [79].

2.6 Contracts

The interfaces we define in order to achieve modularity, encapsulation and reuse can be thought of as a form of contract between a client and an implementation. The contract specifies what is required of both parties: what the client can request, and what the implementation must provide.

Interface declarations in most object oriented languages are very weak contracts — they specify only what methods are available, the number of parameters to those methods, the types of those parameters and the type of the return value. Stronger contracts are often desired but a lack of support for the desired constraints results in many of these constraints being documented informally on the contract. Both the client and the implementation must then rely on this informal agreement being honoured, or write defensive code to detect contract violations.

The constraints which are typically documented informally in Java fall into the following categories:

- Parameter requirements — a method may require that a parameter is not null, or that it is strictly positive. A language may support the specification of such restrictions through scalar subtypes, such as in Ada [142], but often more complex *composite* constraints which express relationships between parameters are often required. For example, the `subList(from, to)` method of the `List` type in Java requires that $0 \leq \mathit{from} \leq \mathit{to} < \mathit{length}()$.
- Return guarantees — a method may guarantee that a return value will be non-null, or that a return value will be related to a parameter in some way.
- Aliasing restrictions. For instance, if an element is added to a `HashSet` in Java, it is expected that the value returned by `hashCode()` on that object should remain constant. In practice this translates into a requirement that the client ceases to use the object — the `HashSet` claims exclusive ownership of the object.
- Externalised state — A method such as `length()` may provide some information which provides bounds for the parameters of other methods, or be used to express typestate-like constraints on method availability.

Such constraints can typically be expressed as *pre-* and *post-conditions*, the theoretical foundations of which were established in Floyd-Hoare logic [49, 67]. Eiffel [94] pioneered the concept of formal pre- and post-condition based contracts in APIs, with combined static and dynamic analysis for enforcement.

Extensions and external tools have been developed for popular languages such as Java and C# which do not directly support contracts. These tools typically provide their own *contract languages* in the form of embedded DSLs or APIs which are interpreted by static or dynamic analyses. Code Contracts in .NET 4 [47,95] and ESC/Java2 [30] are two notable tools which provide pre- and post-condition based specification of contracts. Both can be used to express some primitive typestate-like constraints.

Code Contracts are the direct descendant of the Spec# research language [10, 11] which aimed to provide statically enforced API contracts. Spec# extended the C# language, while Code Contracts are provided as an API in order to allow straightforward integration with legacy code, and can be used in a variety of .NET Common Language Runtime (CLR) languages such as C#, F# and Visual Basic. Code contracts can be easily enforced at runtime, though static analysis of code contracts has also been studied [46].

Code Contracts in .NET 4 allow the specification of pre-conditions on methods that require another *pure* method must return a certain value. A pure method is one which does not modify its associated object in any way. Dependencies on pure methods are useful for specifying that the parameter to `get` on an indexed collection must be within the legal index range. However, it may also be used to express typestate constraints through the use of dynamic state test methods, such as `isEmpty()` or `isOpen()`.

The limitation to using "pure" methods in such contract definitions is important: The value of a pure method can be derived without changing the state of the object.

An example of this is shown in Listing 2.16, which demonstrates a code contract definition of an iterator. Pre- and post-conditions are specified as method calls within the body of the associated method; as interfaces do not carry implementations, a surrogate abstract class is used (`IIContract`).

Not all stateful behaviour patterns can be correctly captured by this style of specification. The inability to express any dependency on a method having already been called in the client code and that it returned a particular value forces the introduction of new pure methods that would not normally exist in the interface. For example, the methods `hasCalledNext()` and `hasCalledRemove()` could be added to an iterator interface so that the precondition $hasCalledNext() \wedge \neg hasCalledRemove()$ can be specified on the `remove()` method. This places an additional burden on the implementer to provide these methods and correctly implement them.

```
[ContractClass(typeof(IIContract))]
interface IntegerIterator {
  [Pure] boolean hasNext();
  [Pure] boolean hasCalledNext();
  [Pure] boolean hasCalledRemove();
  int next();
  void remove();
}

[ContractClassFor(typeof(IntegerIterator))]
abstract class IIContract : IntegerIterator {
  boolean hasNext() {
    Contract.Ensures(false == hasCalledNext());
  }

  boolean hasCalledNext() {}
  boolean hasCalledRemove() {}

  int next() {
    Contract.Requires(true == hasNext());
    Contract.Ensures(true == hasCalledNext());
    Contract.Ensures(false == hasCalledRemove());
  }

  void remove() {
    Contract.Requires(true == hasCalledNext());
    Contract.Requires(false == hasCalledRemove());
    Contract.Ensures(true == hasCalledRemove());
  }
}
```

Listing 2.16: Code Contracts form of an Iterator

In contrast, the Java Modelling Language [84] allows for the specification of *model fields* and *ghost fields* [85, Section 2.2] which exist only within the model's abstraction of the type. This would allow for a field to be used to represent the typestate of an object and for methods to specify state requirements and guarantees in terms of this field. It is then the responsibility of enforcement strategy to maintain an accurate view of this model. A JML model of the `Iterator` interface is described in Cok's work [33], along with a discussion of ESC/Java2's ability to dynamically and statically enforce the defined contract.

2.7 Typestate inference

In order for a modular typestate analysis to be possible, it is necessary to precisely specify the *effect* that methods and functions have on their parameters, particularly in languages with dynamic dispatch where the real implementation of a method is difficult to derive statically. Type inference may offer the possibility to reduce the burden on the programmer.

There are two complementary typestate inference problems: inferring the state model of a type from its implementation, and inferring the requirements and effect of an expression which uses typestate-constrained values. Each is discussed separately below.

2.7.1 Typestate model inference

Writing contracts for interfaces and concrete types can be laborious and difficult to do, especially within a large legacy system — the expense of retrofitting such systems with contracts and typestate specifications is difficult to justify.

In a corpus of code in which the usage of an interface is widespread, it is possible to extract *sequences* for each usage and attempt to derive which sequences are the most common, which Ramanathan [128] refers to as *sequence mining*. Given a sufficiently large collection of such observed sequences a confidence heuristic can be used to decide when a sequence should be treated as a requirement of the protocol — for instance, in observing usage of an `Iterator` we may see `hasNext` preceding a call to `next` in the vast majority of code, and therefore conclude that this is part of the protocol description. Ramanathan recommends this approach as it allows protocols to be inferred in code bases with bugs: a sequence which does not fit the pattern of the majority *may* be a bug. Code in which `next` is called repeatedly without checking `hasNext` can be legitimate if the programmer is using some other piece of related information to control iteration, such as the length of the collection. Outliers such as this may give the false impression that `next` and `hasNext` are independent in general; a user-set confidence threshold allows for experimentation and identification of such outliers, and for the programmer to apply judgement with the assistance of the tool.

Logical pre- and post-conditions can also be mined during the same analysis, in order to determine whether parameters should be positive or related in some manner, by observing defensive code in clients and implementations.

Empirical results of this approach were encouraging — the generated contracts against the 242 library procedures in OpenSSH were 77% accurate using a confidence threshold of 100% in their algorithm. This figure is based on manual inspection of the generated contracts and comparing them to the informal documentation and comments in the code. The remaining 23% of incorrect specifications were composed of 5% false positives (contracts which are overly restrictive), 14% false negatives (contracts did not include restrictions that were specified in the documentation) and 4% unverifiable (procedures were not documented, so the intent of the authors could not be determined).

This positive result also demonstrates that while much can be inferred automatically, a programmer must still evaluate the result. The ability of the approach to draw attention to specific outliers helps considerably. Adding domain specific knowledge and heuristics to the analysis, such as in Kremenek's work [83], can further improve the initial accuracy of the analysis.

A broader set of sequences can be derived for such an analysis by observing a program's execution at runtime, as seen in the work of Ernst [45] and Yang [158]. Such analyses also benefit from the availability of additional information that can be very difficult to infer statically, such as exact parameter values, infrequently visited branches of conditional code, and the interaction of threads in concurrent code.

2.7.2 Requirement and effect inference using constraints

Given an expression which interacts with a potentially typestate-constrained object, we may wish to infer what the expression demands of the object. This is the essence of *type inference*, wherein we wish to derive some *principal type* for the expression. If no assumptions are made about the environment in which the term is evaluated, then what we desire is a *principal typing* [80, 149], if one exists.

Damas-Milner type inference [35], and algorithm $\mathcal{W}$ for the ML family of languages in particular, have provided the foundation for most theoretical work on type inference. This approach is fundamentally oriented around *constraint generation* based on the semantics of terms in a host language, followed by *constraint solving*. Types in ML are either *atomic*, such as $\mathbf{Bool}$ or $\mathbf{Unit}$, or *structural*, meaning they are composed of other types. Types are equal if they are *structurally equivalent*, meaning meaning $T = V \to W$ only if $T = X \to Y$ where $X = V$ and $W = Y$.

The constraints generated for ML-like languages are equalities between type expressions which include *type variables*. As equality entails structural equivalence in ML, such constraints are solved by *first order unification* which builds *equivalence classes* of variables and detects violations of structural equivalence (i.e. $\alpha = \mathbf{Bool} \wedge \alpha = \beta \rightarrow \gamma$).

The first algorithm for first-order unification is attributed to Robinson [133], with alternatives later derived which were concerned with efficiency through the use of trees of multi-equations [91]. Rather than representing the constraint $\alpha_1 = \alpha_2 \wedge \alpha_1 = \alpha_3 \rightarrow \alpha_4$ as a set of binary equivalences, one can represent it as a set of *equivalence classes* using a data structure optimised for fast *union* of classes and determining to which class an element belongs (the *find* operation). An efficient union-find data structure [53] is key to the efficient implementation of first-order unification, with the most common union-find structure and algorithm attributed to Tarjan [144].

In languages with subtyping, the constraints generated are *inequalities* rather than equalities. Solving such constraints involves deriving type *bounds* on variables. The ability to solve such constraints is determined by the nature of the subtyping relation.

Object oriented languages typically employ either *structural* or *nominal* subtyping, though both can co-exist in a language [88]. Structural subtying is where $T <: U$ only if T and U have the same shape (e.g. are both functions of the same arity) and their sub-components are also in the subtyping relation, in either a co- or contra-variant manner.

In function types, this typically entails that the functions have the same number of arguments, that the arguments types are contra-variant and the return types are covariant, such that $T_1 \rightarrow U_1 <: T_2 \rightarrow U_2$ only if $T_2 <: T_1$ and $U_1 <: U_2$. For objects, $O_1 <: O_2$ only if the method set of O_2 is a subset of O_1 and the methods are related in a covariant fashion.

Nominal subtyping is where the subtyping relation is explicitly defined by the programmer, typically through declarations such as $T\ extends\ U$ or $T\ implements\ U$. Nominal subtyping entails structural subtyping in Java-like languages.

The first algorithm for type inference in a language with *atomic subtyping* is attributed to Mitchell [99], which he later extended to structural subtyping [100], concurrently with Fuh and Mishra [50, 51]. These algorithms require that types are of *finite depth*, which excludes recursive types. Simonet later provided a more efficient solver [138] which is within a factor of three slower than good first-order unification algorithms; this was measured by defining subtyping to be structural equivalence in an ML-like language: $T <: U \iff T = U$.

Subtyping is defined for session types in terms of a simulation relation [56], and typestate can be viewed in fundamentally the same way. Type inference has been applied to session types in the MOOSE language [41], where sessions are linear and subtyping is not considered.

An algorithm for session type inference in a linearly typed language which exploits the control structure of a term was presented by Collingbourne [34]. This algorithm is presented without proof and is known to fail if procedures are introduced.

No work exists where a constraint generation and solving approach has been used to infer principal types in a language with typestate. The Anek tool [14] infers the requirements of terms in Plural using probabilistic and heuristic techniques, but does not constitute a formal type inference algorithm.

2.8 Language usability & human factors

Programming languages should fundamentally be about the *programmer* rather than the computer. Programming language designers, however, often appear to be oblivious to this basic requirement. Language designers frequently make claims about a feature of their language being desirable and an improvement over existing work, but rarely provide anything other than anecdotal evidence to support such claims, as illustrated in a rather scathing survey of language claims by Markstrum [90]:

> *One aspect of the studied papers was consistent. The designers believe that their own opinions weigh as much as, if not more than, any rigorous survey or user study.*

The psychology of programming is an active research area, though this encompasses a great deal of scope: software process research, the impact of working environments and culture, tools such as integrated development environments, how people learn to program, methods of teaching programming, and so on.

Work which focuses specifically on the impact of syntax and specific language features is rare. Green's cognitive dimensions of notations [63] attempts to define a method by which a syntax should be evaluated. An algorithm can be expressed in many ways, even within one language, but the *notation* strongly determines the ease with which a concept can be expressed. A notation can be measured in terms of a set of properties, or *dimensions*. Unfortunately, defining orthogonal dimensions for measuring notations is difficult, such that independent choices about notation design could be made along each dimension.

Greene also asserts that *design is redesign*, which is to say that design is an iterative process by which a model is progressively refined. Designers are opportunistic, approaching a design problem from the most convenient angle to make some progress and then perhaps changing tack entirely based upon an idea or realisation triggered by this refinement.

As such, notations for expressing designs should be built with the awareness that frequent change and refinement is a necessary part of the process. A notation which expects perfect expression of a concept in one step, or in which making changes is difficult, is not fit-for-purpose.

Greene identified the following dimensions, among others, for design notations:

- *Viscosity* — in what ways does a notation resist or expedite change? This is clearly task dependent. A language which permits global variables strongly resists the renaming of those variables: changes must be made in multiple locations, and care must be taken by the programmer to only change the correct variables if variable names can be reused. Introducing a new function in an imperative language is relatively painless, in contrast.
- *Premature commitment* — in what ways does a notation force a programmer to make decisions before they are relevant? Early languages which required explicit line numbering forced a programmer to guess how likely they were to change a section of a program, with a costly (viscous) change required if they guessed wrong. A less extreme example that exists in Java is requiring a programmer to decide on the type of a variable as soon as they declare it. When initially sketching out the body of a method, they may prefer to leave variables types unspecified.
- *Role-expressiveness* — does the notation employed clearly distinguish different components? Programmers require "beacons" [28] which allow them to quickly navigate a notation, determining where one expression ends and another begins. Appropriate use of delimiting whitespace, keywords and typographical conventions all help in this goal.

Greene analyzed the Smalltalk-80 language in terms of these dimensions, in which he finds that the method of defining inheritance is viscous and forces premature commitment, and that the ability for methods and fields to have the same name can lead to poor role-expressivity. This does not necessarily entail that either decision is wrong; this would require careful consideration of the alternatives, or indicate areas where language-aware tool support will help. Using a different font weight or colour to distinguish method names from variables, for instance, can improve role-expressivity.

Attempting to assess a notation against such dimensions for the most common and critical tasks that a user is likely to perform can help clarify the impact of design decisions in a notation. The idea of cognitive dimensions has been used to assess visual programming languages [64], theorem proving assistants [81], and the C# language during its initial design [32].

Brooks attempted to devise a theory of the process by which programmers approach a programming task [27] and derive the meaning of existing code [28]. Brook's concept of "beacons" has led to further work in understanding how programmers locate *concepts* in code [127], and how tools can be devised to help in this process.

The difficulties in demonstrating, empirically, the impact of a particular notational choice is summarised by Brooks as follows:

> *Even among programmers of very similar experience levels, differences of as much as 100 to 1 were found across programmers in the time taken to write a given program. Additionally, across problems constructed to be of similar difficulty, an individual programmer often displayed a six-fold difference in writing time.*

As such, quantitative studies of programmer behaviour are very difficult to interpret. Such studies are often very time consuming, as even small programming problems can take minutes to hours to complete.

Attempts have however been made to empirically study the effects of notation, with Gannon [55] studying such choices in an early imperative language, TOPPS. In this study, nine changes were made to the definition of the language, where these changes ranged from rules concerning semi-colon usage to evaluation order. The original language and the modified languages were directly compared.

The study identified the difficulty in compensating for the *learning effect* — programmers improve with experience, even over very short periods of time. The paper identified the common problem of confusing := (assignment) and = (equality). In Green's model, symbols which are very similar in this manner with radically different meaning constitute both a *discriminability* and an *action slips* problem — they are both easy to mis-read, and easy to mis-type. A simple solution is to use a different symbol entirely (Gannon proposed <− for assignment) or to disallow assignment in expressions. This problem persists even in contemporary languages, highlighting that language designers are slow to learn from the mistakes of the past.

There is disappointingly little work to be found in the literature that attempts to perform experiments similar to Gannon's work on TOPPS for contemporary languages, and none in the area of typestate modelling.

Recent work by Parnin has attempted to apply advances in cognitive neuroscience to understanding the challenges programmers face [115]. Perhaps unsurprisingly, and also identified by Brooks, the efficacy of our short- and long-term memory is very important to programming, and frequent interruptions are very damaging to our ability to retain a model of a program in short term memory. Few recommendations are offered from this nascent work that would influence language design, instead its focus is directed on tools and processes which help programmers to retain focus and augment their memory.

Parnin rather poignantly summarises the state of usability research in programming languages:

> *Nearly 40 years have passed since some of the earliest cognitive models of programmers have been proposed. Both the programming landscape and our understanding of the human brain have dramatically changed. Unfortunately, in the time since, the impact on practising programmers has been negligible; the predictive power nearly non-existent; and, our understanding of the mind furthered little beyond common sense.*

Chapter 3

Representing Typestate Constraints

Like the representation of any other formal set of rules, there are many different ways in which typestate constraints can be represented. Visual representations related to Harel's statecharts [66] are popular for the abstract representation of state machines, such as in the Unified Modelling Language [134, Chapter 6].

Graphical languages are not however "obviously" superior to a textual languages [62, 152]. Graphical models can be just as overwhelming as textual models — Both require careful use of "secondary notation" [117]. The positioning of symbols in a graphical model (such as a Harel statechart) can strongly influence the ability to understand it, especially if the positioning does not match the user's expectations. For instance, if the graph "flows" from right-to-left or bottom-to-top, this may confuse users who expect the reverse. From my own experience of working with graphical models of state machines, a significant amount of time can be spent attempting to find a workable layout for a state machine, and choices of good layouts is at least as subjective as the formatting of code.

In systems such as Plural and Fugue, the designers sought a way to easily integrate the formal representation of typestate constraints into the existing grammar of Java and C# respectively, using the support for arbitrary annotations. These property set based annotations have a very restrictive syntax, but are already supported by the integrated development environments and tools available for these languages. This pragmatic choice is suitable while experimenting but the syntactic restrictions arguably make models difficult to read and present a barrier to wider adoption.

A variety of different feature sets have also been tried in the design and implementation of typestate systems, though evaluation of which features end up being useful and justified is rarely considered. A holistic approach to the design of a typestate modelling language that can satisfy all of the following criteria would represent a novel and useful contribution:

- The underlying semantics of the language should have a feature set that is simultaneously large enough to model the most important and common typestate constraints, but small enough so as to be easy to learn.

- The language should be easy to adapt such that it can be integrated with commonly used object oriented languages such as Java or C#.
- It must be possible to implement a dynamic checker for the constraints the language can represent, without significant runtime overhead.
- It should at least be possible that the constraints can be statically checked.

The language devised to fit these criteria, named Hanoi, is presented in this chapter, with the implementation of a dynamic checker in Java presented in Chapter 4 and a user study of the usability of the language presented in Chapter 5.

3.1 A minimal feature set for typestate modelling

Thanks to the empirical study conducted by Beckman and Kim [13], common categories of typestate constraints that exist in real Java code have now been enumerated, and can be used as guidance in selecting what semantic features are truly necessary in a typestate modelling language.

Beckman's study involved the construction of a static analysis that inspected methods for defensive checks of field values that throw exceptions. Every defensive check found in this manner is recorded as a *protocol candidates* for manual inspection. By running the analysis over 16 open source applications which collectively defined 15000 classes, 2920 protocol candidates were found, from which 648 were found to be evidence of typestate constraints. These were further analysed for similarity and associated informal documentation in comments to devise a set of categories into which the majority of the typestate constraints fell. In order of frequency of occurrence, the categories identified were:

- *Initialisation*: A method (or sequence of methods) must be called before the main set of capabilities of the object are made available. Activation of the main set of capabilities may be conditional upon a return value, or the absence of an exception being thrown. initialisation is typically monotonic.
- *Deactivation*: Calling a particular method will effectively disable the object, removing the ability to use most or all of its capabilities. Methods such as `dispose` on Abstract Window Toolkit (AWT) widgets fit this pattern, as does the `close` method on IO streams.

- *Type Qualifier*: It is a common pattern for object types to be overly general, and dynamically specialised in response to the constructor used to initialise them. This is related to the initialisation pattern but specifically concerns constructors, and once the object is created the subset of methods made available from the general interface is fixed.

- *Dynamic preparation*: In order to call any method in method set $\overline{m}$, another method m' must be called first. While related to initialisation, the methods in $\overline{m}$ may not be made available permanently — repeated reinitialisation may be necessary. The `remove` method on iterator is an example of this pattern, where the `next` method must be called to enable it, and calling `remove` immediately disables this capability again.

- *Boundary*: A method m may only be invoked based on evidence derived from another method m'. `Iterator` is the most common example of this pattern, where `hasNext` is used to determine whether `next` can be called. `pop` on a stack type, or `get` on a list type (where the size must be known) are also examples.

- *Redundant operation*: A method m may only be called once. This is related to initialisation and deactivation, but specifically identifies the case where a method is self-disabling and not idempotent. The method `getResultSet` on the JDBC `Statement` object, used for interacting with an SQL database in Java, fits this pattern.

- *Domain mode*: The interface of an object has several sets of methods that are related to particular modes of operation. These modes can be enabled and disabled at will. This is typical for UI widgets, where input and focus can be enabled and disabled which consequently enables or disables methods relating to requesting focus or modifying the data bound to the widget.

- *Alternating*: Methods m and n must be called in an alternating fashion.

A related (and earlier) attempt at classifying typestate-like restrictions was undertaken by Dwyer, Avrunin and Corbett [44], which focused on *events* and event ordering rather than methods and objects. The most common pattern discovered by Dywer *et al.* was the *response* pattern, where one event must occur after another event within a specified scope — initialising and disposing of resources is a common example of such a pattern in languages without garbage collection and finalizers.

Common amongst many of these patterns is that return values and exceptions are important and unavoidable parts of the contract. The object moves to a new state synchronously in response to calling a method, but *which* state is often determined by the return value or the type of exception thrown. Booleans and enumerations are the most common return types used to determine which state is reached, followed by integer types. Where integers are used as a return value to indicate the state of an object, this would often be better represented using a boolean or enumeration, but integers are used for legacy or traditional presentation reasons. As an example, `size()` = 0 may indicate the object is in one state, while `size()` > 0 indicates another state. This is equivalent to `isEmpty` on many APIs, and programmers will often use the two interchangeably. It is important to be able to support either usage for this reason.

As the return value determines the successor state, this has consequences for the formal modelling of the object's semantics. Between the beginning of a method call and its return, the receiving object is effectively in an indeterminate state — consequently, defining the semantics of self calls and mutual recursion is problematic. One option is to simply ignore typestate constraints for self calls, and rely on an alias control strategy to ensure that "self" can always be distinguished from other references.

Constructors are clearly important in that they often determine the initial state of the object. Occasionally the exact values of the parameters passed to the constructor may also be relevant, as in the *type qualifier* category of behaviour. It is more common, however, for separate constructors to be used to distinguish initial state, or for explicit initialisation methods post-construction (the *initialisation* category) to be used for this purpose.

Additionally, it is often the case that a method or method set is enabled temporarily, without disabling any of the other methods which were already available. This provides a justification for the use of hierarchical finite state machines in modelling — duplicate definition of available methods and transitions can be avoided through the use of nested states. Concurrently available state machines are also potentially justified, where the object's behaviour has clear partitions and each part behaves independently, such as the `ResultSet` type in Java's SQL interface, studied in detail by Bierhoff [16]. However, in my experience such clear separation is rare, and parallel state machines can be encoded into a single state machine using superposition.

There are some notable examples of constraints which cannot be directly represented using a finite state machine (or, equivalently, be considered to be regular grammars). This is most commonly seen in the *boundary* category of behaviours. However, in all cases I have observed the interface of such types also includes a way to dynamically test what the current state of the object is, which allows the interface to be represented as a finite state machine. Interfaces which are not representable in this form are very inconvenient to work with in practice — interactions with the object must be very carefully controlled and aliasing is particularly difficult to reason about in such circumstances.

From the above, it is clear that a modelling language capable of handling the majority of typestate patterns, without being overly restrictive or verbose, should have the following features:

- Initial state selection based on constructor invocation should be part of the model.
- It should be possible to model transitions which are conditional upon return value, for the most common return value types used to indicate the successor state.
- Exceptions are just as important as return values to the correct expression of typestate, and so should have a similar level of support.
- Support for a hierarchical definition is justified, as in most cases a large subset of methods on an interface are always available.

While alias control is important for static analysis of typestate specifications, it is not necessarily something which is innately part of the typestate constraints on an object. The only methods which can be safely invoked when an object is shared are those which do not change the state of the object, or monotonically increase the size of the available method set. Militão's work on view-based typestate [96] provides the best approach I have seen for modelling typestate in a shared object context, where the interface of an object is explicitly partitioned based on the view (or role) that a client has of the object. This view partitioning can be emulated through objects with restricted view-like interfaces that indirectly interact with the shared, hidden intermediary.

As such, it was not deemed that modelling aliasing as part of the typestate constraints is strictly necessary — the majority of use cases can be handled without this.

3.2 Aspects of typestate modelling languages

In my opinion, there are fundamentally three aspects to the expression of typestate constraints:

1. The states in the model can be explicit or implicit. The names of the states themselves are not important to the underlying semantics of the model, but they often serve a useful purpose in adding a descriptive aspect to the model that helps a user to understand the purpose of each state (in much the same way as names assigned to interfaces are useful, but not strictly necessary, in an object oriented language with structural subtyping).

   In regular expression styles of specification, such as when tracematches are used, the states are implicit. Here, the individual states are not relevant to the model. Reporting of errors with such models typically requires that the tool is able to present a full sequence of method calls and highlight the point at which this diverged from the specification. In contrast, an explicit state model can simply report that an attempted method invocation is not possible in a named state.

2. Where a model has explicit state it can be state oriented or method oriented. The distinction between the two is in how the typestate constraints are grouped: if all constraints pertaining to an individual state are grouped together, the model is state oriented. If all constraints pertaining to an individual method are grouped together, then the model is method oriented.

   Each presentation potentially has advantages and disadvantages based on the task a programmer is performing. Where the programmer wishes to understand the overall structure of the model, a state oriented presentation is likely preferable. Where a programmer simply wishes to know whether a method can be called in a given state, a method oriented presentation is likely preferable — the programmer need not read the entire model to find the declaration of transitions related to a particular method, they can instead seek out the grouping of declarations for that method.

3. A model can be positive or negative in focus — it can focus on what is legal or what is illegal, respectively. In a typestate model where the majority of the methods in the interface are legal with few exceptions, a model which only explicitly describes illegal usage may be much more concise.

Based on these characterisations, both Plural and Fugue have explicit state, are method oriented and are positive. Plaid has explicit state, is state oriented and positive. Tracematches have implicit state and are negative.

3.3 The Hanoi language

Hanoi is a domain specific language designed to support the minimum feature set described in Section 3.1, and is intended primarily to work with Java's object model. Based on the characterisation of typestate modelling languages in Section 3.2, Hanoi has explicit states, is state oriented and positive.

There were a number of considerations that influenced the design of the language. Definitions should, first and foremost, be easy to read and understand as a single unit. Definitions should also be easy to create and modify, so as to encourage the usage of the system and iterative refinement of models. Using Hanoi should not require any changes to the Java language and it should be possible to retroactively define models for existing code, such as the Java SE APIs. Hanoi is primarily designed to allow for the implementation of a dynamic

```
 1 ACTIVE {
 2   NEXT_AVAILABLE {
 3     CAN_REMOVE_MIDDLE { remove() -> NEXT_AVAILABLE }
 4     next() -> CAN_REMOVE
 5   }
 6
 7   CAN_REMOVE {
 8     remove() -> ACTIVE
 9     hasNext() :: true -> CAN_REMOVE_MIDDLE
10   }
11
12   hasNext() :: true -> NEXT_AVAILABLE
13   hasNext() :: <other> -> <self>
14 }
```

Listing 3.1: Hanoi model for java.util.Iterator

checker, though the design is also careful to not include any features which would make an accurate static analysis infeasible.

3.4 An Introduction to Hanoi state machines

The formal grammar of the Hanoi language is shown in Figure 3.1, though it is easier to introduce by example. A Hanoi state model for the Java Iterator interface is shown in Listing 3.1.

Hanoi models are hierarchical, deterministic finite state machines and are bound to a single class or interface. The convention for binding a Hanoi model to a type, for instance an interface named `com.example.IExample`, one simply creates a file named `IExample.state` in directory `com/example` on the program's classpath. This approach does not require the original class' definition to be modified, allowing Hanoi models to be defined for JDK types such as `Iterator` easily.

The Hanoi language is structured such that states are declared as labelled braced blocks, where available methods and child states are declared within a block. Each model has one root state; in the `Iterator` model the root state is named `ACTIVE`. States inherit the transitions of their ancestors subject to the rules described in Section 3.5. The root state therefore declares the operations which are available in all states.

In the model for `Iterator`, the transition on Line 12 states "`hasNext()` can be called, and if it returns the value `true` then the iterator is now in state `NEXT_AVAILABLE`." Similarly, the transition within `NEXT_AVAILABLE` on line 4 states that "`next()` can be called, and the iterator will then be in state `CAN_REMOVE`" (regardless of return value, in this case). The `::` token can be read as 'returns', and `->` can be read as "transition to".

Line 13 demonstrates two special tokens, `<other>` and `<self>`. The condition type `<other>` states that if all other conditions specified for the method call do not match the returned value, then this condition will match and the associated transition will take place. This is similar to the "default" branch on switches in Java.

A transition to `<self>` means that the object will remain in the same state. On Line 13, the transition can be read as "if `hasNext()` returns a value which has not been matched, then stay in the same state". There is a subtle difference between this and `hasNext() :: <other> -> ACTIVE` — if one were to call hasNext() in the `CAN_REMOVE` state and it were to return false, the definition on Line 13 would leave the object in state `CAN_REMOVE` while the alternative will trigger a transition to `ACTIVE`, which is undesirable as it prevents a legal call to `remove()`.

In Hanoi, the convention is that if no condition is specified then we assume the condition is `<other>`. The transition, however, must always be specified. Values of boxed and unboxed primitives (booleans and numbers) may be used for conditions, as well as enumerations and the value `null`.

In addition to inheriting the transitions of a parent state, a child may also override these transitions subject to some restrictions. An example of overriding is shown on Line 9, where the transition for `hasNext()` defined on Line 12 is changed such that the result state will be `CAN_REMOVE_MIDDLE` instead of `NEXT_AVAILABLE`. This override exists to ensure that we do not lose the ability to call `remove()` if a call to `hasNext()` is made, for instance to ensure the simple method shown in Listing 3.2 is legal.

```java
void removeCenter(Iterator it) {
  boolean first = true;
  while(it.hasNext()) {
    it.next();
    if(!first && it.hasNext()) it.remove();
    first = false;
  }
}
```

Listing 3.2: Method which removes the middle elements of a collection

Listing 3.3 demonstrates some additional aspects of the Hanoi language. Where the type being modelled is a class rather than an interface, constructor transitions are specified that indicate the initial state of the object, as shown on Line 1 and Line 2. As constructors are fundamentally different to methods, in that calling a constructor creates an object rather than mutates an existing object, it was decided that constructor transitions should not appear in the bodies of the state declarations. Instead, they exist outside the state hierarchy, referring to it through a "where" clause. A constructor transition must be specified for each public constructor declared on a class. As shown in the model, it often makes sense for constructors

```
 1  new(InetSocketAddress) -> CONNECTED
 2  new() -> DISCONNECTED
 3  where
 4  CLOSED {
 5    DISCONNECTED {
 6      connect(InetSocketAddress) :: true -> READABLE
 7      connect(InetSocketAddress) :: false -> <self>
 8    }
 9    CONNECTED {
10      read() :: -1           -> <self>
11      read() :: <other>      -> <self>
12      read() !! IOException -> DISCONNECTED
13
14      write(int)                  -> <self>
15      write(int) !! IOException -> DISCONNECTED
16
17      disconnect() -> DISCONNECTED
18    }
19    close() -> CLOSED
20  }
```

Listing 3.3: A model for a simple TCP socket

to specify an initial state that is not the root state — here, the root state corresponds to the set of methods available when the socket has reached the end of its usable life, with no transitions out of this state. When a model is structured like this, at least one constructor must necessarily place the object in a more specific child state for it to be usable.

This model also illustrates *exception transitions*, shown on Line 12 and Line 15. When a method has a checked exception in its signature, the model must include a transition for when this exception is thrown. In this case, the model indicates (through `!!` token, read as "throws") that when an `IOException` is thrown that the socket has been disconnected.

3.4.1 Hanoi Annotations

While the Hanoi language as described above is the primary means of providing a typestate model in our system, we also support an annotation based approach similar to that found in Plural or Fugue. The annotation equivalent to the iterator model shown in Listing 3.1 is shown in Listing 3.4. This alternative means of specification was designed in order to illustrate that the underlying semantics of the Hanoi typestate model are independent of the mode of expression, and to allow for direct comparison of the two forms of specification in a user trial (discussed in Chapter 5).

The `@States` and `@Transitions` annotations are required due to the restrictions on annotations in Java — only one annotation of each type can be attached to a definition. The idiomatic work-around for this is to define an annotation like `@States` which takes an array

⟨*model*⟩ ::= [⟨*ctrans*⟩+ ‘`where`’] ⟨*state*⟩

⟨*ctrans*⟩ ::= ‘`new`’ ‘`(`’ ⟨*params*⟩ ‘`)`’ ‘`->`’ ID

⟨*state*⟩ ::= ID ‘{’ ⟨*state*⟩* ⟨*method*⟩* ‘}’

⟨*method*⟩ ::= ID ‘`(`’ [params] ‘`)`’ ⟨*trans*⟩?

⟨*params*⟩ ::= ⟨*type*⟩ (‘`,`’ ⟨*type*⟩)*

⟨*type*⟩ ::= ID (‘`.`’ ID)*

⟨*trans*⟩ ::= ⟨*condition*⟩? ‘`->`’ ⟨*target*⟩

⟨*condition*⟩ ::= ‘`::`’ ⟨*term*⟩ | ‘`!!`’ ID

⟨*term*⟩ ::= [⟨*numop*⟩] NUMBER | [‘=’] ID | ‘`<other>`’

⟨*numop*⟩ ::= ‘=’ | ‘<’ | ‘<=’ | ‘>=’ | ‘>’

⟨*target*⟩ ::= ID | ‘`<self>`’

Figure 3.1: The grammar of the Hanoi language

```java
@States({
  @State(name="ACTIVE"),
  @State(name="NEXT_AVAILABLE", parent="ACTIVE"),
  @State(name="CAN_REMOVE_MIDDLE", parent="NEXT_AVAILABLE"),
  @State(name="CAN_REMOVE", parent="ACTIVE")
})
public interface Iterator<T> {

  @Transitions({
   @Transition(from="CAN_REMOVE", to="CAN_REMOVE_MIDDLE",
               whenResult="true"),
   @Transition(from="ACTIVE", to="NEXT_AVAILABLE",
               whenResult="true"),
   @Transition(from="ACTIVE", to="<self>",
               whenResult="false")
  })
  boolean hasNext();

  @Transition(from="NEXT_AVAILABLE", to="CAN_REMOVE")
  T next();

  @Transitions({
    @Transition(from="CAN_REMOVE", to="ACTIVE"),
    @Transition(from="CAN_REMOVE_MIDDLE", to="NEXT_AVAILABLE")
  })
  void remove();
}
```

Listing 3.4: The iterator model expressed using Hanoi annotations

of `@State` annotations as the default parameter.

This syntax is somewhat cumbersome, and optimised for the convenience of tools that would consume the annotations rather than that of the programmer. An alternative would have been to use annotations that contain strings using a custom grammar:

```
@Transitions({
  "ACTIVE     -> NEXT_AVAILABLE    :: true",
  "ACTIVE     -> <self>            :: <other>",
  "CAN_REMOVE -> CAN_REMOVE_MIDDLE :: true"
})
boolean hasNext();
```

This is a closer match to the main Hanoi language, and arguably easier to read and work with as a programmer. However, this style was not adopted so as to allow for a comparison to be made between the Hanoi domain specific language and the style of specification in Plural / Fugue in Chapter 5.

3.4.2 Representing common patterns in Hanoi

The typestate patterns identified in Beckman's work [13] are easily represented in Hanoi models, with abstract examples for each pattern shown below:

Initialisation

This is easily expressed by allowing the initialisation and deactivation method(s) in the root state, with the initialised functionality of the object represented by a child state. If initialisation is an idempotent operation, the transition should be overridden to target `<self>` in the initialised state.

```
ROOT {
  initA() -> INITIALISED
  initB(int) :: true -> INITIALISED
  initB(int) :: <other> -> <self>

  INITIALISED {
    initA() -> <self>
    initB(int) -> <self>

    m() -> <self>
    // ...
  }
}
```

If multiple method calls are required to initialise, then this pattern can be modified using additional states to introduce the extra initialisation methods.

Deactivation

Similar to initialisation, the deactivation method(s) will typically be present in the root state, which only contains the methods which a deactivated object is capable of:

```
INACTIVE {
  deactivate() -> INACTIVE
  n() -> <self>

  ACTIVE {
    m() -> <self>
    // ...
  }
}
```

If the deactivation method(s) are not intended to be invoked more than once, then the model can be restructured as follows:

```
ROOT {
  ACTIVE {
    deactivate() -> INACTIVE
    m() -> <self>
  }

  INACTIVE { /* deliberately empty */ }

  n() -> <self>
}
```

Type Qualifier

Hanoi cannot capture type qualification based on the value of a parameter, but it can capture qualification based on the use of different constructors. this pattern is represented through the constructors of the object initialising the object into different initial states. Shared methods can easily be placed in a root state or parent state:

```
new(List) -> A
new(Set) -> B
ROOT {
  shared1() -> <self>
  shared2() -> <self>

  A {
    m() -> <self>
  }

  B {
    n() -> <self>
  }
}
```

Dynamic preparation

This is demonstrated in the restrictions on `remove` in `Iterator`, shown in Listing 3.1.

Boundary

This is demonstrated in the relationship between `hasNext` and `next` in Listing 3.1. In addition to boundary checks based on boolean return values, Hanoi's support for numeric return values is also useful for capturing boundary checks based on methods such as `size`:

```
ROOT {
  NOT_EMPTY {
    pop() -> MAYBE_EMPTY
  }

  MAYBE_EMPTY { }

  push(X) -> NOT_EMPTY

  size() :: >0 -> NOT_EMPTY
  size() :: <other> -> MAYBE_EMPTY
}
```

Redundant operation

The operation which cannot be invoked more than once can be placed in its own state, as shown in the following example where re-initialisation is disallowed:

```
new() -> NOT_INITIALISED
where
ROOT {
  NOT_INITIALISED {
    initA() -> INITIALISED
    initB() :: true -> INITIALISED
    initB() :: <other> -> <self>
  }

  INITIALISED {
    m() -> <self>
    // ...
  }
}
```

Domain mode

The methods controlling the selection of a mode can be placed in the root state, with transitions to substates containing the appropriate method sets. Where multiple modes can be active, the transition overriding rules of Hanoi unfortunately require some duplication in the model, as shown for states `A_AND_B` and `B_AND_A` in the following model:

```
ROOT {
  enableA() -> A
  enableB() -> B
  enableC() -> C

  A {
    A_AND_B {
      doY() -> <self>
      disableB() -> A
      disableA() -> B
    }

    doX() -> <self>
    enableB() -> B_AND_A
    disableA() -> ROOT
  }

  B {
    B_AND_A {
      doX() -> <self>
      disableA() -> B
      disableB() -> A
    }

    enableA() -> A_AND_B
    doY() -> <self>
    disableB() -> ROOT
  }

  C {
    doZ() -> <self>
    disableC() -> ROOT
  }
}
```

States `A_AND_B` and `B_AND_A` are effectively equivalent, but must be duplicated as the overrides of `enableB()` and `enableA()` must be covariant to the original target states. A more complex multiple inheritance based semantics for states would eliminate this but at the cost of losing the tree structured presentation of the model, and additional complexity which is not needed in most cases.

Alternating

This pattern is easily expressed with two states:

```
new() -> A
where
ROOT {
  A {
    m() -> B
  }

  B {
    n() -> A
  }
}
```

As demonstrated, Hanoi can represent most patterns without difficulty. Conditional transitions based upon passed parameters are not possible in the current model. Conceptually this would not be difficult to add, but I have observed very few circumstances under which this is essential for correctly modelling a type.

3.5 Semantics

A state S_1 is a *substate* of another state S_2, written $S_1 < S_2$ if $S_1 = S_2$ or S_1 is a descendant of S_2.

States inherit the transitions of their ancestors, which ensures that a state's available method set is a superset of its ancestor's method sets. Where transitions are overridden, it is required that the target state of the override is a substate of all the target states of the transitions that are overridden. As this implies, it is possible for a child transition to override multiple parent transitions simultaneously:

```
PARENT {
  a() :: <= 0 -> PARENT
  a() :: > 0 -> CHILD

  CHILD {
    a() :: >= 0 -> GCHILD

    GCHILD { ... }
  }
}
```

In this example, the transition override defined in state `CHILD` overrides both parent transition definitions, as it overlaps with the first transition condition on value `0` and with the entire range of the second transition. The override is legal in this case, as the new target `GCHILD` is a substate of the original targets, `PARENT` and `CHILD`.

The transition override restriction exist to ensure *safe substitution* of states for their ancestors, a property which is discussed further and proven in Section 3.5.1.

A transition may override more than one parent transition — for instance, a parent state may define two transitions for a `size` method for return values less than 0

The conditions specified on Hanoi models can be easily translated into either numeric intervals (i.e. "[0,10)") or finite sets of values from the return type of the method. If the return type is numeric (i.e. one of `byte`, `char`, `int`, `long`, `float`, `double` and the boxed object types thereof) then the legal set of condition operators are `<`, `<=`, `=`, `>` and `>=`. If the return type is a boolean, an enumeration or an object type which is not a boxed primitive, the only legal condition operator is =. For object types, the special value `null` may be matched against. The default operator is '=', such that "`x() :: 0 -> Y`" is equivalent to "`x() :: =0 -> Y`".

When a set of transitions is defined for a method, all possible return values must be covered by the conditions on those transitions. This includes the declared checked exceptions on a method — if a method throws an `IOException`, a transition must be declared for when this exception type is thrown. A transition declared for an exception type X also applies to all subtypes of X. For example, "`x() !! IOException -> Y`" would trigger a transition to `Y` if a `SocketException` were thrown, as `SocketException` is a subtype of `IOException`.

Exception transitions need not be declared for unchecked exception types (those which are subtypes of `RuntimeException` or `Error` in Java) — these are automatically treated as transitions to the root state. This design decision was made based upon the premise that if an

```
1  ROOT {
2    X { a() -> Y }
3    Y { n() :: >0 -> X }
4    a() -> X
5    m() :: >= 0 -> X
6    m() :: <= 0 -> Y
7  }
```

Listing 3.5: An Illegal Hanoi model

unchecked exception is thrown from an object then either a programmer error has occurred, or some serious or fatal system error has occurred that has likely not been handled within the object. Therefore the object is likely not in a consistent state, and the root state of the object provides the weakest guarantee on what the object is capable of. For this reason, it is often desirable to define models such that the root state is empty, where it is possible that an object could become disabled by such a failure. This default transition for unchecked exceptions can be overridden like any other transition, if this default behaviour is unsuitable.

Declared transitions cannot 'overlap', meaning that the set of return values matched by one transition must be disjoint from the set of return values matched by any other transition, unless one transition matches values which are all subtypes of the values matched by the other transition. These rules have some important consequences. The model shown in Listing 3.5 is illegal for a number of reasons:

- The transitions defined for method `m()` overlap on the value 0. If method `m()` were to return 0, it would be unclear whether the object should be in state X or state Y.
- The transition set for method `n()` is incomplete in state Y. If n returns an integer, it would be unclear what state the object is in if the method returns a value $v \leq 0$. While the documentation may indicate that method `n()` will never return such a value, Hanoi has no way of determining this through inspecting the return type of the method. If the Java language allowed the definition of scalar subtypes, the method could be formally specified as returning values in a specific range and the Hanoi model could exploit this additional information.
- The transition defined for method `a()` on Line 2 conflicts with the parent definition on Line 4: the target state Y is not a substate of state X. This, in turn, means that state X could not be safely substituted for the ROOT state, as the successor states after a call to `a()` offer inconsistent sets of methods.

Listing 3.6 demonstrates some legal instances of transition overriding:

- The transition override of method `a()` on line 4 is legal, as state Y is a substate of state X (as targeted on line 10).

```
 1  ROOT {
 2    X {
 3      Y { b() -> X }
 4      a() -> Y
 5      b() -> ROOT
 6      m() :: >0 -> Y
 7      c() :: <=0 -> Y
 8      c() !! Throwable -> Y
 9    }
10    a() -> X
11    m() -> <self>
12    c() :: <0 -> X
13    c() :: >= 0 -> ROOT
14    c() !! IOException -> ROOT
15    c() !! EOFException -> <self>
16  }
```

Listing 3.6: A Hanoi model with legal transition overriding

- The transition override of method `b()` on line 3 is legal, as state X is a substate of the `ROOT` state (as targeted on line 5).
- The transition override of method `m()` on line 6 is legal. The parent definition on line 10 has a transition to `self`, which is state X within the context of the definition on line 12. State Y is a substate of state X, therefore the definition is legal.
- The transition override of method `c()` on line 7 is legal. This transition overrides both of the transitions defined in the parent. The target state, Y, is a substate of both X and ROOT, therefore the override is legal.
- The exception transitions for method `c()` on lines 14 and 15 are legal, as `EOFException` is a subtype of `IOException` and the target state for when an `EOFException` is thrown is a substate of the target state for when an `IOException` is thrown.
- The exception transition for method `c()` declared on line 8 is legal as the target state Y is a substate of all the target states for thrown exceptions of type `Throwable` declared explicitly and implicitly in the parent state. `Throwable` is the supertype of all exception types, therefore the target state Y must be a substate of `ROOT`, which is the target state for thrown unchecked exceptions (implicitly) and `IOException`. Y must also be a substate of `<self>` which is the target state for when `EOFException` is thrown. In this context, `<self>` is `X`.

3.5.1 Formal definition

In order to formalise the semantics of Hanoi, a flat finite state machine formalism with a translation from hierarchical to flat models is sufficient.

Some simplifications are required in order to keep the formalisation of Hanoi concise. In the presentation of conditional transitions, we shall only consider types which can be easily mapped to integers — floating point values in particular shall not be considered further in this formalisation. Handling floating point numbers requires a more complex representation of conditions as bounded ranges. Bounded ranges are also more efficient, and so are used in the actual implementation of Hanoi — more details can be found on this approach by studying the source code.

Boundary value and condition translation

V shall denote the finite set of integer values in the Java language. Booleans, reference values, enums and `void` can trivially be mapped to V:

Original	Translated
Enum value `x`	`x.ordinal()`
`false` or `null` reference	0
`true` or non-null refernce	1
`void`	0

Conditions involving booleans, references and enums are rewritten as follows:

Original	Translated
`m() :: true -> X`	`m() :: =0 -> X`
`m() :: false -> X`	`m() :: =1 -> X`
`m() :: null -> X`	`m() :: =0 -> X`
`m() :: E.x -> X`	`m() :: =`n `-> X` where n = `x.ordinal`

A state machine formalism for typestate in Java

A typestate model for a Java class O is defined as the tuple $F_O = \langle C, M, R, \Sigma, \alpha, \Delta, \chi \rangle$, where:

- C is the set of constructor signatures in type O.
- M is the set of method signatures in type O.

- $R : M \to \mathcal{P}(V)$ is a total function which provides the set of possible return values for a method.
- Σ is the set of states in the model.
- $\alpha \subseteq C \times \Sigma$ is the relation which determines the initial state for a given constructor. An initial state function $\mathit{init}(C)$ may be extracted from α such that $\mathit{init}(C) = S$ when $(c, S) \in \alpha$. This requires that the initial state be deterministic (formalised below).
- $\Delta \subseteq \Sigma \times M \times V \times \Sigma$ is a relation which defines the state transitions in response method calls. A transition function $\mathit{next}(S, m, v)$ may be produced from Δ such that $(S, m, v) = S'$ when $(S, m, v, S') \in \Delta$. This requires that transitions are deterministic (formalised below).
- $\chi \subseteq \Sigma \times M \times E \times \Sigma$ is a relation which defines the state transitions in response to thrown exceptions. The set of exception types E is a tree, with a greatest element $\top_E$ (in Java, this is $\mathit{Throwable}$) and a parent relation $P_E \subseteq E \times E$ such that $(e, e') \in P_E$ when e is a child of e'. The subtyping relation for exceptions, $e \leq e$, is the reflexive and transitive closure of P_E.

  An exception transition function $\mathit{errnext}(S, m, e)$ may be produced from χ such that $\mathit{errnext}(S, m, e) = S'$ when $(S, m, e, S') \in \chi$. This requires that exception transitions are deterministic (formalised below).

For convenience, we may define the function $\mathit{methods}(F_O, S)$ which derives the set of methods available in state S:

$$\mathit{methods}(S) = \{m \mid \exists v, S'.(S, m, v, S') \in \Delta\}$$

In order for F_O to be considered a valid typestate model, the following conditions must be met:

- There must be a defined initial state for every constructor:
  $\forall c \in C. \exists S \in \Sigma \text{ such that } (C, S) \in \alpha.$
- If a method m is available in state S, then all possible return values of m must have a transition defined: $m \in \mathit{methods}(S) \implies \forall v \in R(m). \exists S' \text{ such that } (S, m, v, S') \in \Delta$.
- An exception transition for $\top_E$ must be defined for all methods which are legal in a state: $m \in \mathit{methods}(S) \implies \exists S'.(S, m, \top_E, S') \in \mathit{dom}(\chi)$.

$$C = \varnothing \quad M = \{h, n, r\} \quad \Sigma = \{A, NA, CR, CRM\}$$

$$R(h) = \{0,1\} \quad R(n) = \{0,1\} \quad R(r) = \{0\}$$

$$\begin{array}{rcl}
\Delta & = & \{(A,h,0,A),(A,h,1,NA)\} \\
& \cup & \{(NA,h,v,NA) \mid v \in R(h)\} \\
& \cup & \{(NA,n,v,CR) \mid v \in R(n)\} \\
& \cup & \{(CR,h,0,CR),(CR,h,1,CRM)\} \\
& \cup & \{(CR,r,v,A) \mid v \in R(r)\} \\
& \cup & \{(CRM,h,v,CRM) \mid v \in R(h)\} \\
& \cup & \{(CRM,n,v,CR) \mid v \in R(n)\} \\
& \cup & \{(CRM,r,v,NA) \mid v \in R(r)\} \\
\\
\chi & = & \{(S,m,\top_E,A) \mid S \in \Sigma, m \in \mathit{methods}(S)\}
\end{array}$$

Figure 3.2: The formalisation of the Hanoi model of `Iterator`. For brevity, method signatures are abbreviated such that h = `hasNext()`, n = `next()` and r = `remove()`, while for the state labels are abbreviated such that A = `ACTIVE`, NA = `NEXT_AVAILABLE`, CR = `CAN_REMOVE` and CRM = `CAN_REMOVE_MIDDLE`.

- The initial state for a constructor must be deterministic: $(C,S) \in \alpha \wedge (C,S') \in \alpha \implies S = S'$.
- Transitions must be deterministic: $(S,m,v,S') \in \Delta \wedge (S,m,v,S'') \in \Delta \implies S' = S''$.
- Exception transitions must be deterministic: $(S,m,e,S') \in \chi \wedge (S,m,e,S'') \in \chi \implies S' = S''$.

The satisfaction of these conditions shall be represented by the predicate $valid_f(F_O)$. State machines which satisfy $valid_f$ correspond to a simplified form of Hanoi without inheritance. An example of a state machine for `Iterator` is shown in Figure 3.2.

Subtyping in typestate

Given an class A whose behaviour is defined by $F_A = \langle C_A, M_A, R_A, \Sigma_A, \alpha_A, \Delta_A, \chi_A \rangle$, a value of of this class is necessarily in a defined state at any point in time: $a : A@S$ where $S \in \Sigma_A$ is the current state of the object. We shall refer to $A@S$ as the *typestate*, or simply *type* of a.

If we have an object value a_1 of type $A@S_1$, we say that it can be *safely substituted* for another value a_2 of type $A@S_2$ if any valid usage of a_2 is also a valid usage of a_1. In order for a_1 to be safely substitutable for a_2, we require that $A@S_1$ be a *behavioural subtype* of $A@S_2$.

We require a formal definition of subtyping. Informally, a behavioural subtype is capable of being used in any way that the parent type can be. As usage is controlled by a finite state machine, this essentially means *language inclusion* — whatever sequence of method calls the parent type would accept, the subtype must also accept. As return values are also relevant in our model, the strings in the language are in fact sequences of pairs of method labels and return values. These are formalised as *interaction traces*:

Definition 3.5.1 (Interaction Traces)**.** Let $I \subseteq M \times (V \cup E)$ be the set of *interactions*, which are method calls and the values they can return for an object type O. The set of *interaction traces* of $O@S$, referred to as $Tr(O@S)$, is defined inductively as:

$$\frac{}{\epsilon \in Tr(O@S)}\ \text{TR\_EMPTY} \qquad \frac{(S, m, v, S') \in \Delta \quad \delta \in Tr(O@S')}{(m, v).\delta \in Tr(O@S)}\ \text{TR\_PREFIX-A}$$

$$\frac{(S, m, e, S') \in \chi \quad \delta \in Tr(O@S')}{(m, e).\delta \in Tr(O@S)}\ \text{TR\_PREFIX-B}$$

▲

The *length* of an interaction trace (written $len(\delta)$) is the number of interactions it contains, where $len(\epsilon) = 0$ and $len(\,(m, v).\delta'\,) = 1 + len(\delta')$.

We can define behavioural subtyping in terms of trace inclusion, as follows:

Definition 3.5.2 (Behavioural Subtyping)**.** Let O_2 be a *nominal subtype* of O_1 in the Java type system. If $Tr(O_1@S_1) \subseteq Tr(O_2@S_2)$ then $O_2@S_2$ is a *behavioural subtype* of $O_1@S_1$, written as $O_2@S_2 \precsim O_1@S_1$. ▲

Where a state machine F_A is derived from a Hanoi model for type A, we wish to prove that $A@S_1$ <: $A@S_2$ when $S_1 < S_2$. First, we must define how F_A is derived from the Hanoi model.

3.5.2 Producing state machines from Hanoi models

Producing a state machine $F_O = \langle C, M, R, \Sigma, \alpha, \Delta, \chi \rangle$ from a Hanoi model of type O requires defining how to produce the α, Δ and χ relations from the model. The sets C, M, R are derived directly from the O, while the set Σ is simply the set of state labels in the Hanoi model of O.

The original model, as provided by the program, is first transformed to use numeric return values for all methods, and `<other>` conditions are made explicit by converting them into one or more conditions such that the entire return value range of the associated method is

```
type method_sig = string
type state_label = string
type java_ex = string

type java_info = {
  topEx : java_ex,
  parentExOf : java_ex -> (java_ex option)
}

datatype condition = LT | LTEQ | EQ | GTEQ | GT
datatype target = SELF | FIXED of state_label

type transition = {
  meth : string,
  cond : condition,
  bound : int,
  target : target
}

type ex_transition = {
  meth : string,
  ex : java_ex,
  target : target
}

type state = {
  label : state_label,
  transitions : transition list,
  exTransitions : ex_transition list
}

type method = {
  name : method_sig,
  retValues : int list
}

type model = {
  states : state list,
  methods : method list,
  root : state_label,
  parentOf : state_label -> (state_label option)
}

type call = method_sig * int
type gamma = (call * target) list
type delta = (state_label * call * state_label) list

type ex_call = method_sig * java_ex
type psi = (ex_call * target) list
type chi = (state_label * ex_call * state_label) list
```

Listing 3.7: The Standard ML data types for representing a Hanoi model and the production of Δ and χ relations

covered. This step is not presented here as it is straightforward and primarily involves the manipulation of Java types.

The α relation can also be derived trivially from the Hanoi model, and checked to ensure that it includes a transition for every constructor as required. The derivation of Δ and χ is more complex — both involve multiple steps, which shall be both informally described and formally defined as Standard ML functions. The Hanoi model shall be represented in SML as a value of type `model`, defined in Listing 3.7:

Producing a Δ relation for a Hanoi model

A Δ relation will be represented by the type `delta` in Listing 3.7. In order to produce a `delta`, we first first construct a precursor relation $\gamma \subseteq M \times V \times \Phi$ (where $\Phi = \Sigma \rightarrow \Sigma$) for each state in the model, represented as type `gamma`.

A γ relation is essentially a Δ relation for which transition targets have not yet been fixed. The "call" (a method signature and return value) is paired with a function $\phi \in \Phi$ which acts as a placeholder for the target state, primarily designed to abstract `<self>` transitions until later in the translation process. The placeholder function can either be the identity function ($\lambda x.x$), used for `<self>` transitions, or a constant function ($\lambda s.s'$), used for fixed targets.

The construction of a γ relation for a state can be achieved by concatenating γ relation fragments produced for each defined transition on a state. Producing a γ fragment for a transition is defined by `gammaForTransition`:

```
fun conditionFilter condition bound =
  case condition
  of LT   => (fn n => n <  bound)
   | LTEQ => (fn n => n <= bound)
   | EQ   => (fn n => n =  bound)
   | GTEQ => (fn n => n >= bound)
   | GT   => (fn n => n >  bound)

fun rangeOf methodName ({methods, ...} : model) =
  case List.find (fn {name, ...} => name = methodName) methods
  of SOME {retValues, ...} => retValues
   | NONE => raise InvalidModel "method␣undefined"

fun conditionMatches model {meth, cond, bound, target} =
  case List.filter (conditionFilter cond bound) (rangeOf meth model)
    of nil => raise InvalidModel "condition␣matches␣no␣values"
     | xs  => xs

fun gammaForTransition model transition =
  let
```

```
    val returnValues = conditionMatches model transition
    val {meth, target, ...} = transition
  in
    map (fn v => ((meth, v), target)) returnValues
  end
```

The function `gammaForTransition` extracts the set of return values that the transition condition matches (using `conditionMatches`), and associates each of these to the target placeholder for the transition.

Concatenating these fragments is defined by `gammaForState`:

```
fun isDeterministic nil = true
  | isDeterministic ((call,_) :: gamma) =
      not (isSome (List.find (fn (c,_) => c = call) gamma))
      andalso isDeterministic gamma

fun gammaForState model ({transitions=ts, ...} : state) =
  let val gamma = List.concat (map (gammaForTransition model) ts)
  in
    if isDeterministic gamma
    then gamma
    else raise InvalidModel "non-deterministic␣transition"
  end
```

Multiple targets the same call are detected by `isDeterministic` and the model rejected if they are found.

The γ relation produced by `gammaForState` is incomplete, as it does not include any transitions defined by its ancestors. We must "overlay" the child γ on the complete γ_p of its parent in order to produce the complete γ' for the child. Overlaying involves replacing all mappings in the parent where a target is defined in the child for the same call. Additionally, in order to ensure safe state substitution, it must be confirmed that the child's target will always be a substate of the parent's target. Any transitions defined in the parent for which there is no override in the child are included in the output γ' without modification. This step is defined by `overlayGamma`:

```
(* targetFor : gamma -> call -> target option *)
(* isSubstate : model -> state_label -> state_label -> bool *)

fun applyTarget SELF       s = s
  | applyTarget (FIXED s2) s = s2

fun overrideTarget model ({label, ...} : state) origTarget newTarget =
  let
    val origStateLabel = applyTarget origTarget label
    val newStateLabel = applyTarget newTarget label
  in
    if isSubstate model origStateLabel newStateLabel
    then newTarget
    else raise InvalidModel "invalid␣override"
  end

fun overrideMapping model currentState ((call, target), gamma) =
  let
    val baseTarget = getOpt (targetFor gamma call, target)
    val t = overrideTarget model currentState baseTarget target
    val gammaMinusCall = List.filter (fn (c,_) => c <> call) gamma
  in
    (call,target) :: gammaMinusCall
  end

fun overlayGamma model currentState base overrides =
  foldl (overrideMapping model currentState) base overrides
```

In order to produce the complete γ for each state, we "bubble up" the γ produced by `gammaForState`, overlaying it on the γ for each parent state to the root. This step is defined by the recursive function `extractGammas`:

```
(* childrenOf : model -> state -> state list *)

fun extractGammas model state =
  let
    val children = childrenOf model state
    val childGammas = List.concat (map (extractGammas model) children)
    val gamma = gammaForState model state
    fun overlay (ch, g) = (ch, overlayGamma model state gamma g)
  in
    (#label state, gamma) :: (map overlay childGammas)
  end
```

Each complete γ can then be checked for "gaps" — method calls for which one or more possible return values have no defined target. If any gaps exist, the model is invalid, as determined by `checkGamma`:

```
(* methodDomain : gamma -> method_sig list *)
(* callsOf : model -> method_sig -> call list *)

fun findGaps model gamma =
  let
    val methods = methodDomain gamma
    val allCalls = List.concat (map (callsOf model) methods)
    fun hasNoTarget call = not (isSome (targetFor gamma call))
  in
    List.filter hasNoTarget allCalls
  end

fun checkGamma model (stateLabel, gamma) =
  if (findGaps model gamma) = nil
  then (stateLabel, gamma)
  else raise InvalidModel "gaps␣in␣gamma"
```

A complete γ with no gaps can then be translated to a Δ relation fragment by applying the target placeholder function to the state the γ is produced for. These Δ fragments are then concatenated to produce the complete Δ relation for the model:

```
fun gammaToDelta (stateLabel, gamma) =
  map (fn (c,t) => (stateLabel, c, applyTarget t stateLabel)) gamma

fun extractDelta model : delta =
  let
    val gammasByState = extractGammas model (rootStateOf model)
    val checkedGammas = map (checkGamma model) gammasByState
  in
    List.concat (map gammaToDelta checkedGammas)
  end
```

Producing a χ relation from a Hanoi model

The production of a χ relation follows a similar process to the production of a Δ relation for a Hanoi model: A precursor relation $\psi \subseteq M \times E \times \Phi$ is built for each state in isolation, checking that only one target is defined for each method and exception pair.

The ψ for each state is overlaid on the complete ψ^p for its parent to produce the complete ψ for each state. The process of overlaying a partial ψ from a child on the complete ψ^p for a parent is more complex than for overlaying Δ fragments, as it must take into account the subtyping relation for exceptions.

First, the two ψ relations are combined into an intermediate form $\omega \subseteq \{0,1\} \times M \times E \times \Phi$, where transitions are tagged with their "source" — 1 for transitions from ψ^p and 0 for transitions from ψ.

Given two entries $t_1 = (v, m, e, \phi)$ and $t_2 = (v', m', e', \phi')$ from ω, we can order these such that:

$$\begin{aligned} m < m' &\implies t_1 < t_2 \\ m = m' \wedge e <^* e' &\implies t_1 < t_2 \\ m = m' \wedge e = e' \wedge v < v' &\implies t_1 < t_2 \end{aligned}$$

The ordering on method signatures can be arbitrary, and the ordering $<^*$ on exceptions can be any linear extension of the parent relation for exceptions. The ordering of targets is irrelevant as we are guaranteed by checking for non-deterministic transitions in each ψ that $(v, m, e, \phi) \in \omega \wedge (v, m, e, \phi') \in \omega \implies \phi = \phi'$. As such, $t < t'$ is a total ordering. Let $t \leq t'$ be the reflexive, transitive closure of this ordering.

The ω relation is processed from greatest element to least element according to this total ordering, in order to check and override the targets as necessary.

Let ω' be a processed fragment of ω and $t_n = (v, m, e, \phi) \in \omega$ be the next transition to be processed, in order to produce a new processed fragment ω''. We search ω' for a least upper bound t' of t where t' applies to the same method: $t' = (v', m', e', \phi')$ such that $m = m'$, $e \leq^* e'$ and $\forall t'' \in \omega'.t_n \leq t'' \implies t' \leq t''$. If t' does not exist, then t_n does not override any transitions and $\omega'' = \omega' + t_n$. If t' exists, then there are three cases to consider:

- $e = e'$ and $v < v'$, meaning t_n is a direct override of t'. If $\phi(S) \lessdot \phi'(S)$, then the override is valid and $\omega'' = \omega' - t' + t_n$ (in order to avoid introducing a non-deterministic transition for a call to method m throwing exception e). Otherwise, the override is invalid.

- $e < e'$ and $v \leq v'$, meaning t_n is an indirect override of t'. If $\phi(S) \lessdot \phi'(S)$, then the override is valid and $\omega'' = \omega' + t_n$. Otherwise, the override is invalid.

- $e < e'$ and $v' < v$. This case is more difficult to interpret — t is a parent transition that *may* override the transition t' from the child:

  - If $\phi(S) \lessdot \phi'(S)$, the parent defines a more specific target than the child, and $\omega'' = \omega' + t_n$.
  - If $\phi'(s) \lessdot \phi(S)$, the child defines a more specific target than the parent, but for a less specific exception type. This is a form of indirect overriding that is permitted for Hanoi. The transition t_n is simply ignored in this case, such that $\omega'' = \omega'$.

    An example of this behaviour can be seen in the following hypothetical model:

```
X {
  m() -> Y
  m() !! E1 -> X
  m() !! E3 -> Y

```

```
    Y {
      m() !! E2 -> Z
      Z {}
    }
  }
```

Let $E_3 < E_2 < E_1$. If an object is in state `Y` and method `m` is called with exception E_3 thrown, the target state is `Z` and not `Y`. An exception of type E_3 is also an exception of type E_2 through subsumption, so it is important that safe substitutability also apply to exception values in the interpretation of the model.

– If neither case applies, then the parent and child transitions are incompatible, and the override is invalid.

This logic is expressed by `overlayExTrans` and `overlayExTransKnown`, while the overall process of overlaying ψ on ψ^p is expressed by `overlayPsi`:

```
fun removeTrans omega meth ex =
  List.filter (fn (_,(c,_)) => c <> (meth,ex)) omega

fun overlayExTransKnown model sLabel t t' omega' =
  let
    val (v,((m,e),phi)) = t
    val (v',((m',e'),phi')) = t'
    val tapp = applyTarget phi sLabel
    val tapp' = applyTarget phi' sLabel
    val childSub = isSubstate model tapp' tapp
    val parentSub = isSubstate model tapp tapp'
  in
    case (e = e', v <= v', childSub, parentSub)
    of (true,  true,  true, _    ) => t :: (removeTrans omega' m' e')
     | (false, true,  true, _    ) => t :: omega'
     | (false, false, true, _    ) => t :: omega'
     | (false, false, false, true) => omega'
     | _                           => raise InvalidModel "bad␣override"
  end

fun overlayExTrans model javaInfo sLabel (t as (_,((m,e),_)), omega') =
  case omegaTransFor omega' javaInfo m e
  of NONE => t :: omega'
   | SOME t' => overlayExTransKnown model sLabel t t' omega'

fun overlayPsi model javaInfo csLabel parentPsi childPsi =
  let
    val omega = buildOmega parentPsi childPsi
    val omega' = foldl (overlayExTrans model javaInfo csLabel) nil omega
    fun omegaToPsi omega = map (fn (_,t) => t) omega
  in
    omegaToPsi omega'
  end
```

By “bubbling up” each constructed ψ in the same manner as performed for γ, we can construct the almost-complete ψ:

```
fun extractPsis model javaInfo (state : state) =
  let
    val cs = childrenOf model state
    val cPsis = List.concat (map (extractPsis model javaInfo) cs)
    val psi = psiForState state
    fun overlay (csLabel, childPsi) =
      (csLabel, overlayPsi model javaInfo csLabel psi childPsi)
  in
    (#label state, psi) :: (map overlay cPsis)
  end
```

In Hanoi, if no exception transition is specified for a method m then the transition `m() !! Throwable => ROOT` is introduced. This behaviour can be included in the model by creating a “default” ψ which includes a transition for all methods to the root state for exception $\top_E$, and overlaying all the ψ relations generated in the previous step onto this. Finally, the ψ relations can be converted into χ fragments and concatenated, as expressed in `extractChi`:

```
fun buildDefaultPsi (model : model) (javaInfo : java_info) =
  let
    val methodNames = map #name (#methods model)
    val rootEx = #topEx javaInfo
    val rootState = #root model
  in
    map (fn m => ((m, rootEx), FIXED rootState)) methodNames
  end

fun psiToChi (sLabel, psi) =
  map (fn ((m,e),t) => (sLabel, (m, e), applyTarget t sLabel)) psi

fun extractChi model (jvInfo : java_info) : chi =
  let
    val rootState = rootStateOf model
    val psisByState = extractPsis model jvInfo rootState
    val defaultPsi = buildDefaultPsi model jvInfo
    fun finalizer (s,psi) = (s, overlayPsi model jvInfo s defaultPsi psi)
    val finalPsis = map finalizer psisByState
  in
```

```
public class TransactionalMap<K,V>
             implements Transaction, Map<K,V> {
  public TransactionalMap() {/*...*/}
  // methods from Transaction:
  public void start() {/*...*/}
  public void commit() {/*...*/}
  public void rollback() {/*...*/}
  // methods from Map
  public V get(K key) {/*...*/}
  public V put(K key) {/*...*/}
  public void remove(K key) {/*...*/}
}
```

Listing 3.8: A Map interface with a transaction subtype

```
    List.concat (map psiToChi finalPsis)
  end
```

3.5.3 Behavioural subtyping in Java

To illustrate behavioural subtyping in Java, consider a simple transactional data structure such as `TransactionalMap` shown in Listing 3.8. The interface `Map` allows any ordering of calls on its methods, while `TransactionalMap` has restrictions as defined in Listing 3.9: a transaction must be started before the map can be manipulated. Our intuition here should be that `TransactionalMap@TRANSACTION` $\lesssim$ `Map@T`, which is true as the state `TRANSACTION` allows all possible sequences of calls allowed by `T` in `Map`. However, neither `ACTIVE` or `NO_TRANSACTION` satisfy this property. This reflects the intuition that if we have a method that expects a `Map` as a parameter, we can only safely give it a `TransactionalMap` instance if it is in state `TRANSACTION`, as otherwise an illegal method may be invoked.

3.6 A Cognitive Dimensions analysis of Hanoi

Green's cognitive dimensions, discussion in Section 2.8, present a useful framework in which to analyse the notational choices made in the Hanoi language. Each dimension presented in Green's seminal work [63] is considered separately below, comparing the DSL and annotations based approaches. The conclusions of this analysis are subjective and potentially biased by my own opinion of the two notations. An impartial evaluation of the dimensions could be derived by presenting Blackwell's cognitive dimensions questionnaire [17] to end users.

```
new() -> NO_TRANSACTION
where
ACTIVE {
  NO_TRANSACTION {
    start() -> TRANSACTION
  }
  TRANSACTION {
    get(K) -> <self>
    put(K) -> <self>
    remove(K) -> <self>
    commit() -> NO_TRANSACTION
    rollback() -> NO_TRANSACTION
  }
}
```

Listing 3.9: Hanoi model for TransactionalMap type in Listing 3.8

3.6.1 Hidden/Explicit Dependencies

In DSL models, dependencies between a state and its ancestors is only partially explicit — consistent indentation of the states can make the relationship apparent, similar to the nesting of blocks in C-like languages. This indicates that perhaps the indentation should be part of the syntax, demanding consistent application from the user to aid future readability. Such a choice would increase the viscosity of the state hierarchy.

Method overriding in both models is largely implicit: there is no notation to distinguish a fresh declaration of a transition from one which overrides a transition in a parent. In deeply nested hierarchies the override may be especially difficult to see.

Children implicitly inherit the legal transition set from their ancestors. This introduces a similar issue to OO languages where a small subclass can be deceptively simple. Implicit inheritance of transitions keeps the models small, but a deeply nested hierarchy may be difficult to interpret.

As DSL models are contained in a separate file from the class that is modelled, there is an implicit dependency on the type declarations of methods in the main class definition. Parameter types are made explicit in the DSL to distinguish overloading of methods, however return types are implicit. This may affect the user's ability to correctly specify return type conditions, as they will have to either remember the return types (putting a load on working memory) or look at the class definition (a change of context that may be disruptive).

In the annotation model, the transitivity of state inheritance is more implicit, as transitions are not structurally affiliated with the states they belong to. Additionally, no visual cues are given to indicate the relationship between states, requiring the user to build a mental model of this from pair-wise declarations of inheritance. The declaration of an `@States` annotation could be indented in a manner similar to the DSL to provide such visual cues, but this cannot be enforced within the Java grammar for annotations.

The spatial separation between the declaration of a state name and where it is used by a transition in an annotation model makes the purpose of a state more implicit. This may weaken the ability of the user to determine what as state is for, i.e. how it is distinguished from its parent states. Descriptive state names helps with this, but cannot be enforced either notation.

The existence of transition overrides is more explicit in annotation models, however the ordering of the overrides is implicit, requiring that the user remember the inheritance hierarchy. Ordering the declared transitions such that overrides appear after the overridden transition may help, but is not enforced by the notation.

3.6.2 Viscosity

In DSL models, the inheritance hierarchy of states is viscous, in a similar fashion to the nesting of control flow in methods. Changing the parent state requires the declaring block to be cut from inside the parent state declaration and moved into the new parent. The implicit inheritance of transitions from a parent complicates this process — it may be necessary to "pull down" inherited transitions into the state before moving it. This process is highly related to the viscosity of inheritance in Java. Refactoring tools can help with the process.

In the annotation model, the inheritance hierarchy is somewhat less viscous than in DSL models, as changing the parent of a state does not require structural changes. However, implicit inheritance of transitions can still introduce unintended consequences, and the separation of the state hierarchy's declaration from that of the transitions is likely to make this operation more viscous in an annotation model in practice.

The introduction of new child states in both models is fluid - the introduction of a child state does not change the parent's behaviour, so this can be done freely.

Changing transitions in a state is fluid when it has no overrides. However when this is not true, the operation is viscous, as the behaviour of child states may also need to be altered. The implicit nature of transition overriding complicates this process — a user must scan the definition of every child state for overrides, and decide on a case-by-case basis whether it must be altered.

When a class has a Hanoi model, adding methods to the class is more viscous. In addition to adding the method, the user must remember to add a transition for the method in at least one state of the model. The user may initially add a self-transition to the root state for the method, which is a low impact and fluid change, but represents a form of premature commitment.

3.6.3 Premature commitment

If the user does not define a Hanoi model for a type, this is equivalent to a fully-permissive model being defined. So, Hanoi does not present an immediate barrier to implementing code as the user would normally do. It can be introduced after the fact, when the requirements are better understood. Constructor transitions need not be specified in either notation - by default, an object is assumed to be in the root state after construction.

One slightly irritating aspect of the semantics of annotations in Java is the rule that more than one annotation of the same type cannot be attached to an entity (class, method, field or parameter). If more than one of the same type is required, then an aggregate annotation, such as `@Transitions` to hold an array of `@Transition` annotations, is required. This requires that the user guess how many transitions they are likely to need (i.e. is one enough), or prematurely commit to using an `@Transitions` annotation, which is syntactically ugly.

3.6.4 Role-expressiveness

The special case transition target of `<self>` is notationally distinguished from other user-declared states by the arrow brackets, which clarifies that "self" is not a user-defined state name.

In DSL models, the structure of the declaration of a transition as opposed to a state makes the distinction between these two concepts very explicit. Adopting a convention such as using all-caps names for states distinguishes them further from method names, but this is not enforced by the notation. Return value conditions and exception transitions are distinguished by different notation (:: for return value, !! for exception). The symbols are different enough to express the role clearly, though a user may forget the relative meaning. On most latin alphabet keyboards the symbols are sufficiently separated that mis-typing one for the other is unlikely, so they are not likely to be a source of *action-slips* errors.

In annotation models, The declarations of states and transitions are strongly distinguished by their position within the source code, as state declarations are annotations on the class body while transitions are annotations on the individual methods. The different transition types are distinguished by the annotation type (`@ExceptionTransition` and `@Transition`).

3.6.5 Hard Mental Operations

In both notations, deciding whether the space of possible return values is fully covered by a condition is difficult. As the return type is not part of the declaration in the DSL model, it is unclear whether a transition for `null` will be required without looking at the class or interface declaration. The proximity of this declaration in the annotation model makes this slightly easier. The use of numeric ranges is expressive, but their independent declaration makes gaps implicit.

Generally, deciding which transition applies when a method is called may be a difficult operation due to transition overriding, particularly when conditional transitions are used. The grouping of transitions around methods in the annotation model reduces the work required in finding overrides, but requires that the user remember the hierarchy. Conversely, the DSL model makes the task of determining hierarchy easy, but finding overrides more complex.

Until a user has some experience of modelling common patterns (alternating legal methods, dynamic tests, etc) it may not be particularly obvious how to encode a particular constraint in either model. This is not fundamentally different to any other programming task, however — programmers often rely on examples and existing code to determine the template for solving a problem.

In annotation models, determining the full set of methods available in a state requires that the user read the annotations of all methods, filtering irrelevant declarations to find those that match the current state. This also involves a knowledge of the inheritance hierarchy. This task is potentially easier in the DSL model, as structural beacons indicate the inheritance hierarchy and there is less syntactic clutter from the overhead of annotations.

3.7 Conclusion

The Hanoi language provides a simple, expressive means of defining typestate constraints for the Java language, and could be easily adapted for use in other similar languages. It has been demonstrated that Hanoi has a sufficient feature set to to express the most common and important typestate patterns observed in Java code as determined by Beckman [13]. The semantics of Hanoi's hierarchical finite state machine have been formalised and it has been proven that substates are substitutable for their ancestors, as one may intuitively expect.

The language does have limitations, however. Firstly, it does not attempt to model aliasing constraints on methods or parameters. Superficially, it may seem sufficient that a transition to `<self>` on a method would be enough to indicate that an object does not change state in response to a method call, and therefore would be safe to invoke through a shared reference. Behavioural subtyping and subsumption invalidate this assumption. For example, consider the following model for a class O:

```
A { m() -> A }
B { m() -> A    n() -> B }
```

An object of type $O@B$ is a subtype of $O@A$, so subsumption would allow it to be passed into a context requiring $O@A$. Within such a context, one may erroneously assume that m does not change the state of the object, when in fact it will, from B to A, removing the ability to invoke $n()$ in the future. As such, a mechanism to denote no *actual* state change in response to a method call is required. A variant of `<self>` which is defined such that state transitions are disallowed in any subtype. This would have the advantage of maintaining orthogonality between the typestate model and any alias control system. Such a separation requires further investigation to determine the impact on the semantics of the model and its practical utility compared to including a permissions system as part of the model.

An additional limitation is that Hanoi does not attempt to represent state change on parameters to methods. This is essential for the static enforcement of Hanoi models — as the implementation of the method is opaque, it is necessary to describe what the method may do its parameters, particularly where they are typestate constrained objects, in a manner that the type system can interpret as part of the method invocation.

Finally, as identified in Section 3.4.2, Hanoi cannot directly represent the "type qualification" pattern where an object's initial state after construction is determined by the value of a parameter passed to the constructor. The generalisation of this is to allow all transitions to be conditional on the values of parameters, such as defining different behaviour when a boolean flag or enum is passed as a parameter, resulting in significantly different behaviour during the execution of the method. Adding this capability would increase the complexity of Δ relation for a model, and should be possible to support in a dynamic hanoi checker. The utility of such a change would need to be investigated in more detail, as it would introduce significant additional complexity.

The limitations of Hanoi in relation to the rest of the thesis are discussed in Section 8.1.

Chapter 4

Dynamic Checking of Hanoi Models for Java

One of the considerations in the design of the Hanoi typestate modelling language was to allow for a practical dynamic checker to be implemented. When enforcing typestate constraints without language or tool support, additional code must be written which can obfuscate the real behaviour of methods and classes. As an example, consider the partial implementation of `Iterator` for an array in Listing 4.1 — the first four lines of the `next` method manually check whether the end of the array has been reached, and throws an appropriate exception (as documented in the contract for `Iterator`) if this is the case. Similarly, a boolean flag is maintained by `next` and `remove` to determine whether the `remove` method should be permitted or not. If the protocol for `Iterator` is obeyed by a client, then such checks are unnecessary and the bodies of `next` and `remove` would contain only the code that is necessary to provide the required functionality.

In more complex cases where much larger sets of methods are enabled and disabled in response to state changes, such checks are duplicated throughout the program code. Unless the externally relevant state of the object is stored as an enumeration, it must be inferred through inspection of the fields of the object as is done in our example, but with potentially many more fields and more complex conditions over them. Where interfaces have many implementations (such as `Iterator`), such checks must be duplicated through all of the implementations.

This is of course assuming a programmer even writes such defensive code — the overhead of implementing and testing the enforcement of constraints is both tedious and expensive, and is a cost that is repeatedly paid during maintenance of the software.

Regardless of the cost, the enforcement of typestate constraints can be important to the integrity of the entire system. Static checking of constraints is often preferable, if meaningful

```
class IntArrayIterator extends Iterator<Integer> {

  private int[] array;
  private int pos = 0;
  private boolean removeEnabled = false;

  public IntArrayIterator(int[] array) {
    this.array = array;
  }

  public boolean hasNext() { return pos < array.length }

  public Integer next() {
    if(pos >= array.length) {
      throw new NoSuchElementException("reached the end of the array");
    }

    removeEnabled = true;
    return array[pos++];
  }

  public void remove() {
    if(!removeEnabled) {
      throw new IllegalStateException("remove is not enabled");
    }

    removeEnabled = false;
    // ...
  }
}
```

Listing 4.1: Manual enforcement of Iterator constraints

```
 1  void addInt(List l) {
 2    l.add(1); // "raw" type treats type parameter as Object
 3  }
 4
 5  void main() {
 6    List<String> ls = new ArrayList<String>();
 7    addInt(ls);
 8    String s = ls.get(0);
 9    // ClassCastException is thrown
10  }
```

Listing 4.2: Invalid usage of a generic type in Java

error messages can be provided, as it gives the earliest possible warning of errors. However static analysis is often not sufficient to ensure safety in runtime systems which permit dynamic loading and reflection — it is possible for code to be dynamically linked which interacts with a typestate constrained object in an illegal manner. The relationship between the client code and implementation cannot necessarily be derived by static analysis. This problem is well known in Java where type parameters are not *reified*, meaning they are not stored in the runtime representation of a parameterised type. As a result, constraints relating to the type parameters cannot be enforced at runtime, which would allow an `Integer` to be added to a `List<String>`, as shown in Listing 4.2.

This code will compile, albeit with a warning on Line 1 due to the use of a "raw" generic type. Raw types treat their generic parameter as `Object`, which allows an integer to be inserted into a `List` of any type on Line 2. As a result, the attempt to extract the value as a `String` on Line 8 will fail at runtime with a `ClassCastException`. If the type parameter were reified and enforced, the attempt to add the value `1` to the collection on Line 2 would be rejected at runtime. The program would have therefore failed at the appropriate point, which would help the programmer to diagnose the issue.

The situation with unenforced typestate constraints is similar, in that the point at which a failure is detected is often not the same as the point at which the violation occurs. An object may operate in a state where its internal invariants are violated for some time before a more critical problem is detected by the runtime system.

In a language such as Java, static enforcement of typestate constraints (even as performed by an advanced system such as Plural) is likely to produce many false positives, to the point that the static analysis becomes an unacceptable source of noise for the programmer. Complementary static and dynamic analysis, as employed by recent efforts at the implementation of tracematches [19, 48], can provide better value for programmers in that "obvious" errors are caught early by the static analysis, while dynamic checking can be used to check potential errors the static analysis is less certain of, or to ensure safety generally.

4.1 The requirements of a practical dynamic checker

In order for a dynamic checker for Hanoi to be practical, it must meet the following requirements:

- The implementation of the dynamic checker cannot require any changes to the Java language. Changing the language may be acceptable for research projects, but prevents widespread adoption unless very carefully managed, as with the introduction of generics to Java 5 based on the ideas of Pizza [113].

  Finding a means to implement a dynamic checker for Hanoi that does not require changes to the compiler, runtime system or standard library will ensure that experimentation with existing code and systems is possible, rather than dealing exclusively with small artificial examples.

- It should be possible to integrate the dynamic analysis with minimal changes to the configuration of a system. As an example, requiring that modules be pre-processed in some way before loading into the runtime system, especially if this is not automatic, is likely to add friction to the process and be a barrier to adoption.

- The ability to configure the analysis to control which objects are monitored and which are not is desirable, if the analysis adds significant runtime overhead. This is particularly useful for testing and diagnosis.

- The ability to configure the analysis to control the type of enforcement is desirable. It may not be possible to perfectly model existing classes, especially if they cannot be modelled using a finite state machine. Reporting warnings to the log for possible violations is preferable to throwing runtime exceptions, where the model cannot accurately represent all legal usage.

- The analysis should not require the explicit cooperation of code which uses a typestate constrained object. Due to reflection in Java, it is possible for code to interact with objects it has no explicit knowledge of beyond some basic type information. Such code, especially code written prior to the introduction of Hanoi, is unlikely to have any knowledge of the typestate constraints on an object used through reflection. As such, the constraints must be enforced in a defensive manner, from the perspective of the typestate constrained object only.

  As an example, consider Listing 4.3, where an object is constructed using only a string representing a class name. When this object is used as a `Queue`, there is the potential on Line 6 that the call to `dequeue` could violate a typestate constraint. If the typestate

```
String queueImplClass = "...";
Class<?> cls = Class.forName(queueImplClass);
Object queueObj = cls.newInstance();
if(queueObj instanceof Queue) {
  Queue queue = (Queue) cls.newInstance();
  Object o = queue.dequeue();
}
```

Listing 4.3: Example of using a potentially typestate-constrained object via reflection

implementation strategy relied on injecting the constraint checks at the call site, it may not be able to determine whether this call should be verified.

4.2 Methods of dynamic checking

A number of options were considered and tested in the search for a practical means of dynamically checking Hanoi models. Each is described below, with advantages and disadvantages discussed.

4.2.1 Wrapper generation

Where a Hanoi model is provided for a Java interface, it is possible to generate code for a wrapper which implements this interface. This wrapper would take a "real" implementation as a parameter to the constructor, and potentially a known initial state. Invoking a method on the wrapper would then involve the following:

- Checking that the object is in an appropriate state. If so, invoke the method on the wrapped object. If not, take appropriate remedial action (typically, logging the error and / or throwing an exception).
- If an exception is thrown, change the known state based on the appropriate declared exception transition, then re-throw the exception.
- If a value is returned, change the known state based on the appropriate declared transition, and return the value.

Part of a wrapper for the `Iterator` interface is shown in Listing 4.4. The states from the Hanoi model are encoded as an enumeration type with a `substateOf` method (the implementation of which is omitted but straightforward) which would return `true` for `CAN_REMOVE_MIDDLE.substateOf(NEXT_AVAILABLE)` or

`CAN_REMOVE.substateOf(ACTIVE)` as expected. The body of `next` demonstrates the pattern of checking a state pre-condition, and the body of `hasNext` demonstrates the pattern of conditionally interpreting the return value. Both show the default handling of thrown exceptions, as the model does not declare any more specific handling.

Wrapper generation does not rate favourably against our criteria for a practical approach to dynamic checking:

- The explicit cooperation of client code is required — "real" implementations must be wrapped manually and then only the wrapped reference used.
- Types must be processed to generate the wrappers, and the wrapper code will become increasingly complicated as more useful diagnostic features are introduced beyond the bare minimum represented in Listing 4.4. This adds additional weight to the deployment in terms of the amount of generated code that must be included.
- Generation of wrappers against classes, rather than interfaces, is not possible in all circumstances. Wrappers for classes extend the class itself, which requires the introduction of object factories into the architecture of the system to make selective wrapping possible. Where the class in question has public final methods (i.e. those which cannot be overridden) the wrapper cannot be produced through conventional means.

Despite these disadvantages, wrapper generation can provide a relatively lightweight option for the enforcement of typestate constraints in a very selective way, which can be useful for debugging. There are other options with this advantage and fewer disadvantages, however.

4.2.2 Dynamic proxies

Java's runtime system makes it possible to implement an interface dynamically through the use of a *dynamic proxy*, which is an object that can handle method calls for a type which is unknown at compile time through the use of reflection.

Similar to wrapper generation, a "real" implementation of an interface can be passed to a dynamic proxy along with code that will be invoked on each method call. This can be leveraged to enforce the state pre-conditions for a Hanoi model, handle thrown exceptions and interpret return values for state transitions.

A simple API that uses dynamic proxies to enforce Hanoi models was implemented, and code using this API to check the usage of an `Iterator` is shown in Listing 4.5. The `StateInspectorFactory` can create a dynamic proxy for any interface through the `create` method, which takes the real implementation as the first argument and the class

```
public class IteratorEnforcer<T> implements Iterator<T> {

  private static enum State {
    ACTIVE, NEXT_AVAILABLE, CAN_REMOVE, CAN_REMOVE_MIDDLE;
    public boolean substateOf(State s) { /* ... */ }
  }

  private State state;
  private Iterator<T> wrapped;

  public IteratorEnforcer(Iterator<T> iter) {
    wrapped = iter;
    state = State.ACTIVE;
  }

  public boolean hasNext() {
    try {
      boolean result = wrapped.hasNext();
      if(result == true) {
        if(state.substateOf(State.CAN_REMOVE))
          state = State.CAN_REMOVE_MIDDLE;
        else if(state.substateOf(State.ACTIVE))
          state = State.NEXT_AVAILABLE;
      }

      return result;
    } catch(Throwable t) {
      state = State.ACTIVE;
      throw t;
    }
  }

  public T next() {
    if(!state.substateOf(State.NEXT_AVAILABLE))
      throw new IllegalStateException(
        "cannot invoke next() in state " + state);

    try {
      T result = wrapped.next();
      state = State.CAN_REMOVE;
      return result;
    } catch(Throwable t) {
      state = State.ACTIVE;
      throw t;
    }
  }

  public void remove() { /* ... */ }
}
```

Listing 4.4: A generated wrapper for the Iterator interface

```java
List<String> list = createList();

StateInspectorFactory factory = new StateInspectorFactory();
Iterator<String> iterator
    = factory.create(list.iterator(), Iterator.class);

while(iterator.hasNext()) {
    Object value = iterator.next();
    // this would be an illegal second call to next, if uncommented
    // iterator.next();
    iterator.remove();
}
```

Listing 4.5: Usage of the Hanoi API for dynamic proxies

object of the interface as the second argument. It can search for a Hanoi DSL model in an adjacent file or read an annotation based model directly out of the interface's bytecode. If no model is found, the object is left uninstrumented.

The created proxy dynamically interprets the Hanoi model against each call. This process is generally more expensive than the generated wrappers, as it is based entirely on reflection rather than on bespoke code for each interface. This additional runtime cost is a tradeoff against a more compact representation — the same code is used to implement the proxy for all types, and no processing of types is required prior to running the code.

Dynamic proxies are, overall, a better option than generated wrappers, but suffer the same primary drawbacks: explicit cooperation is required by code that uses types in constructing and using the wrappers, and working with concrete classes rather than interfaces is problematic. They are however very lightweight, do not require changes to the runtime system, and are easy to inject into specific modules of code where this is desirable — they are ideal for use on the boundaries between modules where strict enforcement of contracts is essential to the stability and security of a system.

4.2.3 AspectJ based solutions

AspectJ provides a powerful Aspect Oriented Programming (AOP) infrastructure for the Java programming language [82]. AOP involves the creation of *advice* (code fragments), which are *woven* into pre-existing code based on defined *join points*. The most common join points are entry to and exit from methods, either within the definition of the method itself or at the location from where the invocation occurs.

As AspectJ manipulates bytecode directly, it has a great deal of freedom in manipulating existing classes in a way that cannot be achieved using regular source code. The weaving process, where advice is injected into the bytecode of existing classes, can either be performed as a preprocessing step before the code is executed, or it can be done dynamically using a specialised class loader. The official Java Virtual Machine allows for the overriding of the standard class loader in this way, making it very easy to integrate AspectJ into the runtime infrastructure of a system.

This makes the use of AspectJ especially appealing in meeting the requirements set out in Section 4.1, particularly in that it may allow for the enforcement of typestate constraints without the explicit cooperation of client code.

AspectJ Tracematches

Tracematches are a special form of join-point that allow advice to be executed when a sequence of calls matching a regular expression is detected [4]. Significant effort has already been directed at optimising the execution of tracematches [7, 19]. Leveraging this work is attractive for performance reasons — where pre-processing of the source of an application before deployment is practical, the overhead of dynamic checking of typestate can be significantly reduced through the use of whole program static analysis.

Utilising tracematches to enforce Hanoi constraints requires that a Hanoi model be converted into a regular grammar where final calls are illegal, and that this grammar is then used to generate a regular expression.

As an example, consider a type with a *boundary* typestate pattern captured in the Hanoi model shown in Listing 4.6. Here a method `a()` can only be called once after `enableA()` has been called, after which it is disabled again. The same restriction applies to `b()` and `enableB()`. Both may be enabled at the same time.

A grammar which captures illegal uses of this type is shown in Figure 4.1. This can then be used to generate a regular expression for a tracematch, shown in Listing 4.7. Conversion between a finite state machine and a regular expression can, in the worst case, involve an exponential increase in size, as even a small grammar such as this demonstrates. The regular expression generated has a “legal prefix” repeated component from Line 15 to Line 30, which corresponds to all legal sequences which start and end in the `INACTIVE` state. This is followed by an “illegal suffix” from Line 31 to Line 46, which corresponds to all sequences which end with an illegal call to `a` or `b`. The longest part of this, from Line 35 to Line 45 contains all the legal sequences that repeatedly revisit the `BOTH` state before ending in an illegal repeated call to either `a` or `b`.

One disadvantage to this approach is that limited information is available for generating a useful error message to help with diagnosing the typestate violation. With a regular expression that captures all possible illegal sequences, we cannot isolate which particular pattern was detected. A collection of complementary regular expressions could be used which each

$$
\begin{aligned}
\langle \mathit{INACTIVE} \rangle ::= {} & \text{enableA } \langle A \rangle \\
\mid {} & \text{enableB } \langle B \rangle \\
\mid {} & \text{a} \\
\mid {} & \text{b}
\end{aligned}
$$

$$
\begin{aligned}
\langle A \rangle ::= {} & \text{enableA } \langle A \rangle \\
\mid {} & \text{enableB } \langle \mathit{BOTH} \rangle \\
\mid {} & \text{a } \langle \mathit{INACTIVE} \rangle \\
\mid {} & \text{b}
\end{aligned}
$$

$$
\begin{aligned}
\langle B \rangle ::= {} & \text{enableA } \langle \mathit{BOTH} \rangle \\
\mid {} & \text{enableB } \langle B \rangle \\
\mid {} & \text{a} \\
\mid {} & \text{b } \langle \mathit{INACTIVE} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{BOTH} \rangle ::= {} & \text{enableA } \langle \mathit{BOTH} \rangle \\
\mid {} & \text{enableB } \langle \mathit{BOTH} \rangle \\
\mid {} & \text{a } \langle B \rangle \\
\mid {} & \text{b } \langle A \rangle
\end{aligned}
$$

Figure 4.1: A regular grammar which captures violations of Listing 4.6

```
new() -> INACTIVE
where
ROOT {
  INACTIVE {
    enableA() -> A
    enableB() -> B
  }

  A {
    a() -> INACTIVE
    enableA() -> <self>
    enableB() -> BOTH
  }

  B {
    b() -> INACTIVE
    enableA() -> BOTH
    enableB() -> <self>
  }

  BOTH {
    a() -> B
    b() -> A
    enableA() -> <self>
    enableB() -> <self>
  }
}
```

Listing 4.6: A type with boundary checks for `a` and `b`

```
tracematch(BoundaryCheck bc) {
  sym enableA before:
    call( * BoundaryCheck.enableA())
    && target(bc);
  sym enableB before:
    call( * BoundaryCheck.enableB())
    && target(bc);
  sym a before:
    call( * BoundaryCheck.a())
    && target(bc);
  sym b before:
    call( * BoundaryCheck.b())
    && target(bc)

  (   enableA+ a
    | enableB+ b
    | (
          enableA+ enableB
        | enableB+ enableA
      )
      (   enableB
        | enableA
        | b enableA* enableB
        | a enableB* enableA
      )*
      (
          b enableA* a
        | a enableB* b
      )
  )*
  (   b
    | a
    | enableA+ b
    | enableB+ a
    | (   enableA+ enableB
        | enableB+ enableA
      )
      (   enableB
        | enableA
        | b enableA* enableB
        | a enableB* enableA
      )*
      (   b enableA* b
        | a enableB* a
      )
  )

  { // the advice to be executed
    throw new IllegalStateException("...")
  }
}
```

Listing 4.7: Tracematch for illegal usage of Listing 4.6

```
public class BoundedQueue<E> {
  public BoundedQueue(int size) {/*...*/}
  public BoundedQueue(E[] buf) {/*...*/}

  public boolean isEmpty() {/*...*/}
  public boolean isFull() {/*...*/}
  public E dequeue() {/*...*/}
  public void enqueue(E elem) {/*...*/}
  public E[] flush() {/*...*/}
}
```

Listing 4.8: A bounded queue data structure

identify a different illegal pattern — by generating separate regular expressions which correspond to reaching a particular state in the Hanoi model followed by an illegal method call, we could regain the ability to report an error like “attempt to call `a` in state `B`”. This would add additional runtime overhead however, as multiple tracematches must be checked against each typestate constrained object.

Another serious disadvantage is that the join-points used to define the symbols (between Line 2 and Line 13) used in the regular expressions cannot test the return value of methods. Consequently, Hanoi models with conditional transitions cannot be converted correctly into tracematches. The model may be approximated in some cases to provide incomplete but still potentially useful enforcement of constraints. As an example, consider the `BoundedQueue` type shown in Listing 4.8, with its Hanoi model shown in Listing 4.9. This type allows elements to be `enqueue`’d when it is not full, and `dequeue`’d when it is not empty. Similar to an `Iterator`, client code must check that the `isEmpty` or `isFull` methods return `false` before elements can be removed or added respectively.

With such a model, we can at least ensure that client code is calling `isEmpty` before calling `dequeue` or `isFull` before calling `enqueue`. The simplified model that enforces this constraint is shown in Listing 4.10. The generation of such a model can be achieved by creating a new state with an available method set equal to the union of the sets of each conditional target. This is likely to be very imprecise in general, but it may at least capture some obvious typestate violations. The tracematch generated for this simplified model is similar to that in Listing 4.7, with some added complexity due to the `flush` method.

Overall, the inability of tracematches to accurately capture the full semantics of Hanoi makes their use impractical in most cases — conditional transitions are very common and essential to the enforcement of typestate constraints.

```
new(int) -> ACTIVE
new(E[]) -> NOT_EMPTY
where
ACTIVE {
  NOT_EMPTY {
    NOT_EMPTY_NOT_FULL { enqueue(E) -> NOT_EMPTY }

    dequeue() -> NOT_FULL
    isFull() :: false -> NOT_FULL_NOT_EMPTY
  }
  NOT_FULL {
    NOT_FULL_NOT_EMPTY { dequeue() -> NOT_FULL }

    enqueue(E) -> NOT_EMPTY
    isEmpty() :: false -> NOT_EMPTY_NOT_FULL
  }

  isEmpty() :: false -> NOT_EMPTY
  isEmpty() -> <self>

  isFull() :: false -> NOT_FULL
  isFull() -> <self>

  flush() -> ACTIVE
}
```

Listing 4.9: The Hanoi model for BoundedQueue

Generation of monitoring aspects

Given that tracematches proved to be unsuitable, a more direct approach was attempted where aspect code would be generated from Hanoi models. This generator takes a set of JAR files as input and scans them for types which have associated Hanoi models in either DSL or annotation forms. An aspect is generated for each type with a model that will intercept all calls to constructors and public methods of that type. The aspect behaves in a similar fashion to the dynamic proxies described in Section 4.2.2 — calls are checked for legality against the currently known state and rejected if they are illegal.

Consider a Java program which is compiled (without AspectJ) into a single JAR file, named `main.jar`, with a main class `com.example.Main`. The standard mechanism for executing this program is shown on Line 43 of Listing 4.11 — this will execute the program without any dynamic checking of typestate constraints. In order to enable dynamic checking, the generated aspects are compiled and placed into a separate JAR file, which is added to the classpath of the program to be monitored. The standard Oracle JVM provides a means to execute additional code prior to the normal start sequence of the program using the `javaagent` flag — AspectJ provides a "load-time weaving agent" which can be used in this manner and weaves applicable advice into all loaded classes. The JVM invocation starting on line Line 38 of Listing 4.11 demonstrates this, where the compiled aspects are

```
new(int) -> ACTIVE
new(E[]) -> NOT_EMPTY
where
ROOT {

  ACTIVE {
    isEmpty() -> NOT_EMPTY
    isFull() -> NOT_FULL
    flush() -> ACTIVE
  }

  NOT_EMPTY {
    isEmpty() -> <self>
    isFull() -> NOT_EMPTY_NOT_FULL
    dequeue() -> ACTIVE
    flush() -> ACTIVE
  }

  NOT_FULL {
    isEmpty() -> NOT_EMPTY_NOT_FULL
    isFull() -> <self>
    enqueue() -> ACTIVE
    flush() -> ACTIVE
  }

  NOT_EMPTY_NOT_FULL {
    isEmpty() -> <self>
    isFull() -> <self>
    enqueue() -> NOT_EMPTY
    dequeue() -> NOT_FULL
    flush() -> ACTIVE
  }
}
```

Listing 4.10: The simplified and flattened Hanoi model for BoundedQueue

contained in `hanoi_aspects.jar`. The entire script provides a straightforward means to enable or disable dynamic checking using a command line flag (`-d` or `--dyncheck`).

Listing 4.12 shows a fragment of the generated aspect for the `BoundedQueue` type. The aspect stores a map of the known states of all objects of this type, and a re-entrant lock is used to ensure that this map can be updated safely in the presence of concurrency. Advice is generated for each public method in the type, such that the advice is executed immediately prior to the call to a `BoundedQueue` method.

Aspects can work with concrete types as they inject code directly into the existing implementation rather than wrapping it, therefore the explicit cooperation of client code is not required. Self calls are ignored (i.e. if a `BoundedQueue` instance were to invoke a method on itself) with the conditional on Line 13. The legality of the method call is checked on Line 20 against the current known state — if it is illegal, an exception is thrown, otherwise the method call is allowed to proceed on Line 31. The return value is then processed to determine what the new state of the object is after the call, before returning to the normal flow of execution.

Two separate exception handlers are required for `RuntimeException` and `Error`, which are the supertypes of all unchecked exceptions. It is not possible to unify these into a single handler for `Throwable` (the common supertype of both) as exceptions of this type cannot be rethrown safely — checked exceptions are descendants of `Throwable`, so this may represent an attempt to throw a checked exception which is not expected in the context where the advice is executing.

Instead of using a *call* based join point, the aspects could use an *execution* join point, which would mean that the advice would be injected into the implementation of the method itself rather than at the site of the call in the client code. This approach is better suited to concrete types and would avoid the need for a global state map as in the aspect below. Instead, a separate aspect instance could be created for each `BoundedQueue` instance to store the local state for that object. With load time weaving and a program which does not use reflection, the two approaches are equivalent. Where reflection is potentially used, an execution based join point is the better choice.

Generating aspects for modelled types and using AspectJ load time weaving provides the best fit to our requirements amongst all the options evaluated:

- This approach requires no changes to the language or runtime. The AspectJ project, with its strong emphasis on practicality, has ensured that aspects can be used in a very straightforward manner with a standard Java virtual machine. The additional configuration required to enable load time weaving is minimal, as shown in Listing 4.11.

```
#!/bin/sh
#
# Runs com.example.Main with dynamic checking if the
# -d/--dyncheck flag is set. Arguments to be passed to
# com.example.Main should be provided after a "--"
# argument separator.
#
# Run without monitoring and no args:
# > run.sh
# Run without monitoring and passing argument "-x":
# > run.sh -- -x
# Run with monitoring and passing argument "-x":
# > run.sh -d -- -x
# > run.sh --dyncheck -- -x

# whether or not to use dynamic checking
dyncheck=0

# use gnu-getopt to validate and sanitize arguments
ARGS=`gnu-getopt --long dyncheck --options d -- "$@"`
if [ $? -ne 0 ]; then
    echo "Invalid␣argument(s)" >&2
    exit 2
fi
eval set -- $ARGS

# process the arguments
while [ $# -gt 0 ]; do
    case "$1" in
    -d | --dyncheck) dyncheck=1;;
    --)              shift; break;;
    esac
    shift
done

if [ $dyncheck -eq 1 ]; then
    # run with monitoring
    java -javaagent:aspectjweaver.jar \
         -cp main.jar:hanoi.jar:hanoi_aspects.jar \
         com.example.Main $@
else
    # run without monitoring
    java -cp main.jar com.example.Main $@
fi
```

Listing 4.11: Enabling or disabling Hanoi dynamic checking using AspectJ

```
public aspect BoundedQueueUsageMonitor issingleton() {

  private final Logger log = // ...
  private final WeakHashMap<Object, IState> currentStates = // ...
  private final ReentrantReadWriteLock currentStatesLock = // ...
  private final StateModelRepository repo = // ...

  boolean around(): call(public boolean isEmpty()) && target(BoundedQueue
      ↪ ) {

    Object thisObj = thisJoinPoint.getThis();
    Object target = thisJoinPoint.getTarget();

    if(thisObj == target) {
      return proceed();
    }

    IState currentState = getState(target);
    Method m = BoundedQueue.class.getMethod("isEmpty");

    if(!currentState.isLegalCall(m)) {
      throw new IllegalStateException(
        "Attempt to call method " +
        ErrorUtils.buildSignature(m) +
        " on object of type " +
        target.getClass().getCanonicalName() +
        " in illegal state " +
        currentState.getName());
    }

    try {
      boolean result = proceed();
      processOutcome(target, m, result);
      return result;
    } catch(java.lang.RuntimeException e) {
      processOutcome(target, m, new ThrownExceptionWrapper(e));
      throw e;
    } catch(java.lang.Error e) {
      processOutcome(target, m, new ThrownExceptionWrapper(e));
      throw e;
    }
  }

  // ... monitoring for other methods ...
}
```

Listing 4.12: Part of the aspect generated for BoundedQueue

- It is feasible to control the type of enforcement through configuration. The prototype implementation of the aspect generator creates aspects which enforce fail fast behaviour by throwing an exception when an illegal method call is detected. It would not take significant additional effort to make this behaviour customisable, or configurable through properties passed to the program at startup.
- Models can be enforced without changes to the implementation or client code that interacts with a class or interface, due to the bytecode manipulation managed by AspectJ.
- The performance of the analysis is likely to be better than that of dynamic proxies, due to the approach not relying on reflection and instead on generated code which can be optimised by the virtual machine. Unfortunately, the performance is likely to be worse than the tracematch based approach, as tracematches can leverage static analysis of the code to avoid monitoring usage that can be proven to be safe. However, the generated aspects can enforce all of Hanoi's semantics, including conditional transitions, which the tracematch based approach cannot.

The primary disadvantage of this approach is that Hanoi modelled types must be preprocessed to generate and compile the aspects, and that the JAR containing these aspects must be added to the classpath. This requirement is not overly restrictive, but a solution without this step would be preferable.

Unified monitoring aspect

The generated aspects for each modelled type are structurally very similar. An attempt was made to determine whether a unified aspect could be defined which could monitor all modelled types, while retaining the practical properties of the individually generated aspects.

The product of this effort is shown in Listing 4.13. The key difference with this aspect from that shown in Listing 4.12 is that it relies upon an abstract pointcut which must be defined in a sub-aspect. This may be defined as `target(BoundedQueue)` to make the aspect effectively equivalent to Listing 4.12. The defined advice will apply to all public methods on a type it matches.

This approach poses some additional challenges:

- As previously discussed, the handling of exceptions thrown by the invoked method in an abstract manner is not possible through conventional means. A possible workaround when the code is being executed using the standard Sun/Oracle Java Virtual Machine is to obtain an instance of `sun.misc.Unsafe`, which contains the method `throwException`. This method allows exceptions of any type to be thrown, regardless of the declared set of exceptions in the context where it is called. Access to `sun.misc.Unsafe` requires the use of a deliberately convoluted process, and can be disabled entirely via a JVM's security policy configuration as it offers many dangerous, low-level operations.

- There is no single pointcut that can be defined for `hanoiType` that will work for all Hanoi types. One option is to target all types which have a special annotation: the pointcut `@target(HanoiModelled)` will match all types which have the annotation `@HanoiModelled`. This is acceptable for new types, but for code which cannot be modified to add this annotation a different approach is required.

  For standard library types, a pointcut which lists all the modelled types is possible: `target(Iterator) || target(InputStream) || ... .`
  This approach could be adopted for all libraries for which Hanoi models are desired but the original code cannot be modified.

The unified monitoring aspect is promising, but the need to define multiple sub-aspects to handle special cases means that it does not fully achieve the goal of a generic, configurationless approach — The sub-aspects must still be compiled and added to the runtime environment in the same way as the generated aspects.

4.2.4 Other AspectJ possibilities

AspectJ allows a number of alternative strategies to the implementation of monitoring aspects, which were not used in the current implementation but are mentioned here for completeness.

Inter-type declarations

The global map of objects to object states on Line 4 of Listing 4.12 is a potential performance bottleneck when a large set of monitored objects exists and multiple threads are in contention to read from or write to the map. This map can be eliminated by injecting fields to represent the current state into the monitored objects themselves, using AspectJ *inter-type declarations*. Inter-type declarations can only rely on static information however in deciding whether to weave the additional fields into a class — as such, inter-type declarations cannot be used to add a state field to types which have been modeled using Hanoi DSLs. One can however use an annotation (such as `@States`) as the trigger for inter-type declarations, as shown in Listing 4.14, where all types annotated with `@States` are modified to implement

```
public abstract aspect AbstractHanoiAspect {

    abstract pointcut hanoiType();

    Object around():
            hanoiType() && call(public * *(..))
            && !within(AbstractHanoiAspect+) {

        MethodSignature msig = (MethodSignature) thisJoinPoint.
          ↪ getSignature();
        Method methodCalled = msig.getMethod();
        Object target = thisJoinPoint.getTarget();
        IState currentState = getState(target);

        if(!currentState.isLegalCall(methodCalled)) {
            // process failure
        }

        try {
            Object result = proceed();
            processOutcome(target, methodCalled, result);
            return result;
        } catch(Throwable t) {
            ThrownExceptionWrapper wrapper =
                new ThrownExceptionWrapper(re);
            currentStates.put(target,
                currentState.getNextState(methodCalled, wrapper));
            getUnsafe().throwException(t);

            return null; // will not be reached
        }
    }

    private sun.misc.Unsafe getUnsafe()
    {
        try {
            Field field = sun.misc.Unsafe.class.getDeclaredField("
                ↪ theUnsafe");
            field.setAccessible(true);
            return (sun.misc.Unsafe)field.get(null);
        } catch (Exception ex) {
            throw new RuntimeException("can’t get Unsafe instance", ex);
        }
    }

    // ...
}
```

Listing 4.13: Part of the abstract aspect

```
interface StateTracking {}

public aspect InjectState {

  declare parents: (@States *) implements StateTracking;

  IState StateTracking.currentState;
}
```

Listing 4.14: Using inter-type declarations to inject state tracking fields into objects

---

the new interface type `StateTracking`, and all implementations of `StateTracking` are given an additional field `currentState`.

A tempting alternative solution would be to add a `currentState` field to all objects in Java, by using an inter-type declaration bound to the `Object` type. However, the `Object` type cannot be modified by AspectJ — a more specific subtype is required and therefore there is no mechanism to add a field to all objects regardless of type.

A better solution would be to augment the aspect generator to generate an additional aspect which modifies all types with Hanoi DSL models such that they implement `StateTracking`. Once this is done, the aspect in Listing 4.14 would then be able to inject the `currentState` field on all tracked objects.

Control-flow pointcuts

The self-call exclusion on Line 4 of Listing 4.12 cannot deal with indirect call loops, such as one might see with mutual recursion between two objects. A potential alternative is to use a control flow (`cflow`) based pointcut, which is sensitive to the entire call stack. Control flow pointcuts can be used to match all pointcuts which occur during another pointcut: for example, `cflow(call(public boolean isEmpty()) && target(Iterator))` will match a call to `isEmpty()` and all other pointcuts which occur while this method is executing. A `cflowbelow` pointcut will instead match the same set of pointcuts, minus the pointcut for the call to `isEmpty` itself.

By negating a `cflowbelow` pointcut, we can exclude all pointcuts that occur during another pointcut: `around(Iterator i): target(i) &&`
`call(public boolean hasNext()) &&`
`!cflowbelow(call(public boolean hasNext()))` will match any call to `hasNext` which has not itself been triggered, directly or indirectly, by any other method named `hasNext`. A more useful pointcut would be
`around(Iterator i): target(i) &&`
`call(public boolean hasNext()) && !cflowbelow(target(i))`

```
public aspect BeforeAfterAspect {

  pointcut isEmpty(BoundedQueue b):
    call(public boolean isEmpty()) && target(b);

  before(BoundedQueue b): isEmpty(b) {
    IState currentState = getState(b);
    Method m = BoundedQueue.class.getMethod("isEmpty");

    if(!currentState.isLegalCall(m)) {
      // throw IllegalStateException ...
    }
  }

  after(BoundedQueue b) returning (boolean result): isEmpty(b) {
    Method m = BoundedQueue.class.getMethod("isEmpty");
    processOutcome(b, m, result);
  }

  after(BoundedQueue b) throwing (Throwable t): isEmpty(b) {
    Method m = BoundedQueue.class.getMethod("isEmpty");
    processOutcome(b, m, new ThrownExceptionWrapper(t));
  }
}
```

Listing 4.15: Monitoring method calls using separate before and after advice

---

which would match all calls to `hasNext` which have not been invoked as a result of some other call to the same `Iterator` instance. This pointcut is not valid however: pointcuts which use negation cannot contain variables. The control-flow restriction `!cflowbelow(target(Iterator))` is valid, however it is often too restrictive as it is insensitive to the specific instance — it will exclude direct and indirect recursive calls as desired, but it will also exclude a call to iterator `i` as a result of some other iterator `j`.

Another possibility is to exclude all pointcuts reached as part of executing the monitoring advice, through the pointcut `!cflow(adviceexecution() && within(IteratorUsageMonitor))`. This too is overly restrictive for the same reason — it is insensitive to the identity of the objects and so would exclude monitoring a call to an `Iterator` instance from another instance.

Before and after pointcuts

Rather than intercepting and completely handling a method call using `around()` advice, another possibility is to split the handling into two related `before()` and `after()` handlers. This has the specific advantage of allowing the handling of thrown exceptions and normal return separately, as demonstrated in Listing 4.15, which is an adaptation of the original generated aspect shown in Listing 4.12.

This presentation is cleaner, as the `after()` advice does not need to rethrow the exception or process specific types of exception.

Aspect generation using MetaAspectJ

The current implementation of aspect generation manipulates string templates — a more maintainable approach would be to use Meta-AspectJ (MAJ) [160], which provides a mechanism for generating aspects in a flexible, type safe manner. MAJ provides a minimal extension to the syntax of Java to provide a convenient and type-safe syntax for meta-programming. A partial implementation of monitoring aspect generation is shown in Listing 4.16.

In Meta-AspectJ, fragments of code are embedded in generating expressions of form `` `[ <code> ] ``, which produce abstract syntax tree (AST) fragments that are typically assigned to variables of an "inferred" type, as seen on Line 9 and Line 37. AST fragments can be composed, typically by including a fragment in a larger fragment as seen on Line 15 — the fragment is referred to by its field name, prefixed by "#".

The Meta-AspectJ compiler is capable of determining the AST node type of generating expressions, and can statically enforce syntactically correct composition of aspects, which regular parameterised string templates cannot. As such, Meta-AspectJ is a better option for the implementation of any future production-ready version of the monitoring aspect generator.

4.3 Evaluating the overhead of dynamic checking

The overhead of the implemented dynamic checkers for Hanoi is important to the assessment of their practicality. Both the dynamic proxy based implementation and the AspectJ based implementation work by instrumenting every method call to a monitored object type, so it is desirable to determine the per-call overhead.

In order to do this, a simple benchmark was constructed using a type which implements the common *left fold* (also known as *reduce*) operation over a list of values. The code for this type and its Hanoi model is shown in Figure 4.2 — an artificial typestate restriction was added such that `foldLeft` must be called before `getResult` can be called. `foldLeft` is $O(n)$ in the size of the list when the binary operation applied is $O(1)$. The aim of the benchmark is to measure the time taken for each call to `foldLeft` with and without dynamic checking, to measure the relative overhead.

The test machine used was a 2012 Retina Macbook Pro with a 2.3 GHz Intel Core i7 processor and 8GB of DDR3-1600 RAM, running Mac OS X 10.8.3 with the Oracle Java Virtual Machine, version 1.7.0_06.

```
public String buildMonitoringAspectFor(Class<?> c) {
  String pack = c.getPackage().getName();
  String aspectName = c.getName() + "UsageMonitor";
  Import[] imports = buildImports();
  AspectMember[] aspectFields = buildFields(c);
  AspectMember[] constructorAdvice = buildConstructorAdvice(c);
  AspectMember[] methodAdvice = buildMethodAdvice(c);

  infer hanoiAspect = `[
    package #pack;

    #imports

    public aspect #aspectName issingleton() {
      #aspectFields
      #constructorAdvice
      #methodAdvice
    }
  ];

  return hanoiAspect.unparse();
}

private AspectMember[] buildMethodAdvice(Class<?> c) {
  ArrayList<AspectMember> allAdvice = new ArrayList<AspectMember>();
  for(Method m : c.getMethods()) {
    // ignore all standard Object methods and non-public methods
    if (m.getDeclaringClass().equals(Object.class)) continue;
    if (!Modifier.isPublic(m.getModifiers())) continue;
    allAdvice.add(buildMethodAdvice(c, m));
  }

  return allAdvice.toArray(new AspectMember[allAdvice.size()]);
}

private AspectMember buildMethodAdvice(Class<?> c, Method m) {
  infer returnType = `[ m.getReturnType().getCanonicalName() ];
  return `[
    #returnType around():
        target(#[c.getName()]) &&
        call(public #returnType #[m.getName()]) {

      IState currentState = getState(thisJoinPoint.getTarget());
      /* ... */
    }
  ];
}
```

Listing 4.16: Partial implementation of monitoring aspect generation using Meta-AspectJ

```
$ java -jar perftest.jar 200
WARMUP
done (7025194 iterations)
MAIN RUN
done (42519079 iterations)
array size: 200
dynamic checking: false
end-to-end time: 59999.877 ms
num iterations: 42519079
approximate mean per iteration: 1411ns
```

Listing 4.17: Output from the `foldLeft` benchmark

The `System.nanoTime()` method returns the elapsed time of the program using the highest precision timer available in the system. With the test machine, the timer is accurate to the nearest microsecond. The speed of the machine and the precision of the clock are such that directly measuring the time taken for each iteration is not possible — for lists of length $n < 200$, it was found that the system would often measure no elapsed time for the operation, indicating the clock is too coarse for measuring the time elapsed for operations of this size. An alternative approach is to invoke `foldLeft` millions of times and measure the total time taken. Dividing this time by the number of iterations gives an approximation of the time per call to `foldLeft`.

The benchmark program takes the array size as a parameter, allowing for the overhead to be determined relative to some known amount of operations. The benchmark first "warms up" the Java Virtual Machine by repeatedly invoking `foldLeft` for 10 seconds — this is necessary as the JVM performs a number of tasks from class loading to just-in-time compilation and optimisation based on observed patterns of execution. After warming up, `foldLeft` is repeatedly invoked for 60 seconds and the number of invocations counted, then divided by the total invocation count to produce an approximation of the call time. This approximation includes some additional overheads such as the evaluation of a loop condition to determine whether the 60 second run time is complete, and the incrementing of the counter, though such operations are inexpensive and necessary regardless of whether dynamic checking is enabled or not. As such, this should not impact the results derived in any meaningful way.

The benchmark results are printed to the terminal in the form shown in Listing 4.17. The approximate mean value was collected for lists of exponentially increasing size from 3 to 800 and are plotted in Figure 4.3, with linear lines of best fit. The observed overhead of Hanoi checking using a dynamic proxy is approximately 58 nanoseconds, which is roughly equivalent to the amount of work undertaken by the body of `foldLeft` with a multiplication `BinOp` and a list of size 3.

This can be quantified more precisely, in terms of the number of method calls and basic

```java
public class ArrayListFolder<T,U> {

  private T lastResult;

  public void foldLeft(T initVal, ArrayList<U> arr, BinOp<T,U> op) {
    lastResult = initVal;

    for(U u : arr) {
      lastResult = op.apply(lastResult, u);
    }
  }

  public T getResult() {
    return lastResult;
  }
}

public interface BinOp<T, U> {
  T apply(T t, U u);
}

public class Multiply implements BinOp<Integer, Integer> {
  @Override public Integer apply(Integer t, Integer u) {
    return t * u;
  }
}
```

```
ROOT {
  GET {
    getResult() -> ROOT
  }

  foldLeft(T,C,BinOp) -> GET
}
```

Figure 4.2: An implementation of fold in Java, with its Hanoi model

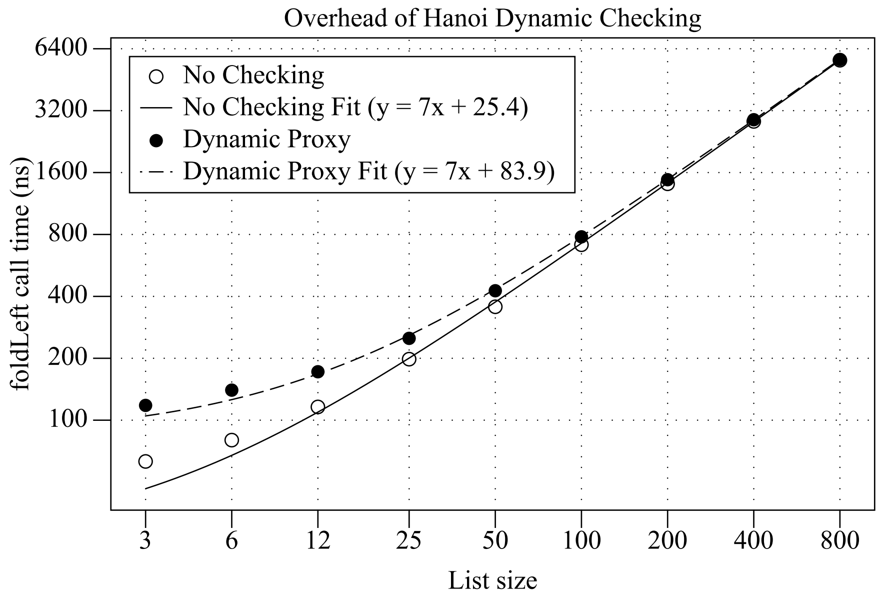


Figure 4.3: Observed overhead for the dynamic proxy based checker

operations undertaken in the implementation of `foldLeft`. The for loop syntactic sugar used on Line 8 of the implementation of `ArrayListFolder` is equivalent to a while loop using an `Iterator`. As such, the cost of `foldLeft` with a list of size n is equivalent to:

- one call to `iterator()` on `ArrayList`
- $n + 1$ calls to `hasNext()`
- n calls to `next()`
- n calls to `apply` on the provided `BinOp`
- $n + 1$ assignments to `lastResult`
- n assignments to `u`, the temporary variable used to hold the values of the list

For a list of size 3, this amounts to 10 method calls, 4 assignments and 3 multiplications.

An unavoidable component of the overhead for a dynamic proxy based checker is the cost of the dynamic proxy itself. The overhead of a dynamic proxy which does nothing more than invoke the wrapped object (shown in Listing 4.18) was measured to be approximately 9ns per call, or approximately 15% of the total cost of a proxy which also enforces a Hanoi model. This overhead can be potentially eliminated by the AspectJ based approach, but both implementations use the same logic for tracking the state of an object and therefore at a minimum Hanoi would introduce approximately 50 nanoseconds of overhead for every call that is monitored. This overhead increases where the return value must also be processed — in the example above, the transition in the state model is not sensitive to the return value.

```java
public class DoNothingInvocationHandler implements InvocationHandler {

  private Object wrappedObject;

  public DoNothingInvocationHandler(Object o) {
    this.wrappedObject = o;
  }

  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    return method.invoke(wrappedObject, args);
  }
}
```

Listing 4.18: The simplest dynamic proxy possible

Checking mode	Time	Factor of uninstrumented
Uninstrumented	0.7s	1x
Optimal wrapper	1.1s	1.6x
Proxy from Listing 4.18	33s	47x
Hanoi Dynamic Proxy	61s	87x

Figure 4.4: Benchmark times for matrix multiplication

It is clear from the results in Figure 4.3 that if the body of a method is complex, the overhead of dynamic checking is insignificant. If the body is trivial, however, the overhead is substantial. This is the case for the implementation of `Iterator` for most data structures — implementing `hasNext` and `next` involves little more than the maintenance of a pointer or an index, and so is very inexpensive compared to the cost of the dynamic checking. This is clearly demonstrated in another benchmark devised to generate a significant volume of calls to `Iterator`. The scenario of this second benchmark was involved multiplying large matrices, where matrices are represented as lists of lists and access to elements is performed using `Iterator` exclusively.

Given two matrices A and B such that $C = A \times B$, the code used to derive the value of C_{ij} (where i is the row and j the column) is shown in Listing 4.19. This code was executed with matrices A and B of size 30×30, requiring that this method be invoked once for each of the 900 cells of C. By instrumenting the code to count the number of invocations of `Iterator` methods, it was determined that 487350 calls are made to `hasNext` and 486450 calls are made to `next` for each matrix multiplication operation. A benchmark was constructed using 1000 matrix multiplication operations, recording the total time to carry out all multiplications (after a warmup period of 100 such multiplications). The times recorded are presented in Figure 4.4.

The overhead recorded in this benchmark for using dynamic proxies is substantial — the dynamic proxy based checker is almost two orders of magnitude slower than the uninstrumented program. At least half of this overhead is due to the cost of using a dynamic proxy, as seen in the time taken using the "do nothing" proxy from Listing 4.18.

Some overhead is obviously unavoidable if we wish to enforce the typestate constraints in a Hanoi model. In order to measure what one could argue would be the minimum overhead possible using a wrapper-based approach, a simple hand-crafted checker was produced. A fragment of this is shown in Listing 4.20. The checking performed is similar in structure to the code for the advice in Listing 4.13. This implementation avoids some unnecessary checks: `hasNext` is available in all states, therefore the current state need not be checked before performing the real call. The return value for `next` need not be processed as the transition for `next` is unconditional. The manipulation of primitive booleans to represent the state of the object is likely to be significantly cheaper than using an object to represent the state, as the Hanoi implementation presently does.

The overhead observed for such a simple checker in the matrix multiplication benchmark is approximately 60%. A more sophisticated implementation of the aspect generator may be able to produce code as efficient as this for checking Hanoi models. This still represents a significant overhead compared to the uninstrumented program, though the scenario is rather extreme — the execution time of the program is dominated by the cost of calls to `Iterator` instances, which is unlikely to be the case in a more realistic program.

The lesson that can be derived from these observations is that the current implementation of Hanoi is suitable for use in checking typestate constrained interfaces in the following circumstances:

- When safety is more important than performance — this is likely to be the case where the interface in question controls communication with other modules in a system or with external systems.
- When the calls are infrequent, and therefore unlikely to seriously impact the performance of the program.
- When the calls are complex, and therefore the overhead introduced by dynamic checking is insignificant next to the complexity of the call itself.
- During testing, where performance is less of a concern than finding potential bugs.

With further optimisation, it may be possible to bring the overhead of the AspectJ based approach down to within an order of magnitude of the uninstrumented run time of the matrix multiplication benchmark. At this level of overhead, it may be acceptable to use this monitoring approach in production code. The overheads of using dynamic, interpreted languages such as Python compared to Java are routinely accepted for the convenience derived — according to the Computer Programming Language Benchmark Game [52], Python 3 code is

```java
int getValue(int row, int col) {
  int value = 0;
  Iterator<Integer> aRowIter = getNth(a.rows.iterator(), row);
  Iterator<List<Integer>> bRowsIter = b.rows.iterator();

  while(aRowIter.hasNext()) {
    int aValue = aRowIter.next();

    if(!bRowsIter.hasNext()) throw new IllegalArgumentException();
    List<Integer> bRow = bRowsIter.next();

    int bValue = getNth(bRow.iterator(), col);

    value += aValue * bValue;
  }

  return value;
}

int getNth(Iterator<Integer> iter, int n) {
  int pos = 0;
  int last = null;
  while(pos <= n) {
    if(!iter.hasNext()) throw new IllegalArgumentException();
    last = iter.next();
    pos++;
  }

  return last;
}
```

Listing 4.19: Matrix multiplication implemented using Iterator to access elements

typically a factor of 40 slower than Java code (while Java code is typically a factor of 2 slower than C code). As such, an order of magnitude slowdown for the convenience of not having to write defensive code to enforce typestate constraints may be acceptable in some circumstances.

4.4 Conclusion

A number of approaches have been evaluated and tested for the dynamic checking of Hanoi models, with an emphasis on approaches which are likely to be practical for deployment outside of a research project environment. An AspectJ based approach was found to provide the most practical solution, against the criteria laid out in Section 4.1: no language changes are required, the checker can be enabled with minimal configuration, explicit cooperation of client code is not required for checking to be enforced, and the prototype implementation could be further augmented to allow for selective monitoring within specific regions of a code base. The current implementation of the AspectJ approach is suitable for use during testing

```
public class OptimalCheckIterator<E> implements Iterator<E> {

  private Iterator<E> realIter;

  private boolean nextAvailable;
  private boolean canRemove;

  public OptimalCheckIterator(Iterator<E> iter) {
    this.realIter = iter;
    resetState();
  }

  @Override
  public boolean hasNext() {
    try {
      boolean result = realIter.hasNext();
      if(result) nextAvailable = true;
      return result;
    } catch(RuntimeException e) {
      resetState();
      throw e;
    } catch(Error e) {
      resetState();
      throw e;
    }
  }

  @Override
  public E next() {
    if(!nextAvailable) throw new IllegalStateException();

    try {
      E result = realIter.next();
      nextAvailable = false;
      canRemove = true;

      return result;
    } catch(RuntimeException e) {
      resetState();
      throw e;
    } catch(Error e) {
      resetState();
      throw e;
    }
  }

    private void resetState() {
    nextAvailable = false;
    canRemove = false;
  }

  /* ... */
}
```

Listing 4.20: A fragment of an “optimal” hand-written wrapper for Iterator to check its typestate constraints

or to monitor interfaces for which performance is not important. With further optimisation, it may be possible to bring the overhead of this approach down to a level which would be acceptable for use in production code.

The matrix multiplication micro-benchmark provides an estimate of the worst-case overhead that Hanoi can introduce: the performance of the program is strongly determined by the cost of calling `Iterator` methods, and the dynamic checking in this example results in such calls costing two orders of magnitude more than when they are uninstrumented. This overhead is similar to what one would experience when using a profiler which collects detailed statistics on every method call made by a program — while useful, this overhead is not acceptable for a program running in a production environment.

However, if instead one were to limit monitoring to methods which are not performance critical in a program, the overhead would be negligible. Monitoring the state of IO channels or interactions with graphical user interface elements and remote services would typically fall into this category — calls are infrequent or expensive, such that the total cost of checking and updating typestate information is a small fraction of the other work being undertaken by the program.

Characterising the overhead in the "general" case is difficult due to the variety of different behaviours exhibited in programs, and the options available in selecting which classes are monitored. The selection of what constitutes an important typestate constraint is very program-specific, and is a trade-off between the value the monitoring provides versus the acceptable overhead for the program in question.

Chapter 5

Can programmers reason about typestate?

Programming language designers are often accused of inventing new programming constructs without having any notion of how well suited such techniques are to industrial software development. While they may be able to show small, pedagogical examples of how a new technique can counter a class of bugs, or allow a more compact expression of a particular concept, this is rarely supported by a study which demonstrates that the technique fits the following important criteria:

- Worth learning — the overall productivity gains made possible by the new feature outweigh the effort of learning the new feature, and any overheads associated with its use.
- Clearly expressed — the particular syntactic elements used in the expression of the new feature are the best that could be found, or equivalent to other evaluated possibilities.
- Justified — the increase in complexity of the language due to the addition of the feature is justifiable, in that the feature is of greater general utility than some other feature that could be added. All languages have a notional "complexity budget" that cannot be exceeded without causing confusion and misuse. The feature set of a language must be chosen carefully to provide the greatest overall utility within that budget for the domain in which the language is expected to be used.

The introduction of typestate to a language should be evaluated against these criteria as well. Can a competent programmer learn what typestate is, identify where it should be used, and reason about code which interacts with objects that have typestate constraints? There are many aspects of this problem that are difficult to quantify:

- What qualifies a programmer as "competent"? In other words, what is the expected or required knowledge that a programmer must have for typestate to be a worthwhile addition to a language?
- What constitutes effective reasoning? Building an accurate mental model of what code does is notoriously difficult and a source of many bugs — despite the best efforts of a type system, mistakes are likely. In the case of typestate, assessing whether a programmer can reason about state transitions most of the time, such that they are not constantly surprised by a static analysis tool disagreeing with their mental model, should be sufficient.
- How can we quantify whether a feature is of more value than another? Such a question is subjective. The features of a programming language interact in non-trivial ways such that swapping one feature for another is not always possible; assessing the impact of such a change is likely to be difficult.

Given the complexity in answering the above questions, it is understandable that applied programming language research is rarely accompanied by user studies that justify the utility of a particular type system feature or static analysis tool.

Regardless, assessing the utility of typestate through experimentation is likely to be essential in convincing those outwith the research community that it has value, and should be included in contemporary programming languages.

This chapter assesses the practicality of the Hanoi modelling language, and directly compares the DSL and annotation based model types. The annotation model type is a good approximation of the syntax used to express the typestate specific features of Plural and Vault, and therefore should contribute some evidence as to the practicality of the relevant subset of such tools as well. By comparing the DSL and annotation model types, we may be able to determine whether the difference in presentation of the model has any noticeable impact on a programmer's ability to understand a typestate model.

5.1 Experiments considered

In order to evaluate the practicality of the Hanoi language for typestate modelling, an experimental design was desired with the aim of evaluating the following research questions:

- Can programmers reason effectively about the semantics of Hanoi?
- Is the effectiveness with which a programmer can reason about a Hanoi model influenced by the presentation of the model?

- Will programmers prefer either the DSL or annotation model types?

These questions should be evaluated in the context of one or more of the following tasks:

- Deciding whether a Hanoi model is semantically valid.
- Writing code that does not violate constraints in a Hanoi model.
- Producing a semantically valid Hanoi model against an informal specification.
- Identifying whether code violates constraints in a Hanoi model.

Different experimental designs were considered, oriented around questions related to these tasks, before settling on a final experimental design.

Can programmers find semantic errors in Hanoi specifications? An experiment could be devised where participants are presented with Hanoi models of varying complexity, and asked to identify whether any of the transitions specified are invalid. While such a study should provide a clear answer as to whether participants understand the rules, and which rules or rule interactions cause the most difficulty, presenting the models in isolation from model usage is unlikely to provide a useful assessment of the practicality of the model. Not only must the model be semantically correct, it should match the intended behavioural restrictions of a class. This semantic relationship dictates the structure of the model and often allows for intuitive deduction of when a particular constraint is nonsensical, rather than from first principles.

It was decided that conducting a study exclusively of this form would not provide sufficient insight to evaluate the hypotheses, though it could be useful to pose questions of this form as part of a larger study.

Can programmers write code that conforms to a Hanoi model? An experiment could be devised where a programmer is asked to write code to solve problems, using an API with typestate restrictions modelled using Hanoi. Such an experiment would provide a realistic setting, but it is not clear how the user's performance should be evaluated. The code could be scored automatically based on whether it passes a set of unit tests, and whether the code violates any typestate constraints. With such an experiment, the user's performance is primarily determined by their ability to code a working solution, with the typestate violations as a secondary concern. Therefore, determining the impact of typestate on the programmer's ability to solve a problem is likely to be difficult without a large study.

A qualitative approach may prove better for a small study, however the results are likely to be similarly indirect and therefore unlikely to adequately evaluate the hypotheses. The best option for a study of this form may be to assess the performance of a small group of programmers over a long period of time, writing significantly more code against an API composed of multiple typestate constrained interfaces, and then interview the participants about their experiences of using the API and tools that provided static and dynamic analysis of the API. Unfortunately, such a study is far too ambitious given the limited time available, and would constitute a PhD in itself.

Can programmers write correct Hanoi models? An experiment to answer this question could provide participants with the implementation of a class, or an interface with an informal description of the desired typestate constraints. From this, a user would be asked to construct a Hanoi model. Such a model could be automatically checked and scored, by testing whether it would allow or disallow sequences of method calls as desired. Ultimately, such a model could be checked as to whether it is directly equivalent to a correct model, by ensuring they are both simulations of each other.

A pilot experiment of this form was conducted with three participants. The time taken for participants to produce two models was measured, along with a score from 0-10 derived by automated analysis of the models as described — 5 points for rejecting illegal sequences of method calls (the exact sequences were not revealed to the participants), and 5 points for providing a model which was directly equivalent to a correct model. Participants were assigned to groups, with group A to provide the first model in annotation form and the second in DSL form, while group B were to do the inverse (this is commonly known as a *within group counter-balanced study*).

While the pilot provided useful insight into how participants attempted to construct typestate models and whether they were correct, it was determined that running a larger scale version of such a study would take more time and resource than could be reasonably allocated — times observed for the production of one model, for a small class whose typestate constraints could be expressed with a 10 line DSL model, were up to 45 minutes. Two models seemed inadequate to provide enough quantitative or qualitative information to evaluate the hypotheses, therefore additional models would be required and a more detailed scoring system would have to be devised.

It was desired to conduct a study where the maximum amount of information could be derived in a 90 minute period (which would include a short tutorial on Hanoi), therefore a study of this form was unlikely to be a good fit.

Can programmers identify typestate violations in existing code? An experiment to answer this question could ask the user to play the role of a static analysis tool, inspecting a fragment of code for typestate violations, given a Hanoi model. By choosing realistic models and realistic code using those models, such an experiment may give useful insight into the performance of programmers in a practical setting. Reading code is at least as important as writing it, therefore testing that the programmers ability to reason about existing code that interacts with Hanoi modelled classes is an essential part of assessing the practicality of Hanoi.

A pilot experiment of this form was conducted with the same three participants from the model writing experiment. Participants were able to answer questions based on existing code quickly, typically in under 5 minutes. As such, it seemed likely an experiment of this form would provide useful data to evaluate the hypotheses, and could be conducted with the time and resource available.

5.2 Experimental design

Based on experiences from the two pilots conducted, it was decided that the experiment should last at most 90 minutes per participant, and that each participant would be individually assessed. Participants ($n = 10$) were selected on the basis of their experience with the Java programming language and their ability to pass a simple test (see Figure 5.1) of their knowledge of the inheritance and overriding rules of Java. Participants were paid the standard hourly rate of the institution at which the experiment was conducted.

The primary objective of the experiment was to compare the DSL and annotation forms of Hanoi, in order to determine whether the presentation of the model had a noticeable impact on performance, or whether users preferred either style of presentation.

The experiment began with a 20 minute tutorial on the conceptual background of typestate, and then of the syntax and semantics of Hanoi using a motivating example presented initially as a state chart. Both the DSL and annotation forms of this model were demonstrated simultaneously, and their semantic equivalence emphasised.

Four typestate models were devised that represented realistic typestate constrained interfaces, and four multiple choice questions were chosen for each model to test the ability of a participant to reason about them effectively. Participants were asked to read a model, and only proceed to the questions on that model once they felt they had correctly understood the restrictions the model described, in an attempt to provide data to distinguish reading times between the two model types. The time taken for a participant to answer a question related to a model was similarly recorded. Participants were instructed not to guess, with the option

Java Question 1 What does the following code output?
Options: byte, short, int, **long**

```
class A {
  public void x(byte b) { System.out.println("byte"); }
  public void x(long l) { System.out.println("long"); }
}

class B extends A {
  public void x(short s) { System.out.println("short"); }
  public void x(int i) { System.out.println("int"); }
}

class Main {
  public static void main(String[] args) {
    A a = new B();
    a.x((short)5);
  }
}
```

Java Question 2 What does the following code output?
Correct response: `6 4 2` (with newline separators)

```
import java.util.*;

import static java.util.Arrays.*;
import static java.lang.System.*;

class Main {
  public static void main(String[] args) {
    List<Integer> list = asList(1, 2, 3, 4, 5, 6);
    Stack<Integer> filtered = new Stack<Integer>();
    Iterator<Integer> it = list.iterator();
    while(it.hasNext()) {
      int val = it.next();
      if(val % 2 == 0) {
        filtered.push(val);
      }
    }

    while(!filtered.isEmpty()) out.println(filtered.pop());
  }
}
```

Figure 5.1: Java puzzles used to test comprehension of Java's semantics

to skip a question (providing no specific answer) if they were unable to determine the correct answer.

It was felt that testing each participant on only one of the two forms of model would not provide reliable information for comparing the two approaches. With four questions, it would be possible to give two models of each type to each participant, in an effort to compensate for the likely differences in innate ability between the participants.

It was also anticipated that a learning effect would be present in the experiment (i.e. the speed and accuracy of answering questions would improve with experience, even over the short period of time involved in this experiment), therefore it would be undesirable to show the annotation models first and then the DSL models, or vice versa. Instead, models were shown alternately, with participants randomly divided between two groups, A and B. Group A were shown annotation models for questions 1 and 3, and DSL models for questions 2 and 4. Group B were shown the inverse.

During the course of the experiment, all participants were asked to think aloud while answering the questions, and were recorded with consent, with the aim of collecting qualitative information on their thought processes, and aspects of the question and models which caused them difficulty.

Finally, participants would receive one point for each correctly answered question, for a total of 16 points over the entire experiment.

After answering all questions, anticipated to take roughly 45 minutes, the participants answered a short survey rating their preference for either the DSL or annotation models, independent of their performance. Responses were recorded using a standard five point Likert scale, with care taken to ensure that acquiescence bias (the tendency to agree with a statement as presented) would not favour either of the two models — questions alternated between statements phrased as “I find it easier to … with a DSL model” and those phrased as “I find it easier to … with an Annotation model”.

5.2.1 Null hypotheses

Formally, the following null hypotheses were evaluated during this experiment:

1. The time taken for a participant to read and answer a question related to a Hanoi model is independent of the model type.
2. The time taken for a participant to read and understand a model is independent of the Hanoi model type.

3. Participant performance in answering questions related to the semantics of a Hanoi model is independent of the model type.
4. Participant performance is independent of their self-rated level of experience with object oriented programming, functional programming or formal methods.
5. Participant performance is independent of their number of years experience as a programmer.
6. Participants express no preference for either Hanoi model type.

5.3 Experiment questions

The description of each Hanoi model and questions related to a model posed to the participants, along with a discussion of their intended goal in evaluating the hypotheses, is presented below. Statechart representations are provided for illustrative purposes, however these were not shown to the experiment participants. Otherwise, the descriptions of the models and questions are exactly as presented to the participants during the experiment.

5.3.1 Model 1 — DistributedWorkQueue

A `DistributedWorkQueue` represents the local handle to a queue of work to be done in a distributed system. Workers signal they are ready to work by calling `join()` and signal they are no longer available to do work by calling `leave()`. While working, the methods `takeInput()` and `produceOutput()` are used to consume and produce data.

Java interface:

```
public interface DistributedWorkQueue<T,U> {
  void join();
  void leave();
  T takeInput();
  void produceOutput(U u);
}
```

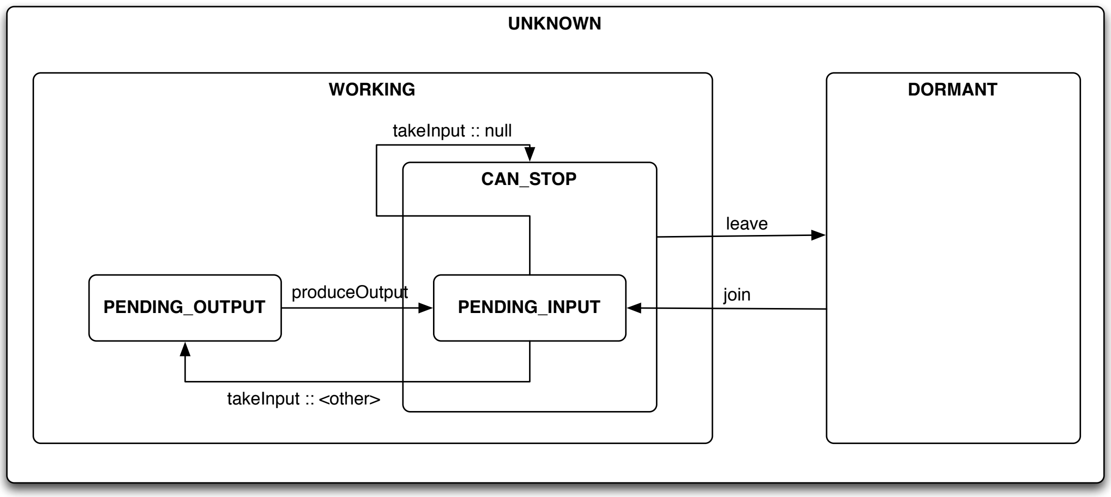
Hanoi DSL model:

```
UNKNOWN {
  WORKING {
    PENDING_OUTPUT {
      produceOutput(U) -> PENDING_INPUT
    }

    CAN_STOP {
      PENDING_INPUT {
        takeInput() :: null -> CAN_STOP
        takeInput() :: <other> -> PENDING_OUTPUT
      }

      leave() -> DORMANT
    }
  }

  DORMANT {
    join() -> PENDING_INPUT
  }
}
```

Hanoi annotation model:

```
@States({
  @State(name="UNKNOWN"),
  @State(name="WORKING", parent="UNKNOWN"),
  @State(name="PENDING_OUTPUT", parent="WORKING"),
  @State(name="CAN_STOP", parent="WORKING"),
  @State(name="PENDING_INPUT", parent="CAN_STOP"),
  @State(name="DORMANT", parent="UNKNOWN")
})
public interface DistributedWorkQueue<T,U> {

  @Transition(from="DORMANT", to="PENDING_INPUT")
  void join();

  @Transition(from="CAN_STOP", to="DORMANT")
  void leave();

  @Transitions({
    @Transition(from="PENDING_INPUT",
                to="CAN_STOP",
                whenResult="null"),
    @Transition(from="PENDING_INPUT",
                to="PENDING_OUTPUT",
                whenResult="<other>")
  })
  T takeInput();

  @Transition(from="PENDING_OUTPUT",
              to="PENDING_INPUT")
  void produceOutput(U u);
}
```

State chart:



Discussion This model exhibits the common *alternating behaviour* pattern between `takeInput` and `produceOutput`, but with a conditional transition based upon the return value of `takeInput` (intended to indicate that no more work is available in the queue). Client code must take care to correctly check the return value. Additionally, the client may only call `leave` when it has not taken a pending work item or there is no more work. The `WORKING` and `DORMANT` states are purely descriptive, as they define no transitions of their own.

Due to the depth of the hierarchy, it was speculated that this model may be easier to work with in the DSL form than in the annotation form, as the structure of the hierarchy and the inherited transitions (in particular, the ability to call `leave` from `PENDING_INPUT`) may be more readily apparent.

Question 1A

Consider the method `doPrimeWork`, which computes whether numbers provided through the `DistributedWorkQueue` handle are prime numbers or not.

```
public void doPrimeWork(DistributedWorkQueue<BigInteger,Boolean>
    ↳ workHandle, int numIters) {
  workHandle.join();
  for(int i=0; i < numIters; i++) {
    BigInteger input = workHandle.takeInput();
    workHandle.produceOutput(isPrime(input));
  }
  workHandle.leave();
```

```
 8 }
 9
10 private boolean isPrime(BigInteger i) { /*...*/ }
```

A `DistributedWorkQueue` instance in state `DORMANT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If not, choose “no violation”.

Options: (A) line 2, (B) line 4, **(C) line 5**, (D) line 7, (E) no violation

Discussion The call to `join` on Line 2 results in a state transition to `PENDING_INPUT`.

If `numIters` ≤ 0 then `leave` on line 7 will be called, resulting in a transition back to `DORMANT` (as this transition is inherited from the definition of `CAN_STOP`.

If `numIters` > 0 then the body of the loop will be executed. The transition triggered by the call to `takeInput` on line 4 is conditional upon the return value, which is stored in `input`. This value is not inspected and therefore `workHandle` is either in state `PENDING_OUTPUT` or `CAN_STOP`. The call to `produceOutput` on line 5 is not legal in state `CAN_STOP`, therefore a typestate violation can result from this call.

Correctly answering this question requires that the participant can reason about conditional transitions, and what can be done with an object in an indeterminate state.

Question 1B

Consider the `doSearchWork` method, which attempts to count the number of matches of a regular expression within a large corpus of data shared by each node. This operation can fail, resulting in an `IOException` being thrown.

```
 1 public void doSearchWork(DistributedWorkQueue<String,Integer> workHandle)
       ↳  throws IOException {
 2   workHandle.join();
 3   String regex = workHandle.takeInput();
 4   try {
 5     if(regex != null) {
 6       findOccurrences(regex);
 7     }
 8   } catch(IOException e) {
 9     workHandle.produceOutput(null);
10   }
11   workHandle.leave();
12 }
13
14 public int findOccurrences(String regex) throws IOException
```

```
15    { /* ... */ }
```

If `doSearchWork` is run and an `IOException` is thrown by `findOccurrences` on line 6, what state is the `workHandle` instance in on line 11 prior to the call to `leave`?

Options: (A) UNKNOWN, (B) WORKING, (C) PENDING_OUTPUT, (D) CAN_STOP, **(E) PENDING_INPUT**, (F) DORMANT

Discussion The calls to `join` and `takeInput` (with a non-null value, as checked on line 5) result in a transition to state `PENDING_OUTPUT`. If an `IOException` is thrown as indicated, `produceOutput` will be called, resulting in a transition to state `PENDING_INPUT`, prior to the call to `leave`.

Correctly answering this question requires that the participant be able to reason about the impact of exception control flow. In this particular scenario, it was not expected that this combination of exceptions with typestate would prove particularly taxing, but nonetheless such an assumption was worth checking.

Question 1C

Consider the `fetchResources` method, which retrieves the String content of a URL through an HTTP request.

```
1  public void fetchResources(DistributedWorkList<URL, String> workHandle) {
2    workHandle.join();
3    try {
4      URL resLoc = workHandle.takeInput();
5      if(resLoc != null) {
6        workHandle.produceOutput(fetchPage(url));
7      }
8    } finally {
9      workHandle.leave();
10   }
11 }
12
13 public String fetchPage(URL url) throws IOException { /* ... */}
```

A `DistributedWorkQueue` instance in state `DORMANT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If not, choose "no violation".

Options: (A) line 2, (B) line 4, (C) line 6, **(D) line 9**, (E) no violation

Discussion The initial call to `join` results in a transition to `PENDING_INPUT`.

If the call to `takeInput` on line 4 returns a null value, this results in a transition to `CAN_STOP` and execution skips to line 9. The call to `leave` then results in a transition to `DORMANT`.

If the call to `takeInput` returns a non-null value, this results in a transition to `PENDING_OUTPUT` and execution of line 6 occurs.

If an exception is not thrown by `fetchPage`, then the call to `produceOutput` will result in a transition to `PENDING_INPUT`, followed by the call to `leave` resulting in a transition to state `DORMANT`.

If an exception is thrown by `fetchPage`, then execution skips to line 9 with `workHandle` still in state `PENDING_OUTPUT`. The call to `leave` is not legal in this state. Therefore, a typestate violation may occur on line 9.

Correctly answering this question requires that the participant be able to reason about the impact of exception control flow, in a more complex scenario than in the previous question. There are multiple paths to the `leave` call and all must be considered for legality; in particular, the user must be aware of the fact that `fetchPage` may throw an exception, and that this will result in `produceOutput` not being called.

Question 1D

Consider the method `identityWorkStep`:

```
public void identityWorkStep(DistributedWorkList<T,T> workHandle) {
  workHandle.produceOutput(workHandle.takeInput());
}
```

A `DistributedWorkQueue` instance in state `PENDING_INPUT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate which method will cause the violation. If no violation will occur, choose “no violation”.

Options: **(A) produceOutput**, (B) takeInput, (C) no violation

Discussion As `workHandle` is already in state `PENDING_INPUT`, the call to `takeInput` will result in a transition to `CAN_STOP` if a null value is returned, or to `PENDING_OUTPUT` otherwise. In the first case, the call to `produceOutput` is not permitted.

Similar to Question 1, this requires that the user be able to reason about conditional transitions, and pay attention to the case where null may be returned. Additionally, it breaks from the pattern of the previous questions of starting in state DORMANT, to test that they can reason about interacting with the object from a different start state than they might expect.

5.3.2 Model 2 — Calculator

A Calculator instance represents a simple event driven calculator, which responds to input from a user interface. Rules govern the order in which symbols may be entered in order to produce valid expressions, so the UI code must ensure that it only enables the correct buttons at each stage. The Calculator interface is defined as follows:

Java interface:

```java
public interface Calculator {
  public void begin();
  public void onNum(int num);
  public void onPlus();
  public void onMinus();
  public int onEquals();
}
```

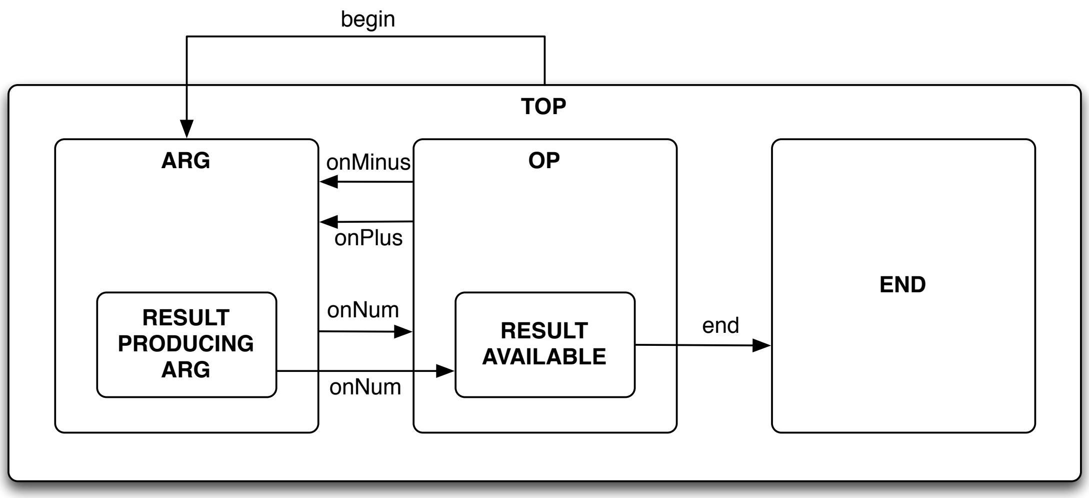
Hanoi DSL model:

```
TOP {
  ARG {
    RESULT_PRODUCING_ARG { onNum(int) -> RESULT_AVAILABLE }
    onNum(int) -> OP
  }

  OP {
    RESULT_AVAILABLE { onEquals() -> END }
    onPlus() -> RESULT_PRODUCING_ARG
    onMinus() -> RESULT_PRODUCING_ARG
  }

  END {}

  begin() -> ARG
}
```

Hanoi annotation model:

```java
@States({
  @State(name="TOP"),
  @State(name="ARG", parent="TOP"),
  @State(name="OP", parent="TOP"),
  @State(name="END", parent="TOP"),
  @State(name="RESULT_PRODUCING_ARG", parent="ARG"),
  @State(name="RESULT_AVAILABLE", parent="OP")
})
public interface ICalculator {

  @Transition(from="TOP", to="ARG")
  public void begin();

  @Transitions({
    @Transition(from="ARG", to="OP"),
    @Transition(from="RESULT_PRODUCING_ARG", to="RESULT_AVAILABLE")
  })
  public void onNum(int num);

  @Transition(from="OP", to="RESULT_PRODUCING_ARG"),
  public void onPlus();

  @Transition(from="OP", to="RESULT_PRODUCING_ARG")
  public void onMinus();

  @Transition(from="RESULT_AVAILABLE", to="END")
  public int onEquals();
}
```

State chart:



Discussion This model is intended to provide an example of a stateful event-driven interface, as would likely be found in user interface toolkits or SEDA[1] networking toolkits like Netty [86].

[1]Staged Event Driven Architecture [150]

The interface exhibits alternating behaviour, expecting calls to `onNum` and `onPlus` / `onMinus`. Upon entering a full binary operator expression, the `onEquals` method is enabled. At any point, `begin` can be called to reset the state of the object, as this is in the `TOP` state — this is intended to be analogous to the all clear button (AC) on a calculator. The `END` state serves no purpose other than documentation, as it does not modify or add any behaviour to the `TOP` state.

While conditional transitions are not present in this model (in contrast to Model 1), inheritance and overriding are present and essential to understanding this model. It was expected that the syntactic relationship between parent and child states in the DSL model would make overriding more obvious, particularly in the case of `onNum` in `RESULT_PRODUCING_ARG`.

Question 2A

Consider the following test code:

```
1 public int twoPlusTwo(Calculator calc) {
2   calc.begin();
3   calc.onNum(2);
4   calc.onPlus();
5   calc.onNum(2);
6   return calc.onEquals();
7 }
```

A `Calculator` instance in state `END` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If no violation will occur, choose “no violation”.

Options: (A) line 2, (B) line 3, (C) line 4, (D) line 5, (E) line 6, **(F) no violation**

Discussion The sequence of transitions is as follows:

Line	State prior to call	Method	
1	`END`	begin	✓
2	`ARG`	onNum	✓
3	`OP`	onPlus	✓
5	`RESULT_PRODUCING_ARG`	onNum	✓
6	`RESULT_AVAILABLE`	onEquals	✓
—	`END`		

Deriving the correct answer for this question requires that the participant be able to correctly identify that `begin` is available in the `END` state, and that the second call to onNum results in a transition to a different state from the first due to the override in RESULT_PRODUCING_ARG.

It was not expected that this question would be more difficult to answer given either model as each transition (with the exception of the begin call) is explicitly declared for each relevant source state.

Question 2B

Which of the following methods is not available in the OP state?

Options: **(A) onEquals**, (B) begin, (C) onPlus, (D) onMinus

Discussion The OP state explicitly permits onPlus and onMinus, and inherits a transition for begin from TOP. As a result, the only method that is not permitted is onEquals.

This simple question specifically tests the participant's ability to understand the inheritance of the begin method, in isolation from any other concern. It was expected that this should be marginally easier to identify using the DSL model.

Question 2C

Consider the following sequence of method calls, produced from a user interacting with the calculator:

```
calc.begin();
calc.onNum(2);
calc.onMinus();
calc.begin();
calc.onNum(2);
calc.onPlus();
calc.onNum(5);
calc.onMinus();
```

What state is the calc instance in at the end of this sequence?

Options: (A) TOP, (B) ARG, **(C) RESULT_PRODUCING_ARG**, (D) OP,
(E) RESULT_AVAILABLE, (F) END

Discussion The sequence of transitions is as follows:

Line	State prior to call	Method	
1	???	begin	⤦
2	`ARG`	onNum	⤦
3	`OP`	onMinus	⤦
4	`RESULT_PRODUCING_ARG`	begin	⤦
5	`ARG`	onNum	⤦
6	`OP`	onPlus	⤦
7	`RESULT_PRODUCING_ARG`	onNum	⤦
8	`RESULT_AVAILABLE`	onMinus	⤦
—	`RESULT_PRODUCING_ARG`		

This question is intended to test the user's ability to trace longer sequences of transitions than in prior questions. They are not provided with an initial state, but should correctly identify that this is irrelevant as the `begin` method may be called in any state due to inheritance.

Question 2D

Consider the `multiply` method, which implements multiplication using a `Calculator` instance.

```
 1  public int multiply(Calculator calc, int a, int b) {
 2    calc.begin();
 3    calc.onNum(0);
 4    for(int i=0; i < Math.abs(b); i++) {
 5      if(b < 0) {
 6        calc.onMinus();
 7      } else {
 8        calc.onPlus();
 9      }
10      calc.onNum(a);
11    }
12    return calc.onEquals();
13  }
```

A `Calculator` instance in state `TOP` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If no violation will occur, choose "no violation".

Options: (A) line 2, (B) line 3, (C) line 6, (D) line 8, (E) line 10, **(F) line 12**, (G) no violation.

Discussion If the body of the loop is executed (i.e. $abs(b) > 0$) then there is no violation. However, if $b = 0$, then the call to `onEquals` occurs when the object is in state `OP`:

Line	State prior to call	Method	
2	`TOP`	begin	⤦
3	`ARG`	onNum	⤦
12	`OP`	onEquals	×

`onEquals` is not permitted in this state, therefore a typestate violation may occur on line 12 if $b = 0$.

Deriving the correct answer for this question relies on the user being able to correctly identify all possible paths through the code, and either trace the state of the object in each case separately or reason about all of them simultaneously, by determining the set of states that the object may be in prior to each method call.

5.3.3 Model 3 — GearControl

Consider the following type:

```
public interface GearControl {
  Gear currentGear();
  void shiftUp();
  void shiftDown();
  void neutral();
  void declutch();
  void clutch();
  boolean isDeclutched();
}

enum Gear { NEUTRAL, ONE, TWO }
```

`GearControl` represents the sequential gear box and clutch of an automobile controlled by an engine management unit. The car has two gears and a neutral position represented by the `Gear` enum.

Hanoi DSL model:

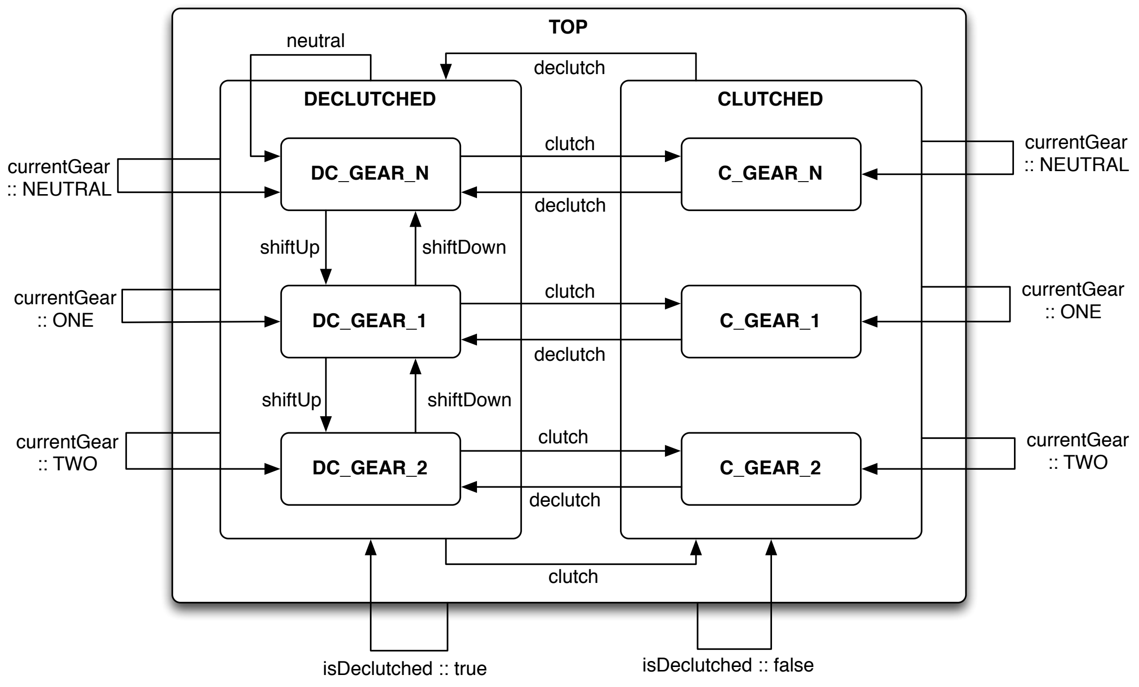
```
TOP {
  DECLUTCHED {
    DC_GEAR_N {
      shiftUp() -> DC_GEAR_1
      clutch() -> C_GEAR_N
    }
    DC_GEAR_1 {
      shiftUp() -> DC_GEAR_2
      shiftDown() -> DC_GEAR_N
      clutch() -> C_GEAR_1
    }
    DC_GEAR_2 {
      shiftDown() -> DC_GEAR_1
      clutch() -> C_GEAR_2
    }

    currentGear() :: NEUTRAL -> DC_GEAR_N
    currentGear() :: ONE -> DC_GEAR_1
    currentGear() :: TWO -> DC_GEAR_2

    clutch() -> CLUTCHED
    neutral() -> DC_GEAR_N
    isDeclutched() :: true -> <self>
  }

  CLUTCHED {
    C_GEAR_N { declutch() -> DC_GEAR_N }
    C_GEAR_1 { declutch() -> DC_GEAR_1 }
    C_GEAR_2 { declutch() -> DC_GEAR_2 }

    declutch() -> DECLUTCHED

    currentGear() :: NEUTRAL -> C_GEAR_N
    currentGear() :: ONE -> C_GEAR_1
    currentGear() :: TWO -> C_GEAR_2

    isDeclutched() :: false -> <self>
  }

  isDeclutched() :: true -> DECLUTCHED
  isDeclutched() :: false -> CLUTCHED
}
```

Hanoi annotation model:

```
@States({
  @State(name="TOP"),
  @State(name="DECLUTCHED", parent="TOP"),
  @State(name="DC_GEAR_N", parent="DECLUTCHED"),
  @State(name="DC_GEAR_1", parent="DECLUTCHED"),
  @State(name="DC_GEAR_2", parent="DECLUTCHED"),
  @State(name="CLUTCHED", parent="TOP"),
  @State(name="C_GEAR_N", parent="CLUTCHED"),
  @State(name="C_GEAR_1", parent="CLUTCHED"),
  @State(name="C_GEAR_2", parent="CLUTHCED")
})
public interface GearControl {
```

```
@Transitions({
  @Transition(from="DECLUTCHED", to="DC_GEAR_N", whenResult="NEUTRAL"),
  @Transition(from="DECLUTCHED", to="DC_GEAR_1", whenResult="ONE"),
  @Transition(from="DECLUTCHED", to="DC_GEAR_2", whenResult="TWO"),
  @Transition(from="CLUTCHED",   to="C_GEAR_N",  whenResult="NEUTRAL"),
  @Transition(from="CLUTCHED",   to="C_GEAR_1",  whenResult="ONE"),
  @Transition(from="CLUTCHED",   to="C_GEAR_2",  whenResult="TWO")
})
  Gear currentGear();

  @Transitions({
    @Transition(from="DC_GEAR_N", to="DC_GEAR_1"),
    @Transition(from="DC_GEAR_1", to="DC_GEAR_2")
  })
  void shiftUp();

  @Transitions({
    @Transition(from="DC_GEAR_1", to="DC_GEAR_N"),
    @Transition(from="DC_GEAR_2", to="DC_GEAR_1")
  })
  void shiftDown();

  @Transitions({
    @Transition(from="DECLUTCHED", to="DC_GEAR_N")
  })
  void neutral();

  @Transitions({
    @Transition(from="CLUTCHED", to="DECLUTCHED"),
    @Transition(from="C_GEAR_N", to="DC_GEAR_N"),
    @Transition(from="C_GEAR_1", to="DC_GEAR_1"),
    @Transition(from="C_GEAR_2", to="DC_GEAR_2")
  })
  void declutch();

  @Transitions({
    @Transition(from="DECLUTCHED", to="CLUTCHED"),
    @Transition(from="DC_GEAR_N", to="C_GEAR_N"),
    @Transition(from="DC_GEAR_1", to="C_GEAR_1"),
    @Transition(from="DC_GEAR_2", to="C_GEAR_2")
  })
  void clutch();

  @Transitions({
    @Transition(from="TOP", to="DECLUTCHED", whenResult="true"),
    @Transition(from="TOP", to="CLUTCHED", whenResult="false"),
        @Transition(from="DECLUTCHED", to="<self>", whenResult="true"),
        @Transition(from="CLUTCHED", to="<self>", whenResult="false")
    })
    boolean isDeclutched();
}
```

State chart (excluding self transitions):



Discussion This model has substantially more states and transitions than any of the other models in the experiment, and exhibits the interaction of all the main features of Hanoi: inheritance, overriding, conditional transitions and self transitions. Despite this, it has a very regular, predictable structure, which should help a participant to quickly build a working mental model of its behaviour.

Neither model is significantly shorter or neater than the other, though it was expected that the presentation in the DSL model would make the regular structure more apparent.

Question 3A

After an accident, the event log from the automobile ended with the following sequence of method calls:

```
declutch();
shiftUp();
clutch();
```

```
4 declutch();
5 shiftUp();
```

The final call to `shiftUp` resulted in a typestate violation. What state must the `GearBox` instance have been in at the start of the sequence?

Options: (A) TOP, (B) DECLUTCHED, (C) DC_GEAR_N, (D) DC_GEAR_1, (E) DC_GEAR_2, (F) CLUTCHED, (G) C_GEAR_N, **(H) C_GEAR_1**, (I) C_GEAR_2.

Discussion This question is unique in the experiment in that it asks participants to reason about a sequence of method calls, knowing that it causes a typestate violation but without any knowledge of the start or end states. This is similar to the kind of diagnostic abstract reasoning an engineer may be expected to undertake in response to a bug report.

A possible approach to solving this issue is to first identify all states in which `shiftUp` is not permitted — `DC_GEAR_2` and its parents, `CLUTCHED` and its children. The call to `declutch` on line 5 was permitted, and this method is only permitted in `CLUTCHED` and its children, and guarantees that the object will be in one of the `DECLUTCHED` states. Therefore, the object must have been in state `DC_GEAR_2` prior to the call to `shiftUp` on line 6.

The calls can be traced back from there:

Line	Method	State prior to call
5	—	`DC_GEAR_2`
4	declutch	`C_GEAR_2`
3	clutch	`DC_GEAR_2`
2	shiftUp	`DC_GEAR_1`
1	declutch	`C_GEAR_1`

It was expected that participants would find this question difficult, and that their method of deriving an answer would provide some useful insight into their mental model of typestate generally.

Question 3B

Consider the method `safeReset`, which is intended to take a `GearBox` in any state and safely put it into state `C_GEAR_N`:

```
public void safeReset(GearBox gb) {
  if(!gb.isDeclutched()) {
```

```
3       gb.declutch();
4     }
5     gb.neutral();
6     gb.clutch();
7   }
```

Could a typestate violation occur? If so, please indicate on which line the violation will occur. If no violation will occur, choose “no violation”.

Options: (A) Line 2, (B) Line 3, (C) Line 5, (D) Line 6, **(E) No violation**

Discussion The method `isDeclutched` is permitted in the root state, and therefore in all states through inheritance. If it returns `true`, then the object is at least in one of the `DECLUTCHED` states, otherwise it is in one of the `CLUTCHED` states. In the latter case, the call to `declutch` on line 3 will transition the object to one of the `DECLUTCHED` states. Therefore, prior to the call on `neutral` on line 5, we are guaranteed that the object is in one of the `DECLUTCHED` states, all of which permit a call to `neutral` as this is inherited from the definition on state `DECLUTCHED`. This will result in a transition to `DC_GEAR_N` specifically regardless of which `DECLUTCHED` state the object is in, as there are no overrides of this transition. The final call to `clutch` on line 6 will result in a transition to `C_GEAR_N`. Therefore, no typestate violation will occur.

Line	State prior to call	Method	
2	X <: TOP	isDeclutched	↙
3	X <: CLUTCHED	declutch	↙
5	X <: DECLUTCHED	neutral	↙
6	DC_GEAR_N	clutch	↙
—	C_GEAR_N		

Deriving the correct answer for this question relies on the participant’s ability to reason about inheritance and overriding of transitions correctly, where the state of the object is not accurately known. The regular structure of the model and relationship between the method and state names should make the task manageable.

Question 3C

Consider the method `topGear`, which is intended to take a `GearBox` in any state and put it into state `C_GEAR_2`:

```
1   public void topGear(GearBox gb) {
2     if(!gb.isDeclutched()) {
```

```
3       gb.declutch();
4     }
5
6     switch(gb.currentGear()) {
7       case NEUTRAL: gb.shiftUp();
8       case ONE:     gb.shiftUp();
9     }
10
11    gb.clutch();
12  }
```

Will this method always result in the `GearBox` instance reaching state `C_GEAR_2`?

Options: **(A) Yes**, (B) No

Discussion Similar to the previous question, the participant must be able to reason effectively about inheritance and overriding. One added complexity is the fall-through in the control flow of the switch statement from the `NEUTRAL` case to the `ONE` case.

Line	State prior to call	Method	
2	X <: `TOP`	isDeclutched	⤦
3	X <: `CLUTCHED`	declutch	⤦
6	X <: `DECLUTCHED`	currentGear	⤦
7	`DC_GEAR_N`	shiftUp	⤦
8	`DC_GEAR_1`	shiftUp	⤦
11	`DC_GEAR_2`	clutch	⤦
—	`C_GEAR_2`		

Question 3D

Which method is not available in the `DC_GEAR_1` state?

Options: (A) shiftUp, (B) shiftDown, (C) neutral, (D) clutch, **(E) declutch**, (F) isDeclutched, (G) currentGear

Discussion The `DC_GEAR_1` state explicitly permits the methods `shiftUp`, `shiftDown` and `clutch`. It inherits transitions for `currentGear` and `neutral` from `DECLUTCHED`, and transitions for `isDeclutched` from `TOP`. Therefore, the method `declutch` is the only method not permitted in this state.

5.3.4 Model 4 — Iterator

An `Iterator` is a simple abstraction of a means of traversing and filtering a data structure in a sequential fashion. In Java, its interface definition is given by:

```java
public interface Iterator<T> {
  T next();
  boolean hasNext();
  void remove();
}
```

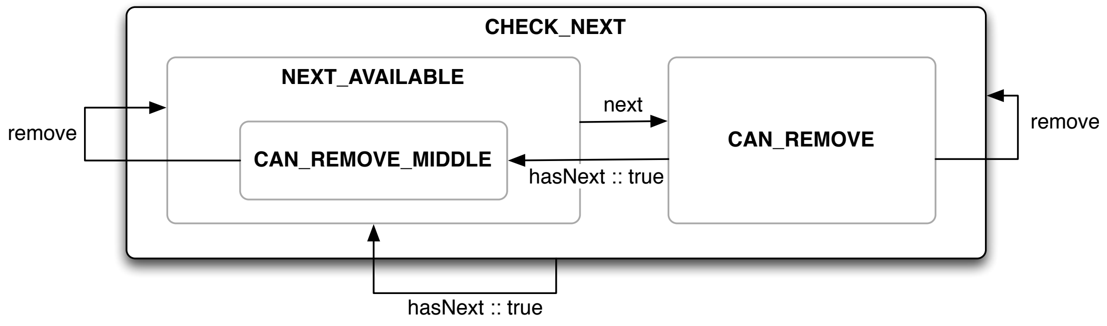
Hanoi DSL model:

```
CHECK_NEXT {
  NEXT_AVAILABLE {
    CAN_REMOVE_MIDDLE { remove() -> NEXT_AVAILABLE }
    next() -> CAN_REMOVE
  }

  CAN_REMOVE {
    hasNext() :: true -> CAN_REMOVE_MIDDLE
    remove() -> CHECK_NEXT
  }

  hasNext() :: true -> NEXT_AVAILABLE
  hasNext() :: false -> <self>
}
```

Hanoi annotation model:

```java
@StateModel({
  @State(name="CHECK_NEXT"),
  @State(name="NEXT_AVAILABLE", parent="CHECK_NEXT"),
  @State(name="CAN_REMOVE_MIDDLE", parent="NEXT_AVAILABLE"),
  @State(name="CAN_REMOVE", parent="CHECK_NEXT")
})
public interface Iterator<T> {

  @Transitions({
    @Transition(from="CHECK_NEXT",
                to="NEXT_AVAILABLE", whenResult="true"),
    @Transition(from="CAN_REMOVE",
                to="CAN_REMOVE_MIDDLE", whenResult="true"),
    @Transition(from="CHECK_NEXT", to="<self>", whenResult="false")
  })
  boolean hasNext();

  @Transitions({
    @Transition(from="NEXT_AVAILABLE", to="CAN_REMOVE")
  })
  T next();

  @Transitions({
    @Transition(from="CAN_REMOVE", to="CHECK_NEXT"),
    @Transition(from="CAN_REMOVE_MIDDLE", to="NEXT_AVAILABLE")
  })
  void remove();
}
```

State chart (excluding self transitions):



Discussion This final model is the classic `Iterator` interface from Java. While compact, this interface contains some subtle restrictions that Java programmers are often not aware of. The single-use enabling of the `remove` method in response to a call to `next` is often a source of confusion, where programmers occasionally expect that `remove` is idempotent.

This model was included in the experiment as it provides an opportunity to see whether programmers still explicitly check the model or base their decisions on their intuitive understanding of the interface, even when this is potentially incorrect.

Question 4A

Consider the method `printAll`:

```
void printAll(Iterator<T> iter) {
  while(iter.hasNext()) {
    System.out.println(iter.next());
  }
}
```

An `Iterator` instance in state `CHECK_NEXT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If no violation will occur, choose “no violation”.

Options: (A) Line 2, (B) Line 3, **(C) No violation**

Discussion This question is a straightforward and intuitively correct with the understanding that most Java programmers have of the `Iterator` interface. Alternate calls to `hasNext` and `next` are permitted when `hasNext` returns true.

Question 4B

Which method is not available in the `CAN_REMOVE` state?

Options: (A) hasNext, **(B) next**, (C) remove

Discussion The `CAN_REMOVE` state explicitly permits the `remove` method and `hasNext` method. The `next` method is the only method that is not permitted.

Question 4C

Consider the method `removeMiddle`, which will remove all elements from a collection which are not the first or last.

```
 1 void removeMiddle(Iterator<T> iter) {
 2   if(!iter.hasNext()) {
 3     return;
 4   }
 5
 6   iter.next();
 7   while(iter.hasNext()) {
 8     iter.next();
 9     if(iter.hasNext()) {
10       iter.remove();
11     }
12   }
13 }
```

An `Iterator` instance in state `CHECK_NEXT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If no violation will occur, choose “no violation”.

Options: (A) Line 2, (B) Line 6, (C) Line 7, (D) Line 8, (E) Line 9, (F) Line 10, **(G) No violation**

Discussion The points of interest are the conditional behaviour on lines 2, 7 and 9. There are four scenarios to consider:

- If there are no elements available in the iterator, the method exits early on line 3.
- If there is only one element available, the body of the loop does not execute and the object is left in state `CAN_REMOVE`.
- If there are two elements available, the body of the loop executes but the call to `remove` does not occur, the loop does not execute a second time and the object is left in state `CAN_REMOVE`.
- If there are three or more elements, the body of the loop executes and after the first execution the object is in state `NEXT_AVAILABLE`. The call to `hasNext` will result in no transition, leaving the object in the same state on line 8 as on the first execution of the loop.

No typestate violation will occur.

To derive the correct answer, the participant must correctly reason about the conditional behaviour in this manner, tracing the multiple paths through the code.

Question 4D

Consider the method `removeLast`, which will remove the last element from a collection.

```
1 void removeLast(Iterator<T> iter) {
2   while(iter.hasNext()) {
3     iter.next();
4   }
5   iter.remove();
6 }
```

An `Iterator` instance in state `CHECK_NEXT` will be passed into the method. Will a typestate constraint be violated? If so, please indicate on which line the violation will occur. If no violation will occur, choose “no violation”.

Options: (A) Line 2, (B) Line 3, **(C) Line 5**, (D) No violation

Discussion If the iterator contains no elements (i.e. the first call to `hasNext` returns false), then the iterator will be in state `CHECK_NEXT` prior to the call to `remove`, which is not permitted in this state. Therefore, a typestate violation can occur on line 5.

5.3.5 Survey

In the survey at the end of the experiment, the participants were asked to state their level of agreement with the following questions, on a standard five point Likert scale labelled from strongly disagree through neutral to strongly agree:

1. *I write code that interacts with typestate constraint interfaces.* This question was included as a way to gauge whether the participants believed that typestate, regardless of whether it was formally presented or not in the language they used, was something that they regularly had to take into account.

2. *I write classes which have typestate constraints of their own.* With language features such as generics in Java, a programmer is much more frequently going to act as a consumer of generic types than as a producer — the feature is mostly applicable for collection types and frameworks, but the number of programmers responsible for writing such APIs is very small compared to the number who use them. Typestate may or may not fit the same usage pattern as generics, and this question was intended to determine whether participants felt they were both producers and consumers of typestate by comparing answers to this question to the previous one.

3. *The inheritance rules of Hanoi are easy to understand.* After an hour of working with Hanoi, determining whether participants felt that the inheritance rules (arguably the most complex part of the semantics of Hanoi) were easy to work with would provide information on whether or not the complexity is justified by the brevity of expression it affords compared to a flat state machine model.

4. *I find it easier to determine whether a method is legal in a state in the annotation form.* It was anticipated that, as the state tree is less readily apparent in the annotation model, determining whether a method is legal or not in a given state would be harder in the annotation form than in the DSL form. As such, it was expected that participants would lean towards the disagree end of the spectrum for this question.

5. *I find it easier to determine what state the object will be in after calling a method in the DSL form.* It was anticipated that the visual hierarchical presentation of transition overrides in the DSL form would make it easier for participants to identify which transition is relevant for a given state, and therefore to determine the correct target state after a method call. As such, it was expected that participants would lean towards the agree end of the spectrum for this question.

6. *I find it easier to visualise the state tree with a model defined in the annotation form.* It was anticipated that the hierarchy should be much more readily apparent in the DSL

form than in the annotation form, for all but the most trivial hierarchies. As such, it was expected that participants would lean towards the disagree end of the spectrum for this question.

7. *I find it easier to determine whether code conforms to the specification with the DSL form.* This very broad question was intended to determine the participant's overall view of the two notations, for the primary task presented in this experiment of assessing the correctness of code against a model.

8. *I find it easier to determine which legal methods are inherited by a state in the annotation form.* This question specifically evaluates whether the participants felt that inherited method sets are easier to determine using the annotation or DSL forms. It was anticipated that the visual hierarchical presentation of inheritance in the DSL form would make this particular task much easier. As such, it was expected that participants would lean towards the disagree end of the spectrum for this question.

5.4 Results

Sourcing suitably qualified candidates (those with sufficient programming experience, generally excluding undergraduate students) proved difficult, hence the experiment was only conducted with 10 participants, all of whom were either masters degree students or PhD students in Computer Science from Carnegie Mellon University or Glasgow University.

5.4.1 Statistical tests & measures used

The alternatives to the null hypotheses presented in Section 5.2.1 in which we are interested are *non-directional*. When the null hypothesis states that there will be no difference in performance based on the Hanoi model type provided to the participant, the alternative is that *some* difference in performance will be observed, and not specificially that the DSL model group will perform *better* than the annotation model group. The former is non-directional, while the latter is directional. Statistical tests for non-directional alternative hypotheses are known as *two-tailed* tests.

Analysis of ordinal data samples

With a small sample size and no prior knowledge of the expected form of the results which will be collected, it is desirable to use non-parametric statistical tests — those which do not require that the data fit a particular probability distribution. The *Mann-Whitney U* test [89] is a reliable non-parametric statistic that can be used for the ordinal data collected in the experiment, and is easy to calculate for small sample sizes.

Given two groups a and b of size n_a and n_b, such that samples derived from the members of each group are independent, we can calculate a statistic U such that $0 \le U \le n_a \times n_b$. For example, consider groups such that $n_a = n_b = 5$ and the samples collected are times in seconds to complete a task:

Time:	72	73	91	125	136	247	302	327	337	343
Group:	A	B	B	B	B	A	A	A	B	A

U can be calculated by the following procedure:

1. Sort the samples. The above table is already sorted.
2. For each sample a_i from group a, count the number of samples which are less than it that belong to group B. If samples exist which are equal to a_i from group b, count these as 0.5. Refer to this sum as a'_i.
3. Add the counts together: $U_a = \Sigma_{i=1}^{n_a} a'_i$.

U_b can be similarly calculated, however the test is defined such that $U_a + U_b = n_a \times n_b$. The statistic $U = min(U_a, U_b)$.

From the sample data above, we would derive that $U = 8$, as $U_a = 17$ and $U_b = 8$:

Time:	72	73	91	125	136	247	302	327	337	343
Group:	A	B	B	B	B	A	A	A	B	A
a'_i:	0					4	4	4		5
b'_i:		1	1	1	1				4	

The value of U which would indicate that the null hypothesis is false is determined by the group sizes. For small group sizes ($n \le 20$), such as those found in this experiment, a lookup table is used to determine the two-tailed significance threshold corresponding to a type I error rate of $\alpha = 0.05$ — this is the probability of incorrectly rejecting the null hypothesis based on the observed data. We shall represent the threshold value as U^* wherever the U test statistic is discussed. For $n_a = n_b = 5$, the threshold is $U^* = 2$.

A corresponding measure of the relation between the samples of the two groups can be easily calculated using the formula $\rho_a = U_a/(n_a \times n_b)$. ρ gives a value on a fixed range while U does not, therefore it can be useful for determining the relation between two groups at a glance. A value of $\rho_a = 0.5$ indicates perfect overlap (the groups are effectively indistinguishable), while $\rho_a = 0$ indicates perfect separation, with all of group a preceding group b (with $\rho_a = 1$ meaning group b precedes group a). There is no fixed ρ value for significance, therefore it it not a meaningful statistic on its own — for example, if $n_a = n_b = 10$, then the significance threshold for ρ would be 0.23 as $U^* = 23$ for such a sample size. If $n_a = 20$, $n_b = 4$ then $\rho \leq 0.175$ indicates significance as $U^* = 14$.

Values such as the mean, median and standard deviation values for reading or question answer times for each group shall be presented where appropriate. Generally, a subscript of D denotes the DSL group (those who were presented a DSL based model), while a subscript of A denotes the annotation group (those who were presented an annotation based model). The symbols $\overline{t_A}$ and $\overline{t_D}$ are used to denote mean times, $\tilde{t}_A$ and $\tilde{t}_D$ denote the median times, while σ_A and σ_D denote the standard deviations.

Analysis of contingency tables

In analysing results for a correlation between model type and performance in answering individual questions, Fisher's exact test is suitable due to the small sample size and the categorical structure of the data. The data can be arranged as a *2x2 contingency table*, relating the model type shown to the number of correct / incorrect answers observed against that model type.

For instance, consider the following contingency table which relates groups A and B to some measured binary variable v:

Contigency Table A		
	$v = 0$	$v = 1$
Group A	m	n
Group B	o	p

The sizes of the groups are given by $n_a = m + n$ and $n_b = o + p$. The total number of samples where $v = 0$ is given by $v_0 = m + o$, and similarly $v_1 = n + p$.

Fisher's exact test determines the probability of observing such data under the null hypothesis, that groups A and B are the same:

$$p(A) = \frac{\binom{n_a}{m}\binom{n_b}{o}}{\binom{n_a+n_b}{v_0}}$$

In order to derive a significance level which can be used to decide whether to reject the null hypothesis, the Fisher test requires that we must also consider all other tables which are *at least* as extreme as the observed table. Consider the following table:

Contigency Table B		
	v = 0	v = 1
Group A	q	r
Group B	s	t

This table is at least as extreme as the previous table if the following conditions hold:

$$m + o = q + s \quad n + p = r + t$$
$$m + n = q + r \quad o + p = s + t$$
$$p(B) \leq p(A)$$

The significance value for the observed data is the sum of the probabilities of all tables at least as extreme as the observed table: $p = \Sigma\{p(T) \mid p(T) \leq p(A)\}$. The threshold used for rejecting the null hypothesis where the Fisher test is used in the following analysis is $p \leq 0.05$.

The Fisher test can be used for samples of any size. It is better suited to smaller sample sizes due to the exponential growth of the binomial terms involved in the calculation of $p(T)$ — for large values in the contingency table, the accurate calculation of p becomes prohibitively expensive.

The Pearson χ^2 test

Another common test used for categorical data is the Pearson χ^2 test [116]. This test assigns an *expected* value E_i for each cell in the contingency table based on the null hypothesis, while O_i denotes the observed value for each cell in the table (which contains n cells). From this, the statistic χ^2 is calculated using the formula

$$\chi^2 = \Sigma_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

This value is then compared to the standard χ^2_k probability distribution, where k is the number of *degrees of freedom* in the data — for a 2x2 contingency table where the size of the two groups are fixed, there is one degree of freedom: knowing the value of one cell in the table is sufficient to derive the values of all other cells. Similarly for a fixed sample size of responses against a five-point Likert scale, there are 4 degrees of freedom: if we know the number of responses for four of the columns, the value of the final column can be derived.

Let $CDF(\chi^2_k, x)$ be a function which provides the probability of observing a value between 0 and x for the χ^2_k probability distribution function. Then $p(x) = 1 - CDF(\chi^2_k, x)$ gives the probability of observing a value between x and $\inf$. $p(\chi^2)$ therefore provides the significance value for an observed χ^2 statistic under the null hypothesis — if $p(\chi^2) < 0.05$, we may reject the null hypothesis.

The Pearson χ^2 test is known to be potentially inaccurate for small sample sizes, particularly if any E_i is smaller than 5. This is unfortunately the case for the majority of the results in this experiment, as such the Fisher test is better suited to the analysis of 2x2 contingency tables while another approach is used for the analysis of survey results, discussed below.

Analysis of survey results

The survey results collected are a form of *multinomial* data — the responses that participants provide to questions are based on a five point Likert scale, which is a a form of categorical (ordinal, but arguably not interval) data. Participants are not distinguished by which group they were assigned to for this analysis; they are treated as a single group as it was believed this would be an irrelevant detail for the questions presented.

The responses are analysed on a per-question basis, against an expected distribution consistent with the null hypothesis: we expect participants to consistently choose values closer to neutral than the extreme ends of the scale (responses to Likert scale questions suffer from a *central tendency bias*, where participants avoid extreme responses), with an equal distribution of agree and disagree responses. A categorical probability distribution such that we expect 10% of participants to choose strongly disagree, 25% to choose disagree, 30% to choose neutral, 25% to choose agree and 10% to choose strongly agree is a reasonable formulation of this.

Due to the small sample size, Pearson's chi-square test of goodness-of-fit is not directly suitable, as the expected frequencies for all of the responses are below 5. A randomization test of goodness-of-fit is used instead to derive a probability that the observed distribution of responses fits the expected distribution. First, the χ^2 value is derived for the observed data, and then thousands of samples of the same sample size are randomly drawn from the expected categorical probability distribution. The observed fraction of these randomly drawn samples which have a χ^2 value greater than that observed in the real data provides the significance value for the data. The more random samples generated, the greater the confidence that the derived p-value is correct.

As an example, consider the following responses to a Likert scale question, with the expected distribution also displayed:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Observed:	1	1	1	5	2
Expected %:	10%	25%	30%	25%	10%

The χ^2 value for the above data is 5.733. By generating 10000 random samples from the expected distribution, and calculating the fraction of those which have a $\chi^2 \geq 5.733$, we may derive the significance value of $p = 0.22$.

The significance of $p < 0.05$ will be used to reject the null hypothesis, therefore in the above example the null hypothesis could not be rejected.

Computational support

The R project for statistical computing [76, 77] provides a straightforward means of computing the statistics described above. Firstly, the Mann-Whitney U test (also known as the *Wilcoxon-Mann-Whitney* test) can be computed for the sample data for groups a and b used in the discussion using the following commands:

```
> a = c(72, 247, 302, 327, 343)
> b = c(73, 91,  125, 136, 337)
> U_a = wilcox.test(a, b, alternative="two.sided")$statistic
> U_b = wilcox.test(b, a, alternative="two.sided")$statistic
> min(ua, ub)
[1] 8

> wilcox.test(a, b, alternative="two.sided")$p.value
[1] 0.4206349
```

As demonstrated, the significance value can be directly derived for the data in addition to the U value.

Fisher exact test To calculate the Fisher exact test for a 2x2 contingency table, the following command can be used:

```
> table = rbind(c(6,4),c(1,9))
> table
     [,1] [,2]
[1,]    6    4
[2,]    1    9

> fisher.test(table, alternative="two.sided")$p.value
[1] 0.05728
```

Randomization test To calculate the significance value for survey results, R provides a randomization test of goodness-of-fit using the Pearson chi-square test, following the same approach as described previously. This can be used as follows:

```
> responses =  c(1,   1,    1,   5,    2)
> expected_p = c(0.1, 0.25, 0.3, 0.25, 0.1)
> chisq.test(responses,
             p = expected_p,
             simulate.p.value = TRUE,
             B = 100000)

  Chi-squared test for given probabilities with simulated
  p-value (based on 1e+05 replicates)

data:  responses
X-squared = 5.7333, df = NA, p-value = 0.2188
```

5.5 Participant demographics

At the start of the experiment, participants were asked to state the number of years of experience they have had as a programmer (professionally or otherwise), and their proficiency (one of low, medium or high) with object oriented programming, functional programming and formal methods. Their responses are displayed in Figure 5.2.

The demographic groupings were as follows:

- 70% of participants rated themselves as of "medium" proficiency with object oriented programming, while 30% rated themselves as of "high" proficiency. This is as expected, as candidates for the experiment were filtered based on their ability to reason about non-trivial aspects of the Java programming language.
- 40% of participants rated themselves as of "low" proficiency with functional programming, while 60% rated themselves as of "medium" proficiency.
- 30% of participants rated themselves as of "low" proficiency with formal method related tools, while 70% rated themselves as of "medium" proficiency.

An analysis of whether self-rated proficiency and experience are correlated with overall score is presented in Section 5.5.1.

PID	Prog. Experience (Years)	OOP Prof.	FP Prof.	FM Prof.
1	6	Medium	Medium	Medium
2	4	Medium	Medium	Medium
3	3	Medium	Low	Medium
4	12	High	Medium	Medium
5	12	Medium	Medium	Medium
6	5	High	Low	Medium
7	12	Medium	Low	Low
8	3	Medium	Low	Low
9	7	High	Medium	Low
10	5	Medium	Medium	Medium

Figure 5.2: Raw participant demographic information

PID	Q1A	Q1B	Q1C	Q1D	Q2A	Q2B	Q2C	Q2D	Q3A	Q3B	Q3C	Q3D	Q4A	Q4B	Q4C	Q4D	TOTAL
2 (A)	E	E	D	A	F	A	C	G	G	E	A	E	C	B	G	C	13 (81%)
4 (A)	C	E	D	A	F	A	C	F	H	E	A	E	C	B	G	C	16 (100%)
6 (A)	B	C	E	C	F	B	C	G	F	E	A	E	C	B	C	C	8 (50%)
8 (A)	C	C	E	A	F	A	C	G	H	E	A	E	C	B	G	C	14 (88%)
10 (A)	C	E	D	A	F	A	C	F	H	E	A	E	C	B	G	C	16 (100%)
1 (B)	C	D	E	B	A	B	C	G	H	E	A	E	C	B	G	C	10 (63%)
3 (B)	D	D	D	A	F	A	C	G	H	E	A	E	C	B	G	—	11 (69%)
5 (B)	C	E	E	A	F	A	C	G	H	E	B	E	C	B	G	C	13 (81%)
7 (B)	D	E	D	A	F	A	C	G	H	E	A	—	C	B	D	C	12 (75%)
9 (B)	C	E	D	A	F	A	C	G	H	E	B	E	C	B	G	C	14 (88%)
p:	1	1	1	1	1	1	1	0.4	0.4	1	0.4	1	1	1	1	1	**0.69**

Figure 5.3: Choices and scores. Incorrect answers are highlighted with a black background. Questions correctly answered against a DSL model are highlighted with a grey background. The p row indicates the probability that the model type and accuracy with which participants answers questions are independent variables.

5.5.1 Scores

The responses to questions from each participant are shown in Figure 5.3. The outcome of the experiment was generally positive, with a mean score of 76% and a median score of 81%, with the lowest observed score at 50%. This provides evidence that participants were able to reason effectively about the models after only a 20 minute tutorial. Participants began to answer questions with greater accuracy towards the end of the experiment, with the majority of incorrect answers occurring in the questions relating to the first two models.

There is no evidence that the model type presented to a participant is correlated with their accuracy in answering questions related to that model. All but three questions had answer accuracy statistics that were entirely consistent with the null hypothesis:

- 2D — only two participants answered correctly, and both were presented an annotation model. These two participants (PIDs 4 and 10) were also the highest performing across all participants , for whom the model type presented had no impact on their accuracy. All other participants answered incorrectly in the same way, missing the possibility that the loop body in this question may not execute and therefore missing the potential violation on the final line of the method.
- 3A — two participants answered incorrectly, and both were presented with a DSL model.
- 3C — two participants answered incorrectly, and both were presented with an annotation model. This is the reverse of the situation for question 3A.

In all three cases, the significance value for the observed data pattern was $p = 0.4$, which is insufficient to reject the null hypothesis. In the case of 2D, it does not seem likely that the ability to answer this question correctly was influenced by the model, but instead on the participant's ability to reason about control flow. The results of 3A and 3C contradict each other, providing further evidence that the results are due to chance rather than any meaningful correlation between performance and model type.

The relationship between the model type shown and the accuracy of response across all questions is as follows:

	Correct	Incorrect
DSL	62	18
Annotation	65	15

Using the Fisher exact test, $p = 0.69$ that this data distribution would be observed under the null hypothesis, therefore the results as a whole provide no evidence for a correlation between model type and accuracy.

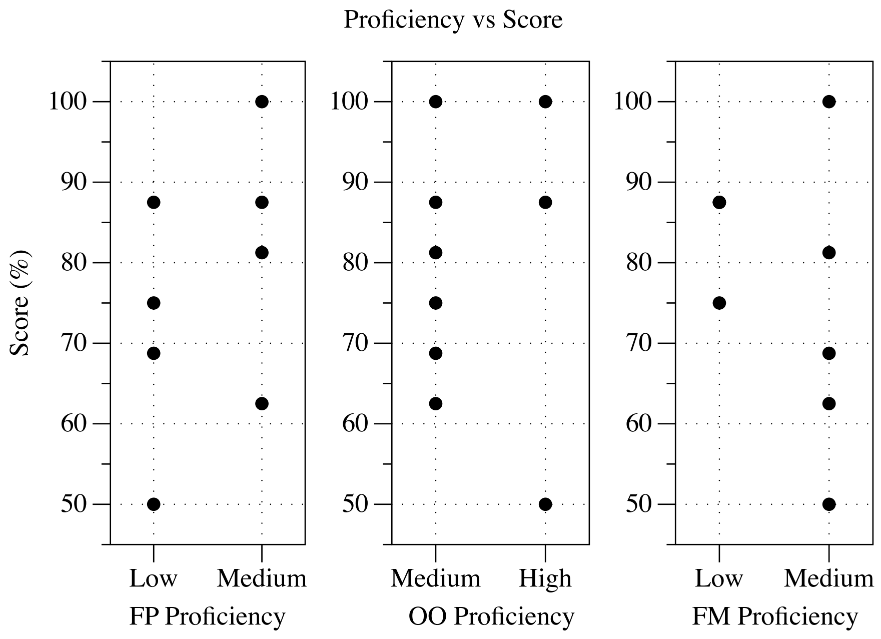


Figure 5.4: Scatter plot of self-rated proficiencies against score.

Proficiency vs. score Scatter plots of experience and proficiency against score are shown in Figure 5.5 and Figure 5.4 respectively.

There is no evidence of any correlation between experience and score, and no evidence of a correlation between object oriented programming proficiency and score (U = 12, U^* = 1, ρ = 0.48) or formal methods knowledge and score (U = 8, U^* = 1, ρ = 0.32). There is a weak signal of correlation between functional programming proficiency and score (U = 5, U^* = 2, ρ = 0.2), but still not at a statistically significant level.

5.5.2 Reading times

The time taken for each participant to read the models is shown in Figure 5.6, with a scatter plot of these values shown in Figure 5.7.

There is no evidence of a correlation between model type and time spent reading a model. A large difference in mean time to read all questions between groups A and B is apparent, with group A being significantly faster on average. This unfortunate random allocation of participants to groups results in faster mean reading times for the DSL model type for models 1 and 3, and faster mean reading times for the annotation model type for models 2 and 4.

The difference in reading times on a per person basis is attributable to the different approaches participants took in attempting to gain an understanding of the model. The slowest participants would often trace through every transition of the model, and draw out statechart-like diagrams to aid their understanding. The faster participants would often do little more

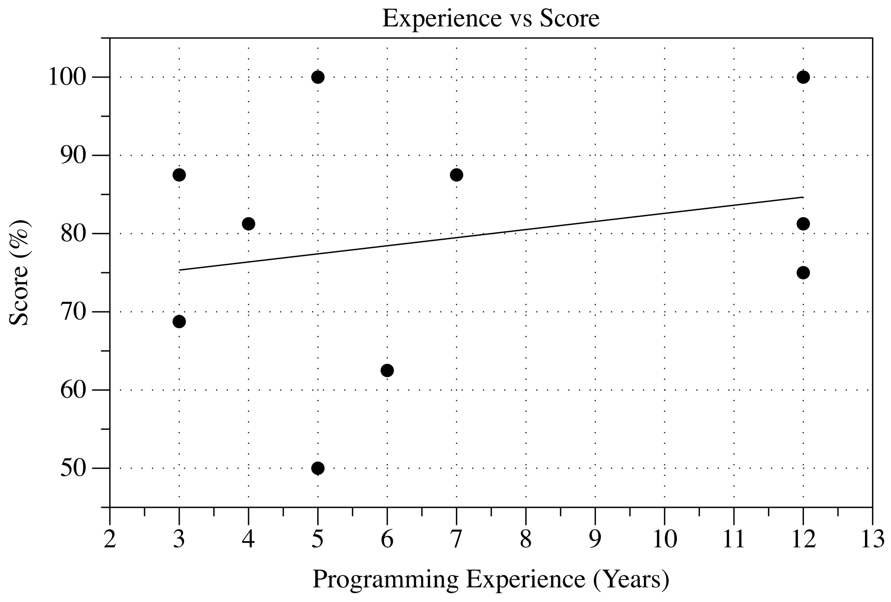


Figure 5.5: Comparison of programmer's stated experience as a programmer and their performance in the test, with a line of best fit.

PID	Q1	Q2	Q3	Q4	TOTAL
1	5:02	3:33	6:39	2:41	17:55
2	5:37	0:07	0:04	1:53	07:41
3	4:07	7:21	5:54	4:13	21:35
4	2:16	2:43	4:05	2:17	11:21
5	5:43	4:48	6:20	3:14	20:05
6	1:13	0:31	0:44	0:58	03:26
7	5:27	2:14	2:20	1:34	11:35
8	2:05	2:18	1:12	0:49	06:24
9	1:12	1:18	1:51	0:45	05:06
10	1:31	1:11	1:14	0:45	05:05
$\bar{t}$:	3:25	2:38	3:02	1:54	
σ:	1:56	2:05	2:23	1:11	
$\tilde{t}$:	3:11	2:18	2:20	1:43	
U:	8	5	2	6	
$U*$:	2	1	1	2	
ρ_D:	0.32	0.75	0.1	0.76	

Figure 5.6: Question reading times, per participant. Times against models provided in DSL form are highlighted with a grey background. In two cases, participant 2 failed to follow instructions resulting in abnormally low reading times — these results were discarded from the analysis.

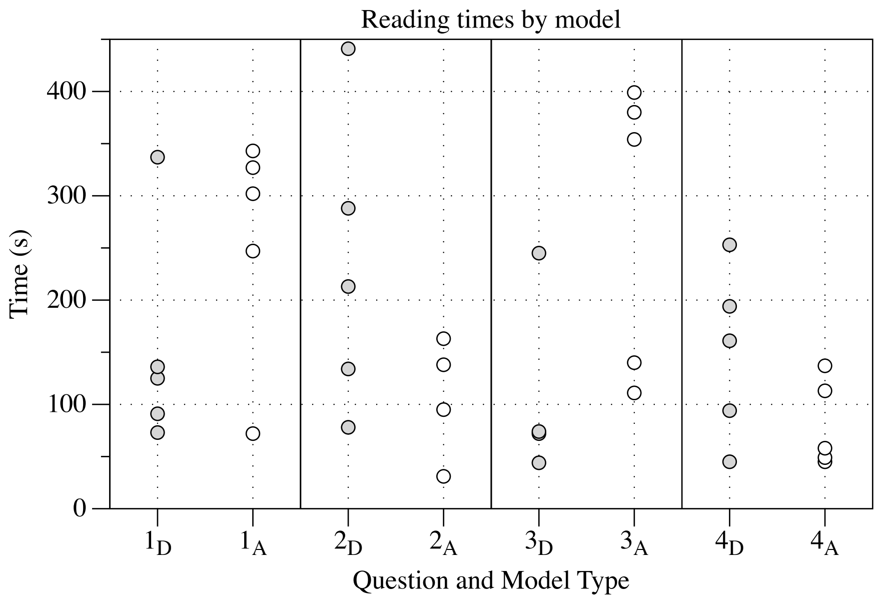


Figure 5.7: Scatter plot of question reading times. Grey dots (labelled with subscript D on the x-axis) show times recorded for participants given a DSL model. White dots (labelled with a subscript A on the x-axis) show times for those given an annotation based model.

than skim the model before moving on to the questions. Participant 2 was a particularly extreme example — after being the second slowest participant in reading the first model, they changed strategy and spent less than 10 seconds reading models 2 and 3. The participant justified this behaviour by stating that they preferred to read the models in the context of a specific question, rather than attempting to read and understand them in isolation as directed.

Consequently, the reading times do not appear to be particularly useful in determining the difference between the models, as the individual reading styles of the participants are likely to dominate any smaller effect that the models themselves have.

5.5.3 Answer times

The time taken for each participant to read and answer the questions related to each model is shown in Figure 5.8, with a scatter plot of these values shown in Figure 5.9.

PID	Q1A	Q1B	Q1C	Q1D	Q2A	Q2B	Q2C	Q2D	Q3A	Q3B	Q3C	Q3D	Q4A	Q4B	Q4C	Q4D	ALL
2 (A)	5:38	4:02	4:55	1:16	2:43	3:05	3:09	2:36	9:09	2:11	1:35	1:02	2:00	1:58	5:41	4:10	55:10
4 (A)	3:51	3:35	2:27	2:08	1:47	0:35	1:48	2:01	3:40	4:02	4:03	0:56	3:47	0:47	8:17	1:38	45:22
6 (A)	1:59	1:44	1:27	1:04	2:00	0:43	2:40	1:45	0:40	1:16	1:47	0:39	1:19	0:26	1:23	0:26	21:18
8 (A)	2:30	2:46	1:34	1:14	1:48	1:35	2:39	3:12	3:39	2:30	3:19	0:51	2:15	0:56	5:29	1:12	37:29
10 (A)	1:10	1:44	2:10	1:16	1:11	0:48	1:51	1:38	2:45	2:23	2:05	0:57	1:59	0:35	3:38	0:46	26:56
1 (B)	5:45	2:19	1:47	1:56	1:55	0:22	2:43	1:26	3:19	2:20	3:46	1:07	2:17	0:28	3:35	2:00	37:05
3 (B)	7:28	2:32	4:18	1:53	1:47	1:29	1:43	2:11	3:55	4:11	2:27	2:14	1:59	1:45	6:21	0:07	46:13
5 (B)	5:28	7:18	3:32	2:31	1:23	1:43	1:59	3:39	1:35	1:16	1:51	1:31	1:34	0:19	3:12	1:23	40:14
7 (B)	1:59	2:52	2:29	1:21	1:31	2:06	2:41	1:34	6:14	3:50	4:51	0:59	2:26	0:33	5:26	1:19	42:11
9 (B)	4:54	2:49	1:52	0:50	2:25	1:00	2:00	3:00	4:04	1:58	1:44	0:48	1:10	0:36	2:22	1:01	32:33
$\bar{t}$:	4:04	3:10	2:39	1:33	1:51	1:21	2:19	2:18	3:54	2:36	2:45	1:06	2:05	0:50	4:32	1:33	38:27
σ:	2:05	1:37	1:12	0:32	0:27	0:50	0:30	0:46	2:22	1:04	1:10	0:27	0:44	0:34	2:04	1:05	09:50
$\tilde{t}$:	4:23	2:48	2:19	1:19	1:48	1:15	2:20	2:06	3:40	2:22	2:16	0:58	2:00	0:36	4:32	1:19	38:52
$\bar{t}_A$:	5:07	3:34	2:48	1:42	1:54	1:21	2:25	2:14	3:49	2:43	2:56	1:20	2:16	0:56	4:54	1:38	—
σ_A:	2:00	2:06	1:06	0:38	0:33	1:03	0:35	0:39	1:40	1:15	1:21	0:34	0:55	0:36	2:34	1:29	—
$\tilde{t}_A$:	5:28	2:49	2:29	1:53	1:48	0:48	2:39	2:01	3:55	2:20	2:27	1:07	2:00	0:47	5:29	1:12	—
$\bar{t}_D$:	3:02	2:46	2:31	1:24	1:48	1:20	2:13	2:22	3:59	2:28	2:34	0:53	1:53	0:44	4:11	1:26	—
σ_D:	1:45	1:03	1:24	0:25	0:24	0:40	0:27	0:57	3:08	0:60	1:04	0:09	0:31	0:35	1:39	0:25	—
$\tilde{t}_D$:	2:20	2:46	2:10	1:16	1:47	1:29	2:00	2:11	3:39	2:23	2:05	0:56	1:59	0:33	3:35	1:12	—
U:	5	10	9	8	10	11	12	12	10	12	10	5	11	8	9	12	—
ρ_D:	0.2	0.4	0.36	0.32	0.4	0.56	0.48	0.48	0.4	0.52	0.4	0.2	0.44	0.32	0.36	0.48	—

Figure 5.8: Question answer timings and statistical analysis. Times against models provided in DSL form are highlighted with a grey background. Times for incorrect answers are highlighted with a black background. $U^* = 2$ for each question.

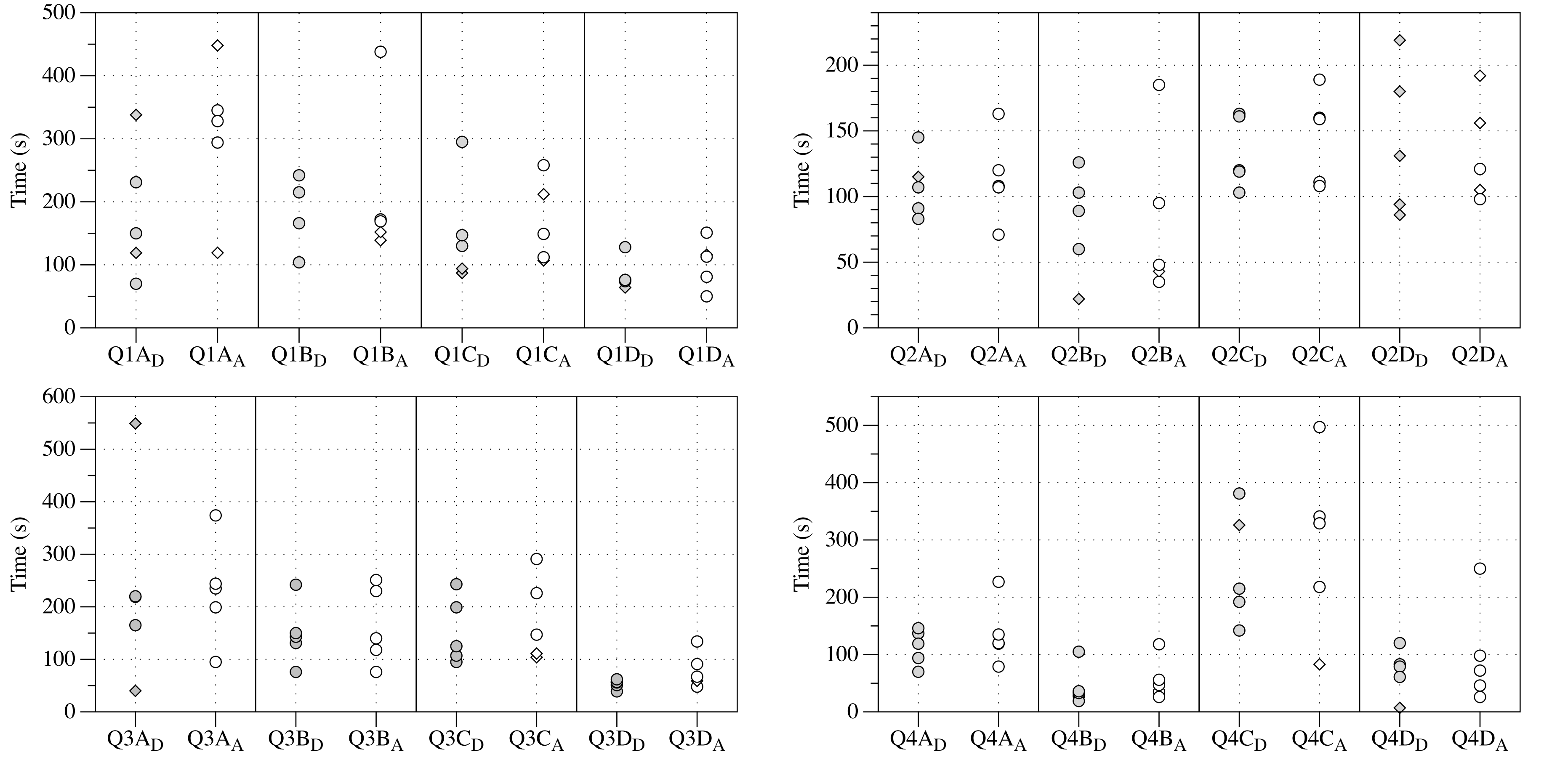


Figure 5.9: Scatter plot of question answer times. Grey markers (labelled with a subscript D on the x-axis) show times recorded for participants given a DSL model. White markers (labelled with a subscript A on the x-axis) show times for those given an annotation based model. Circular markers indicate correct answers while diamonds are incorrect answers.

There is no evidence of a correlation between the model type and the time spent answering questions pertaining to the model.

Unlike with the model reading times, there is no discernible difference in the mean answering times between groups A and B over all questions, as such the data was more likely to provide meaningful results. There is however still a great deal of variance in the time taken to answer each individual question between the participants. This is partly accounted for by the variance in the amount each participant said while thinking aloud, though differences in the strategy employed to answer a question and hesitance in committing to an answer dominated this: Some participants were confident with the first answer they reached and would immediately commit, while others would double- or triple-check their answer before committing to it.

In general, the mean answer times for the DSL group are faster than those for the annotation group, but this performance difference is not statistically significant. The answer times for Q1A and Q3D came the closest to a statistically significant difference between the model types to reject the null hypothesis ($U = 5$), but still not beyond the critical value of $U^{*} = 2$. Additional participants would be required for a definitive result, but based on the current results it does not appear that any statistically significant difference between the model types is likely.

5.5.4 Qualitative results

Qualitative data was collected in the form of notes taken during observation of participants undertaking the experiment, and recordings of their interaction with the experiment and statements made while thinking aloud. The recordings were later reviewed for the emergence of themes in the actions and statements of the participants. The qualitative data added a layer of insight to the quantitative results that helped to interpret the results of the experiment.

The following themes were observed:

- Most participants commented that the annotation form of presenting the model was “ugly” and typically larger than the DSL form — consequently, they found it harder to find specific declarations they were interested in.
- When exposed to the annotation form, participants complained that more steps were involved in determining whether one state was a descendant of another; consequently, more steps were involved in determining whether a transition was inherited by a state, compared to the other DSL models they were exposed to.
- Participants were adept at reasoning correctly about state transitions, but often failed to reach a correct answer by missing other possible paths through the code (i.e. missing

the possibility that the body of a conditional might never be executed, or a loop body may be executed more than once). It was unclear whether such mistakes were made as a result of the additional cognitive load that reasoning about typestate added, or if such mistakes would also have been made regardless in similar situations without typestate.

Overall, the participants seemed more confident while interpreting the DSL model than the annotation model, and while this did not translate into a statistically significant increase in performance, this does align with a preference for the DSL model as observed in the survey results.

5.5.5 Survey

The frequency table of responses to the survey questions with statistical analysis is shown in Figure 5.10, and frequency diagrams for each question are shown in Figure 5.11.

The survey results are the most definitive from the whole experiment — in general, participants preferred to work with the DSL model type over the annotation model type.

1. *I write code that interacts with typestate constrained interfaces* — The null hypothesis cannot be rejected ($p = 0.22$), though the data weakly supports the notion that most participants believe they write code that interacts with typestate, with 70% of participants choosing to agree with the statement, and 20% disagreeing.

2. *I write classes which have typestate constraints of their own* — The null hypothesis cannot be rejected ($p = 0.42$). Support for the statement is weaker than the previous statement, due to the higher incidence of neutral or disagree responses. This provides some evidence that users believe they are more likely to interact with code that has typestate constraints than write code that has typestate constraints, consistent with the usage pattern of a feature like Java generics.

3. *The inheritance rules of Hanoi are easy to understand* — The null hypothesis can be rejected ($p = 0.001$). Participants all agreed with the statement, with an equal split between agree and strongly agree.

4. *I find it easier to determine whether a method is legal in a state in the annotation form* — The null hypothesis cannot be rejected ($p = 0.25$). 60% of participants disagreed with the statement, indicating weak support for it being easier to determine whether a method is legal in a given state with the DSL model type.

5. *I find it easier to determine what state the object will be in after calling a method in the DSL form* — The null hypothesis cannot be rejected ($p = 0.12$). Weak support exists for the statement, with 70% of participants agreeing.

Response	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Meta:
Str. Disagree (1):	1	0	0	3	0	3	0	5	0
Disagree (2):	1	2	0	3	1	5	2	2	8
Neutral (3):	1	2	0	2	2	1	3	1	9
Agree (4):	5	5	5	2	4	1	4	2	18
Str. Agree (5):	2	1	5	0	3	0	1	0	15
Median:	4	4	4.5	2	4	2	3.5	1.5	4
Mode:	4	4	4, 5	1, 2	4	2	4	1	4
p (H_0):	0.22	0.42	0.001	0.25	0.12	0.04	0.78	0.01	0

Figure 5.10: Frequencies of responses to survey questions and statistical analysis

6. *I find it easier to visualise the state tree with a model defined in the annotation form* — The null hypothesis can be rejected ($p = 0.04$). 80% of respondents disagreed with the statement, representing a strong preference for the DSL form in this case.

7. *I find it easier to determine whether code conforms to the specification with the DSL form* — The null hypothesis cannot be rejected ($p = 0.78$) — indeed, the data indicates there may be no overall preference for either the DSL or annotation form for the general problem of checking the correctness, with only a slight lean towards the DSL form.

8. *I find it easier to determine which legal methods are inherited by a state in the annotation form* — The null hypothesis can be rejected ($p = 0.01$). 70% of participants disagreed with the statement, with a median response of "strongly disagree". This provides strong support for the DSL form being preferred for this task.

Meta analysis

Questions 4 through 8 all compare the DSL and annotation forms. By composing the responses of these questions it may be possible to provide an overall indication of whether participants preferred one model type to the other.

This can be done by inverting the responses to questions 4, 6 and 8 such that "agree" responses become "disagree", and summing all the responses from each category over all the questions. This shown in the "Meta" column of Figure 5.10, and the null hypothesis (that no overall preference is shown) can be rejected ($p = 0$). The meta-analysis strongly supports a preference towards the DSL model across all the tasks.

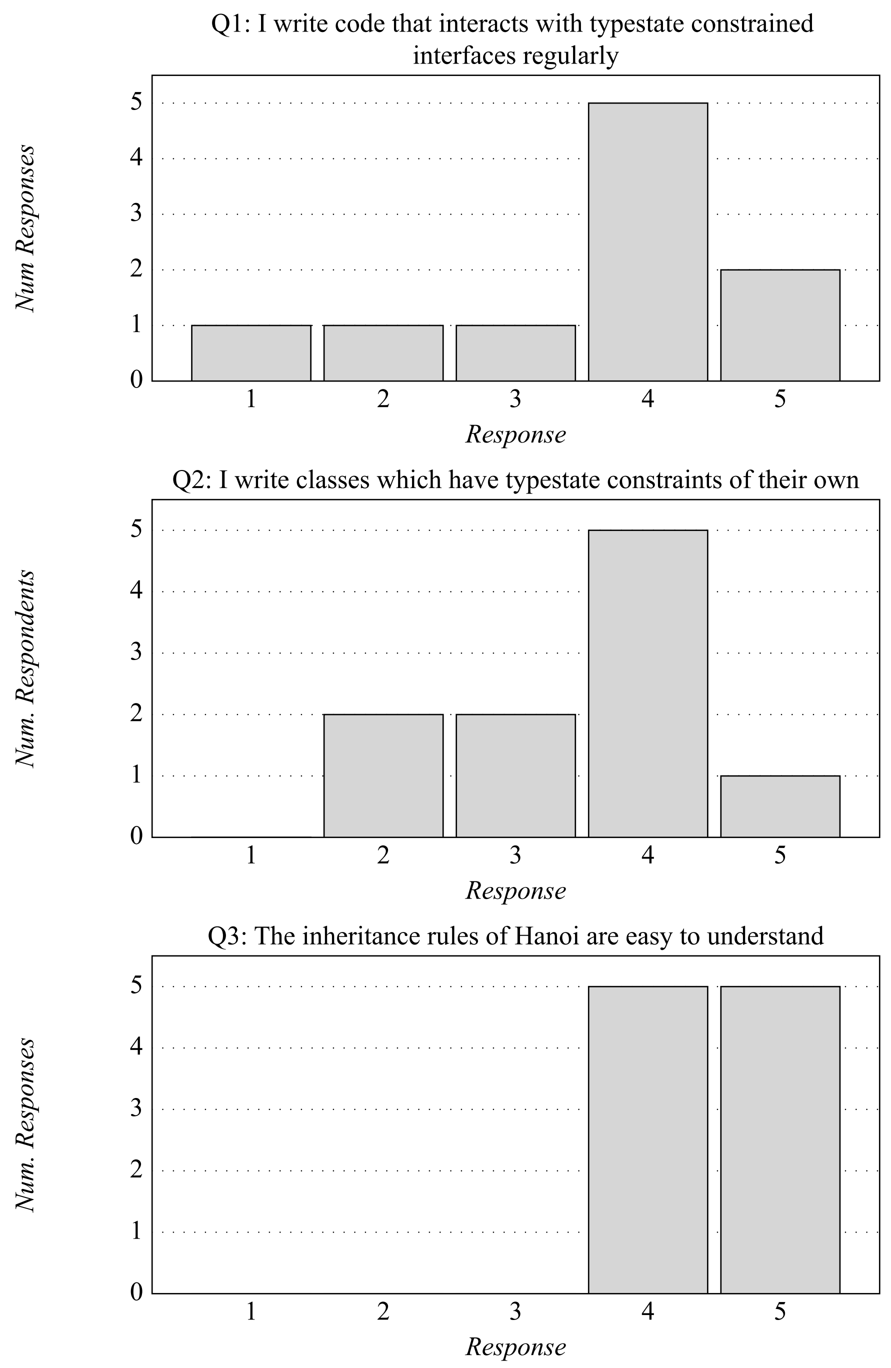
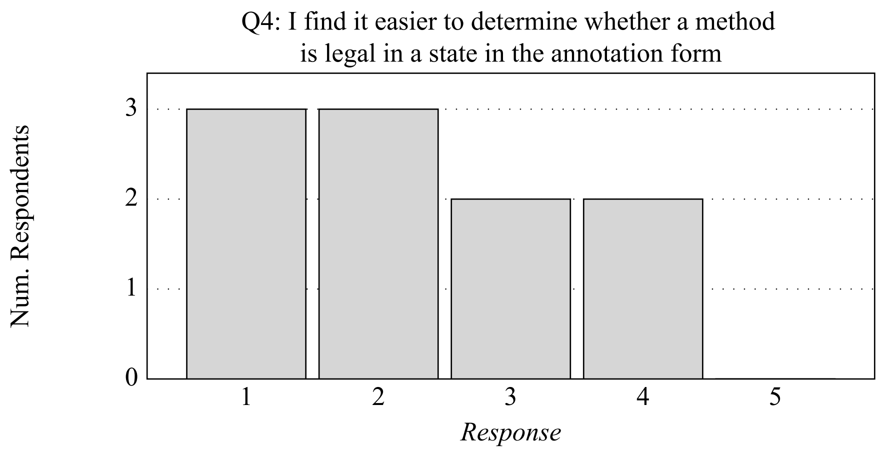
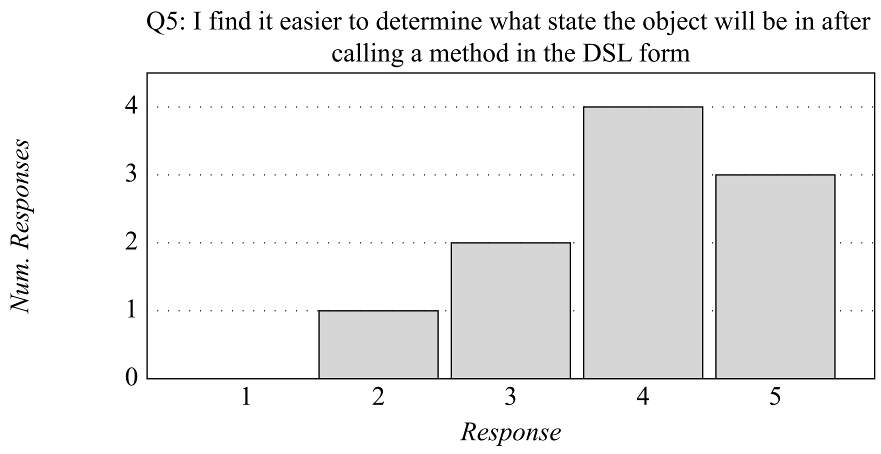


Figure 5.11: Participant responses to survey questions (Page 1 of 3)





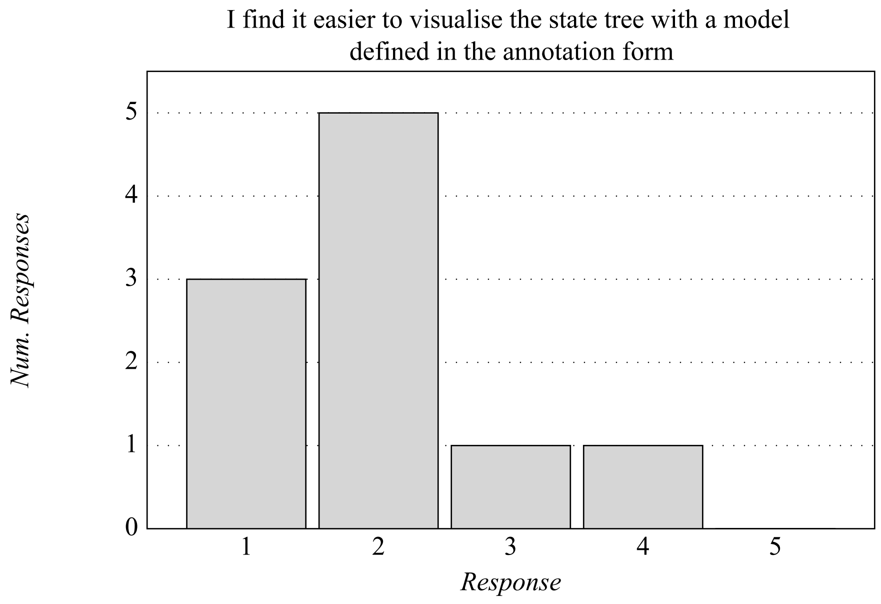
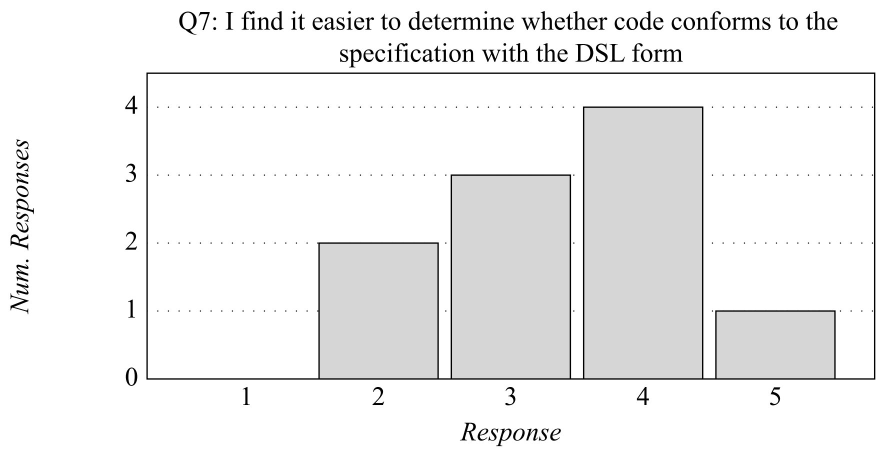
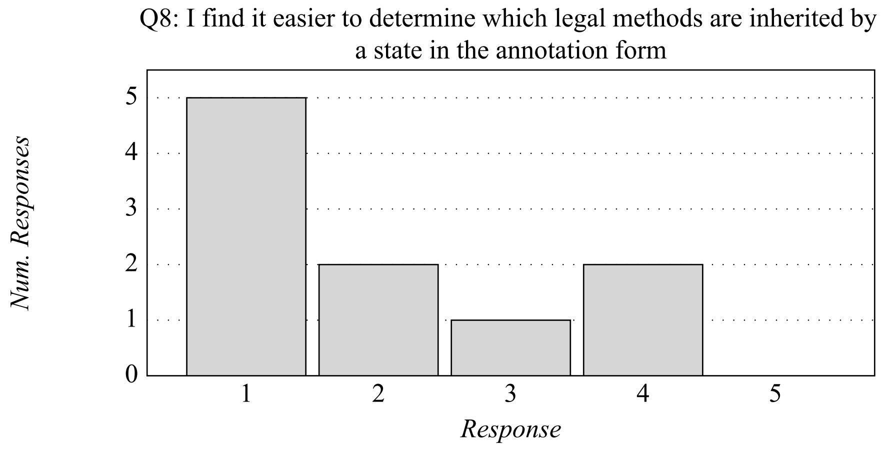


Figure 5.11: Participant responses to survey questions (Page 2 of 3)





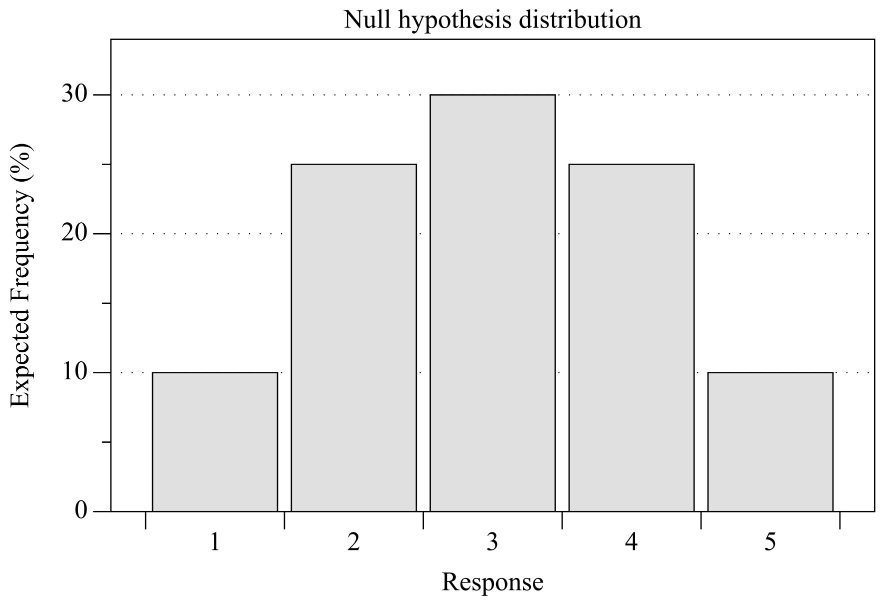


Figure 5.11: Participant responses to survey questions (Page 3 of 3)

5.6 Conclusion

The main results derived from this experiment were as follows:

- After 20 minutes of training on both model presentation types, participants answered 76% of questions correctly on average, with a standard deviation of 15%.
- A statistically significant, strong preference for the DSL presentation of Hanoi models over the annotation presentation was shown.
- No statistically significant correlation between presentation type and completion time or performance was found.

Based on the observations made during the experiment, the ability for a participant to read and correctly reason about Hanoi models with minimal training is confirmed, for models of the size and type present in this study. Such a result is promising, as it indicates that users are likely to be able to work with existing typestate models as may be found in a standard library. The experiment does not, however, say anything about how effectively users can write or modify models, another important aspect of designing such typestate models.

The experiment participants generally expressed frustration with the awkward annotation syntax and structure of the information presented as compared to the DSL model type. While the model type does not appear to affect performance directly, an irritating syntax is likely to have indirect consequences, such as influencing the willingness of a programmer to adopt a particular language for frequent use.

The main caveat to these results is the small sample size of 10 participants limits the strength of any claims which can be made. With additional data, we may discover that a statistically significant difference in performance between the presentation types would emerge, or that the preference for the DSL presentation would be weakened. The data gathered in this small experiment cannot provide any indication as to how likely any of these outcomes are.

The results are also quite specific to both the Java language and the model types used — one cannot make strong claims that these experimental results could be generalised to other typestate models, or even similar typestate models in other languages with sufficiently different semantics (e.g. a Hanoi typestate model for OCaml). Additionally, the experiment does not provide any specific guidance as to whether further optimal syntax variants exist — specific syntactic issues may only be found through a lengthier qualitative study performed with significantly more participants.

However, I believe that due to the similarity of Hanoi annotations to the Plural language, one may reasonably infer that a variant of the Hanoi DSL with support for the additional features of Plural is likely to be preferable to the existing Plural mechanism for defining typestate models. Providing support for Plural's fractional permission model and state dimensions presents a new, interesting syntax design challenge. There are many possibilities for doing this, which could be evaluated with similar experiments to that conducted here for Hanoi.

Regardless of the specificity of such experiments, conducting them to evaluate language design decisions adds scientific rigor to the process, substituting conjecture for real data. The study overall demonstrates the value of exploring different methods of encoding typestate rules and evaluating them with the target audience — assumptions about what will affect performance and what users will prefer can be evaluated with meaningful, reproduceable results. Such experiments are certainly not straightforward to design or execute but the observations made can be useful in challenging language design assumptions, and better serving the user.

Chapter 6

Typestate Inference in an Imperative First-Order Calculus

Often it is apparent from a code fragment what demands it makes of the entities it interacts with, and what *effect* it will have on those entities. Consider the pseudo-code in Figure 6.1a, which defines a function f. For such a function to be able to execute without failure at runtime, we may reasonably assume that the following properties must hold for parameters a and b given the semantics of a Java-like language:

- It must be possible to invoke methods m, p and q on parameter a. Similarly, it must be possible to invoke methods n and o on b.
- Method m must be invokable with a single integer parameter. All other methods must be invokable with no parameters.
- Method m must return a value which can be treated as a `Boolean`. Method q must return a value which can be treated as an `Integer`.

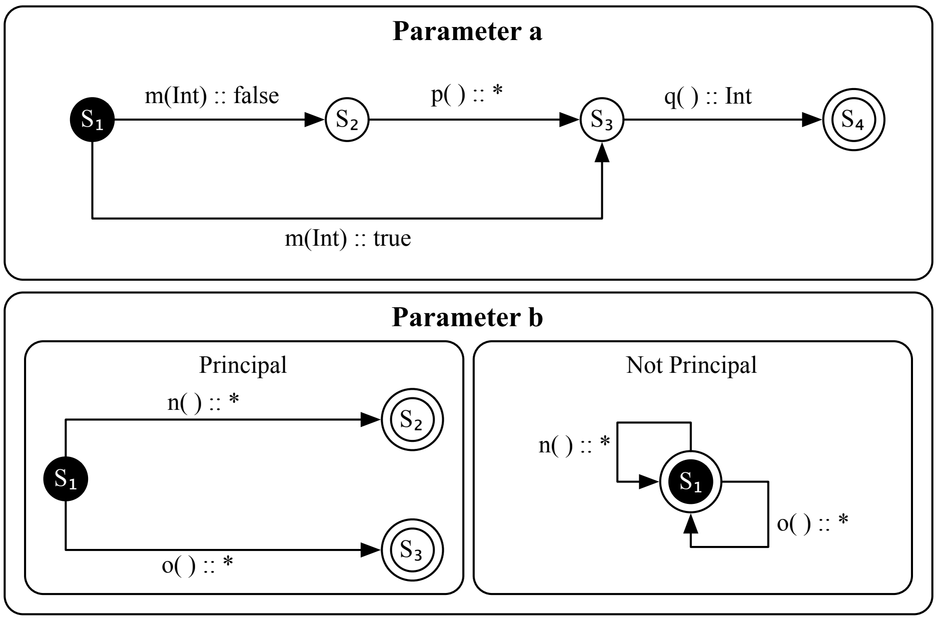
In a language with integrated typestate, it would also be desirable to infer the following:

- Method m must be invokable on a. If this returns false, then p must be invokable, after which q must be invokable. If m returns true, then q must be invokable.
- Either method n or method o must be invokable on b.

State machines which conform to these typestate constraints are shown in Figure 6.1b. Infinitely many state machines exist which would permit the usage of parameters a and b in function f, however we are interested in those which capture just the essential information.

```
def f(a, b) : Integer = {
  if( a.m(1) ) {
    b.n()
  } else {
    b.o()
    a.p()
  }

  return a.q()
}
```

(a) Example function with unspecified parameter effects



(b) Possible types for parameters a and b

```
type AType = {
  S1 { m(Integer) : Bool => S2 }
  S2 { p() : Top => S3 }
  S3 { q() : Int => S4 }
  S4 { }
}

type BType = {
  S1 {
    n : Top => S2
    o : Top => S2
  }
  S2 { }
}

def f(a : AType@S1 >> AType@S4, b : BType@S1 >> BType@S2) : Integer = ...
```

(c) All necessary type declarations and annotations for function f

Figure 6.1: Example of type inference for an unannotated function.

The state machine shown for parameter a allows the behaviour required of a in f and nothing more (it is *principal*), and does this with the minimum number of states (it is *minimal*).

The state machine on the left for b is principal but not minimal (states S_2 and S_3 could be collapsed into one state), while the state machine on the right is neither — it allows repeated calls to n and o, which represents much more than what is actually required.

Two very important assumptions have been made in the inference of these constraints: a and b are not references to the same object, and the references are unique. If it is possible that a and b are references to the same object, then rather different state machines must be constructed based upon the semantics of interacting with shared references. A reasonable choice for such semantics would be that only methods that do not trigger a state change could be invoked, and therefore we may infer that all methods invoked on a and b in f must be of this form — essentially, treat the object as though no typestate constraints exist for the methods used.

If we can guarantee that a and b are references to different objects, but that the references are potentially not unique, then a variety of other options for inference may be available — we may allow state changes for the observed methods, but insist that the object be in the same state at the end of the function as it was at the start. This however would not be valid in a language with support for concurrency: even temporary state change is unsafe in such a setting without explicit synchronization [12].

Regardless of these challenges, it is clear that if a typestate inference algorithm can be devised that would allow f to be typed without the addition of any annotations, this is likely to provide a significant usability and productivity gain for programmers over existing systems such as Plural, which require parameter effects to be specified manually. The explicit type definitions for a and b are shown in Figure 6.1c, and require an additional 13 lines of code that the programmer would have to manually specify if type inference were unavailable.

If f were part of some public API, documenting its requirements and effect as shown is good practice, and would be significantly easier with tool support that could be provided with a type inference algorithm. $AType$ and $BType$ could be generated and then modified as the programmer desired, rather constructing them from scratch, where the type checker only provides basic support in the form of correctness checking.

In this chapter, the semantics of a minimal imperative language with typestate-constrained values is described and its properties explored. A typestate inference algorithm for this language is also presented, though some aspects of it are left unproven and must be tackled in future work.

6.1 A minimal typestate interaction formalism

The language which is to be formalised and explored in this chapter focuses on *interactions* with typestate constrained objects, and avoids defining a full object-oriented calculus as might be found in Featherweight Java [75] or the work of Abadi and Cardelli [1]. Formalising the definition of typestate constrained objects with fields and method implementations involves checking that objects support their defined interface — this is a complex topic in its own right, requiring model checking, explicit state invariant annotations (as in Plural), or both. Model checking involves exploring the full state space of the object to ensure that no errors would arise from a method call sequence that is declared to be legal. Explicit invariants require that the type system check that fields match the stated invariants for whatever state they indicate the object should be in after invocation — this is easier to check and less computationally expensive than model checking, but requires the programmer to do significantly more work in return.

A first-order imperative language with simple object values is sufficient to explore the inference of typestate constraints of the form presented in the introduction to this chapter. As discussed in Chapter 3, a deterministic finite state machine is adequate to model the most common typestate constraint patterns. The state machine explicitly describes the states of the object, the methods which can be called in each state and the transition triggered by a method call in a specific state. Object fields do not exist in the language, but could be simulated by paired get / set method calls as is common with Java Beans, or in Abadi and Cardelli's formal treatment of objects [1].

For the sake of simplicity and clarity, the language will be based upon a simplified version of the typestate model in Hanoi:

- The state machine used to represent the capabilities of an object will be flat. As has already been demonstrated (see Section 3.5.1) a valid hierarchical state machine can be transformed into a flat state machine. This allows for a simpler formalism without loss of expressivity, at the cost of additional duplication in the state machine definitions.
- Conditional transitions based on return values will not be considered. While the additional syntax and semantic rules to support conditional transitions may be straightforward to add to the type system, it may significantly complicate the type inference process.
- Method parameters shall not be included in the formalism. Hanoi does not consider method parameters in the definition of method transitions, as there are very few use cases where the values of parameters are important to state transition decisions.

As a consequence, method overloading is not included in the formalism either. Overloading is essentially a trick that allows different methods to have the same name, but be otherwise unrelated (i.e. different parameter list lengths, different parameter types) — this is convenient for the programmer, but adds additional unnecessary complexity to the formalism.

An object value at any point in time may be described as the combination of a finite state transition system (which we will refer to as the *object protocol*) and a state from within that machine. We write this as $O@S$, which can be read as "protocol O in state S".

Formally, an object protocol O is defined as a triple (M, Σ, Δ) where M is the set of all method names available in the object, Σ is the set of state labels for the object and $\Delta \subseteq \Sigma \times M \times \mathbb{T} \times \Sigma$ is the state transition relation, where $\mathbb{T}$ is the set of all types in the language. If $(S, m, T, S') \in \Delta$ then method m is available in state S, and invoking it will return a value of type T and change the state of the object to S'. Transitions for a given state and method must be unique, to guarantee that the protocol is deterministic, such that:

$$(S, m, T, S') \in \Delta \implies (S, m, T', S'') \in \Delta \implies T = T' \wedge S' = S''$$

6.2 TS - an imperative calculus with typestate

TS is a small imperative language which models interactions with typestate-constrained objects, supports function literals without implicit capture, and employs a form of structural subtyping. The grammar of the language is shown in Figure 6.2, with operational semantics in Figure 6.4 and typing rules in Figure 6.15. The language uses a technique similar to Reynold's syntactic control of interference [131] to avoid the issue of alias control entirely - there is no means within the language to produce an alias to an object. This focuses attention on the fundamentals of typestate and typestate inference.

6.2.1 Notational conventions

Definition 6.2.1 (Sets). The notation $\overline{x_i}$ indicates a *set* of elements, where each element is assigned a subscript to distinguish it from the others. The subscript does not imply any order. If the elements need not be distinguished from each other, the subscript is omitted, as in $\overline{x}$. The size (or *cardinality*) of a set is denoted $|\overline{x}|$.

Sets are often used to represent mappings from keys to values, using the notation $\overline{x_i : y_i}$ or $\overline{x_i \mapsto y_i}$, where x_i is the key and y_i is the value for x_i. ▲

$$\begin{array}{llll}
t & := & v & \text{value} \\
 & | & \textbf{let } x = t_x \textbf{ in } t_b & \text{let bind} \\
 & | & t_a \,;\, t_b & \text{sequencing} \\
 & | & x(\overrightarrow{x_i}) & \text{function call} \\
 & | & x.m & \text{method call} \\
 & | & \textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f & \text{conditional} \\
 & | & \textbf{while } t_c \textbf{ do } t_b & \text{repetition} \\
 & & & \\
v & := & \texttt{unit} & \text{unit literal} \\
 & | & \texttt{true} & \text{boolean true} \\
 & | & \texttt{false} & \text{boolean false} \\
 & | & \left[\overline{S_i\{\overline{m_{ij} = (v_{ij}, S_{ij})}\}}\right]@S & \text{object literal} \\
 & | & \lambda(\overrightarrow{x_i : E_i}).T & \text{function literal} \\
 & & & \\
E & := & T \gg U & \text{flow effect} \\
 & | & T \ggg U & \text{update effect} \\
 & | & T & \text{sugar for } T \gg T \\
 & & & \\
T, U, V & := & \top & \text{top type} \\
 & | & \textbf{Unit} & \text{unit type} \\
 & | & \textbf{Bool} & \text{boolean type} \\
 & | & \left\{\overline{S_i\{\overline{m_{ij} : T_{ij} \Rightarrow S_{ij}}\}}\right\}@\overline{S} & \text{object type} \\
 & | & (\overrightarrow{E_i}) \rightarrow V & \text{function type}
\end{array}$$

Figure 6.2: Grammar of TS

Definition 6.2.2 (Vectors)**.** The notation $\overrightarrow{x_i}$ is an *ordered set* or *vector* of elements, where the subscript distinguishes the elements and also implies their order: x_0 precedes x_1, and so on. The length of a vector is denoted $\left|\overrightarrow{x}\right|$. Vectors are used in the language to describe parameter and argument lists, and ordered mappings using the notation $\overrightarrow{x_i : y_i}$. ▲

6.2.2 Values

The language has the following values:

- `unit`, the sole value of the $\mathbf{Unit}$ type, which is used for terms which have no meaningful result.
- `true` and `false`, of type $\mathbf{Bool}$, which are used for decisions in conditional evaluation. The language contains no operations for manipulating booleans, to keep the calculus small, however such operations would be straightforward to add.
- Function literals of form $\lambda(\overrightarrow{x_i : E_i}).t$, where t is the body of the function and $\overrightarrow{x_i : E_i}$ is a vector of parameters where each x_i must be unique. Each E_i is an *effect type* which declares what is required of a parameter and how it will be changed as part of applying the function. The formal definition and interpretation of effects, particularly in the presence of subtyping, is described in Section 6.3.4.

  Parameters are passed by reference. Function literals do not allow for implicit capture of variables into their scopes — every free variable of the body must be bound by a parameter. Defining and interpreting effects for implicitly captured references is not well understood in the current literature, and is a problem worthy of futher study. Requiring that all references be explicitly passed to the function allows for all effects to be explicitly declared.

  The explicitly declared effects on function literals exist only for the benefit of the type system. The primary objective of a type inference algorithm for TS would be to allow effect declarations to be omitted.
- Object literals of form $o@S$, where $o = \left[\overline{S_i\{\overline{m_{ij} = (v_{ij}, S_{ij})}\}}\right]$ is the object protocol, and S is the current state (where $S \in \overline{S_i}$). The object protocol contains a set of distinct state labels $\overline{S_i}$, within which is a set of methods $\overline{m_{ij}}$ that can be called in that state. The actual behaviour of methods is abstracted to just returning a single value of a fixed type, rather than evaluating some term with a handle to the object. Method evaluation in a more realistic language often produces a range of values; this could be simulated in TS by specifying a set of return values for a method and allowing the operational semantics to return one of these randomly. This does not otherwise affect the semantics

of the language, and could be useful for simulating "external choice" for conditional transitions.

The language has the following terms:

- Conditionals of form $\textbf{if}\ t_c\ \textbf{then}\ t_t\ \textbf{else}\ t_f$, which evaluates t_c first to a boolean result which decides which of t_t (the "true" branch) or t_f (the "false" branch) is evaluated.
- Function calls of form $x(\overrightarrow{x_i})$, which extract the function literal stored in x and apply it to the vector of parameters $\overrightarrow{x_i}$, which must match the declared parameter types. Each x_i must be a distinct variable, and is passed by reference, allowing the typestate of x_i to be changed as a result of the function application.
- Method invocations on objects of form $x.m$. A method call is only valid when the method is available in the current state of the object stored in x. After a valid invocation, the state of the object will have changed as specified by the object's protocol.
- A let-bind construct $\textbf{let}\ x = t_x\ \textbf{in}\ t_b$, which creates a new variable x within the scope of the body term t_b, which is a reference to the value derived from the term t_x. Rebinding a variable name is not allowed.
- Sequencing of form $t\,;\,t'$, which evaluates t, discards the result value and then evaluates t'. A sequence $t\,;\,t'$ is effectively equivalent to $\textbf{let}\ x = t\ \textbf{in}\ t'$, where x is a fresh variable name that does not occur in t'.
- While loops of form $\textbf{while}\ t_c\ \textbf{do}\ t_b$, which evaluate the condition t_c and if true, execute the body t_b. This is repeated until t_c evaluates to false.

While variables are references to a value, there is no construct which can duplicate a reference. This very simple mechanism ensures that references are unique, allowing us to focus our attention on the typestate properties of objects without the added complication of alias control annotations in the types. The restriction that each variable passed to a function be unique is for this reason, to prevent implicit duplication by passing the same variable more than once to the same function.

The example in Figure 6.3 shows a short TS program that emulates the usage of a file handle. The object `file` provides a simple contract where a file handle can only be `read` if the file is open. The object `console` is effectively stateless, allowing the method `print` to be called without restriction.

The function `f` interacts with the file, opening it and while `read` continues to return `true` it will invoke `print` on the console, before closing it.

```
let file =
[
  CLOSED { open = (true,OPEN) }
  OPEN {
    read = (true,OPEN) ;
    close = (unit,CLOSED)
  }
]@CLOSED
in
let console = [S { print = (unit,S) }]@S
in
let f = λ(fh,c).(
  fh.open;
  while fh.read do console.print;
  fh.close
)
in
f(file, console);
```

Figure 6.3: An example program in TS

As the effects on parameters `fh` and `c` are not specified for the function literal assigned to `f`, the term is not typeable as is. One would hope that the effect on these parameters could be inferred, and therefore be able to tell whether `file` and `console` would in fact be used safely by `f`. If type inference is not possible, then a type would need to be explicitly declared.

The definitions of `file` and `console` specify fixed return values for their methods; as such the evaluation of the while loop in *f* would be non-terminating as `read` will always return true. Consequently the language is very limited in terms of its ability to perform meaningful computation. This practical limitation has few consequences for the study of typestate inference in the language however, where the exact values returned by methods are irrelevant — the *structure* of a term is much more important, as the control flow of a term which interacts with an object is strongly tied to what is required of that object.

6.2.3 Term evaluation

As all variables are references, terms are evaluated in the context of a *store*:

Definition 6.2.3 (Stores)**.** A store is a set of unique variable names mapped to values: $\mu = \overline{x_i \mapsto v_i}$. A store location is created (or overwritten) upon entry to the body of a let-binding, written as $\mu[x \mapsto v]$, which is defined as follows:

$$\frac{\mu = \overline{x_i \mapsto v_i} \quad x \notin \overline{x_i}}{\mu[x \mapsto v] = \mu, x \mapsto v} \qquad \frac{\mu = \overline{x_i \mapsto v_i}, x \mapsto v'}{\mu[x \mapsto v] = \overline{x_i \mapsto v_i}, x \mapsto v}$$

▲

The mechanism for evaluating terms is described using the relation $t \mid \mu \longrightarrow t' \mid \mu'$, which can be read as "the term t and store μ can be reduced to the term t' and store μ' ". This relation defines the small step operational semantics of the language, and is defined inductively in Figure 6.4.

As there is no way for a reference to be duplicated, the mapping from variable names to store locations is injective and therefore we have no need for explicit "location values" as in the typical treatment of languages with references (such as in [118, Chapter 13]). There is no need for a separate store typing, as a well-typed store can be defined directly in relation to a context as in Figure 6.18. A well-typed store may contain mappings for variables which are not in the context as shown in rule ST_EXTRA. Such mappings are candidates for garbage collection, or an explicit deallocation instruction could be injected after the evaluation of the body of a let-binding.

There are three ways the evaluation of a term can become "stuck":

- An attempt is made to use a variable which does not exist in the store (R_FUN_CALL, R_METH_CALL).
- The value derived for the condition of an if-then-else or while loop expression is not a boolean (R_IF_TRUE, R_IF_FALSE).
- An attempt is made to call a method on a variable which is not an object, or where the method is not available in the current state (R_METH_CALL).

We wish to design a type system that will prevent the expression of stuck terms of the above kinds. First, the types of the language and their properties shall be defined, before describing the typing relation in Section 6.4.

6.3 Types

There are four kinds of type in TS, corresponding to the four kinds of values: Boolean, Unit, object types and function types. The boolean, unit and function types are referred to collectively as the *primitive types*. Function types of form $(\overrightarrow{E_i}) \rightarrow T$ specify the *effect* E_i that a function will have on each passed parameter; these include a *requirement* and *guarantee* component, such that a passed parameter must be a subtype of the requirement, and that after the evaluation of the function the parameter will be of some type determined by the guarantee. Effects are the main source of complexity in the TS language, and are described in more detail in Section 6.3.4.

$$\frac{t' \mid \mu \longrightarrow t'' \mid \mu}{\textbf{let } x = t' \textbf{ in } t \mid \mu \longrightarrow \textbf{let } x = t'' \textbf{ in } t \mid \mu'} \text{ R\_LET\_TERM}$$

$$\frac{}{\textbf{let } x = v \textbf{ in } t \mid \mu \longrightarrow t \mid \mu[x \mapsto v]} \text{ R\_LET\_VALUE}$$

$$\frac{t \mid \mu \longrightarrow t'' \mid \mu'}{t\,;\, t' \mid \mu \longrightarrow t''\,;\, t' \mid \mu'} \text{ R\_SEQ\_LEFT\_TERM}$$

$$\frac{}{v\,;\, t \mid \mu \longrightarrow t \mid \mu} \text{ R\_SEQ\_LEFT\_VALUE}$$

$$\frac{\mu(x) = \lambda(\overrightarrow{y_i : E_i}).t}{x(\overrightarrow{x_i}) \mid \mu \longrightarrow t\{\overline{{}^{x_i}/_{y_i}}\} \mid \mu} \text{ R\_FUN\_CALL}$$

$$\frac{O = [\ldots S\{\ldots m = (v, S') \ldots\} \ldots]}{x.m \mid \mu, x \mapsto O@S \longrightarrow v \mid \mu, x \mapsto O@S'} \text{ R\_METH\_CALL}$$

$$\frac{t_c \mid \mu \longrightarrow t'_c \mid \mu'}{\textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f \mid \mu \longrightarrow \textbf{if } t'_c \textbf{ then } t_t \textbf{ else } t_f \mid \mu'} \text{ R\_IF\_TERM}$$

$$\frac{}{\textbf{if } \texttt{true} \textbf{ then } t_t \textbf{ else } t_f \mid \mu \longrightarrow t_t \mid \mu} \text{ R\_IF\_TRUE}$$

$$\frac{}{\textbf{if } \texttt{false} \textbf{ then } t_t \textbf{ else } t_f \mid \mu \longrightarrow t_f \mid \mu} \text{ R\_IF\_FALSE}$$

$$\frac{}{\textbf{while } t_c \textbf{ do } t_b \mid \mu \longrightarrow \textbf{if } t_c \textbf{ then } (t_b\,;\, \textbf{while } t_c \textbf{ do } t_b) \textbf{ else } \texttt{unit} \mid \mu} \text{ R\_WHILE}$$

Figure 6.4: Operational semantics

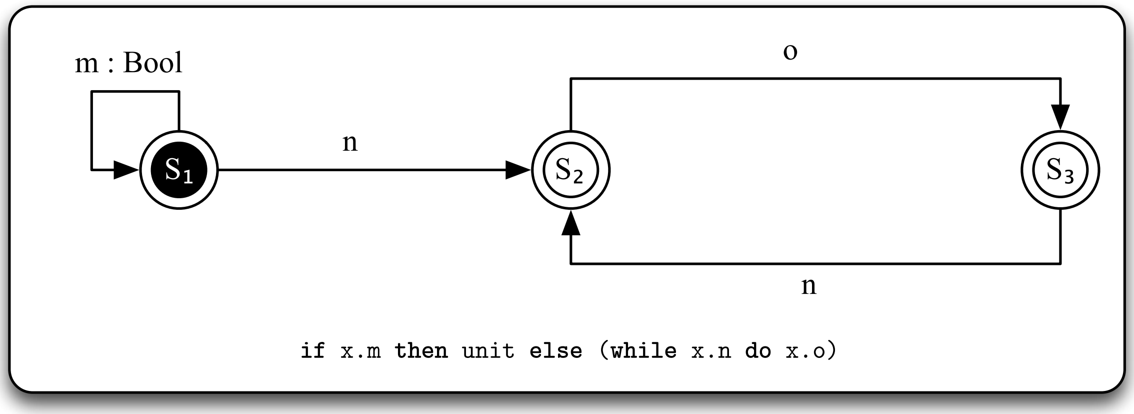


Figure 6.5: The variable x, starting in state S_1, can be in state S_1, S_2 or S_3 after the evaluation of the example term.

Object types in TS are the combination of an object protocol and a *set* of states, rather than an individual state. This allows for the convenient description of the statically indeterminate state of an object after the evaluation of conditionals and loops — depending upon which branch of a conditional is executed, or how many times the body of a loop is executed, an object may be in a variety of different states. We can represent this ambiguity with a set of all possible states the object may be in, which in the worst case is the full set of states defined in the object protocol.

Consider the term and object protocol in Figure 6.5. If x starts in state S_1, then depending upon the exact return values of m and n, the object will be in state S_1, S_2 or S_3 after evaluation. The object is of type $O@\{S_1\}$ before evaluation, and after evaluation it is of type $O@\{S_1, S_2, S_3\}$. In general, we may express the type of an object as $O@\overline{S}$ where $\overline{S}$ is some non-empty set of states, where the specific state labels are not relevant to the discussion at hand. The type $O@\{S\}$, where the exact state of the object is known, can be abbreviated as $O@S$.

Intuitively, the only methods that can be safely called on an object of type $O@\overline{S}$ are those which are available in *all* of the states in $\overline{S}$. Formally, $O@\overline{S}$ is equivalent to $\bigsqcup\{O@S' \mid S' \in \overline{S}\}$, where $T \sqcup T'$ is the *least upper bound* (or *join*) of T and T'. The join (and *meet*) operations for types are formally defined in Section 6.3.3.

6.3.1 Extra notational conventions for objects

Given an object type $O@\overline{S}$, to save referring to the state set Σ, method set M and transition relation Δ that formally define O the following notational conventions shall be used:

- $S \in O$ is the *state existence predicate* asserting S is a state of O. This is equivalent to $S \in \Sigma$.

- $m : T \Rightarrow S' \in O@S$ is the *method existence predicate* asserting the existence of method m with return type T in state S, which triggers a transition to state S' when called (where both $S \in O$ and $S' \in O$). This is equivalent to $(S, m, T, S') \in \Delta$.

- $m : T \Rightarrow \overline{S'} \in O@\overline{S}$ is the extension of the method existence predicate to state sets, meaning that for all $S \in \overline{S}$ there exists a state $S' \in \overline{S'}$ and type T' where $(S, m, T', S') \in \Delta$. T is defined as $\bigsqcup_{S \in \overline{S}} \{T' \mid \exists S'.\ (S, m, T', S') \in \Delta\}$ and $\overline{S'}$ is defined as $\{S' \mid \exists T'.\ (S, m, T', S') \in \Delta\}$.

  This may also be written as $m : T \Rightarrow U \in V$, which is to be interpreted as $U = O@\overline{S'}$ and $V = O@\overline{S}$ such that $m : T \Rightarrow \overline{S'} \in O@\overline{S}$, for some object protocol O and state sets $\overline{S}$ and $\overline{S'}$, where the specifics of the object protocol and state sets are either irrelevant or obvious from the context where this is used.

6.3.2 Subtyping

Subtyping in TS is intended to be implicit — it is not necessary for a programmer to declare that one type is a subtype of another; this can be decided automatically based upon the definition of the types. Such decisions are made when necessary by the type system, such as when applying a function. Subtyping is defined by the rules in Figure 6.6.

The subtyping relation in TS exhibits some of the characteristics of *structural subtyping* — the decision as to whether $A <: B$ is based upon the shape of the types, and the relation between their sub-components in the same "position". For instance, $() \rightarrow \mathbf{Unit} <: (E) \rightarrow \mathbf{Unit}$ is false as the types have a different shape: the number of parameters do not match. However $(E) \rightarrow W <: (E') \rightarrow W'$ may be true as the types have the same shape, but it must be the case that E is a *sub-effect* of E' and that $W <: W'$ (rule SUB_FN). The sub-effect relation is defined later in Section 6.3.4.

Object subtyping in TS is defined in a similar manner to Hanoi, using trace inclusion. A type $O_1@\overline{S_1}$ is a subtype of $O_2@\overline{S_2}$ if $Tr(O_2@\overline{S_2}) \subseteq Tr(O_1@\overline{S_1})$ (rule SUB_OBJ). Unlike the description of traces in Hanoi, an *interaction* is the combination of a method and its return type, rather than the returned value: (m, T). The set of traces for a type is inductively defined using rules TR_EMPTY and TR_PREFIX. The definition of TR_PREFIX ensures that covariance of the return types of methods is enforced. Let $O_1@\overline{S_1}$ and $O_2@\overline{S_2}$ both have a method m available, such that $m : T' \Rightarrow \overline{S_3} \in O_1@\overline{S_1}$ and $m : T \Rightarrow \overline{S_4} \in O_2@\overline{S_2}$. In order for $O_1@\overline{S_1} <: O_2@\overline{S_2}$ to hold, it must be the case that $T' <: T$, as otherwise (m, T) will not be an interaction available to $O_1@\overline{S_1}$.

TS includes a greatest type, $\mathbf{Top}$, which is a super-type of all other types. This type is written as `Top` in program text but the conventional symbol, $\top$, is often used in the discussion of the type system. TS does not include a least ($\bot$, or $\mathbf{Bot}$) type. The consequences of

$$\frac{}{T <: \top}\ \textsc{sub\_top} \qquad \frac{}{T <: T}\ \textsc{sub\_refl} \qquad \frac{T <: U \quad U <: V}{T <: V}\ \textsc{sub\_trans}$$

$$\frac{\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{E_i'}\right| \quad \forall i. E_i \leq E_i' \quad V <: V'}{(\overrightarrow{E_i}) \to V <: (\overrightarrow{E_i'}) \to V'}\ \textsc{sub\_fn} \qquad \frac{Tr(O'@\overline{S'}) \subseteq Tr(O@\overline{S})}{O@\overline{S} <: O'@\overline{S'}}\ \textsc{sub\_obj}$$

$$\frac{T <: U \quad \neg(U <: T)}{T \lll: U}\ \textsc{sub\_strict} \qquad \frac{T <: U \quad U <: T}{T \equiv U}\ \textsc{ty\_equiv}$$

$$\frac{}{\epsilon \in Tr(T)}\ \textsc{tr\_empty} \qquad \frac{m : T' \Rightarrow \overline{S'} \in O@\overline{S} \quad T' <: T \quad \delta \in Tr(O@\overline{S'})}{(m, T).\delta \in Tr(O@\overline{S})}\ \textsc{tr\_prefix}$$

Figure 6.6: Subtyping and equivalence rules. It should be noted that while the parameter effects on functions are covariant, the sub-effect relation $E \leq E'$ is contravariant on the effect input type as expected.

including $\bot$ were not considered in detail when designing the type system — it is an open question whether $\bot$ can be introduced to the type system while retaining soundness, and whether the type serves any useful purpose, such as simplifying definitions or proofs, or permitting the introduction of exceptions to the language in a more straightforward manner. A (hypothetical) value of type $\bot$ should be usable in any context, in particular it should be possible to pass it as a parameter to any function, or use it as a function to which other variables are supplied as arguments. Consequently, it would be necessary to define what the application of an effect to $\bot$ would be, and what effect $\bot$ has on its parameters. While it is speculated later (in Section 6.5.1) whether $\bot$ would be beneficial for type inference, no work has yet been done to fully assess its impact in the formalism.

Type equivalence

As object types are formalised as finite state transition systems, there are often multiple ways to represent the same object type. Object types $O_1@\overline{S_1}$ and $O_2@\overline{S_2}$ are considered *equivalent*, written $O_1@\overline{S_1} \equiv O_2@\overline{S_2}$, if they are subtypes of each other. Equivalence extends to all other types, such that $T \equiv U$ if $T <: U$ and $U <: T$ (defined as rule TY_EQUIV in Figure 6.6).

The subtyping relation is a pre-order, as reflexivity and transitivity are provided directly by the SUB_REFL and SUB_TRANS rules. The subtyping relation is also a partial order up to equivalance, which is proven in Lemma A.5.5.

6.3.3 Join and meet

It is often necessary to calculate the least upper bound or greatest lower bound of two or more types.

Definition 6.3.1 (Type bounds)**.** An upper bound U of two types T_1 and T_2 is a type such that $T_1 <: U$ and $T_2 <: U$. A lower bound L of T_1 and T_2 is a type such that $L <: T_1$ and $L <: T_2$.

As $\top$ is a supertype of all types, an upper bound exists for any arbitrary pair of types. However, a lower bound does not always exist for any arbitrary pair of types. ▲

Definition 6.3.2 (Join)**.** The least upper bound or *join* of two types, written $T_1 \sqcup T_2 = U$, is an upper bound such that for any other upper bound U', $U <: U'$. Formally:

$$T_1 \sqcup T_2 = U \implies \forall U'.\, T_1 <: U' \wedge T_2 <: U' \implies U <: U'$$

The join of a set of types $\overline{T_i}$ is written as $\bigsqcup \overline{T_i}$. ▲

Definition 6.3.3 (Meet)**.** The greatest lower bound or *meet* of two types, written $T_1 \sqcap T_2 = L$, is a lower bound such that for any other lower bound L', $L' <: L$. Formally:

$$T_1 \sqcap T_2 = L \implies \forall L'.\, L' <: T_1 \wedge L' <: T_2 \implies L' <: L$$

The meet of a set of types $\overline{T_i}$ is written $\bigsqcap \overline{T_i}$. ▲

The join and meet operators for types are defined in Figure 6.7 and Figure 6.8 respectively. The join of two types is always defined (see Lemma A.9.2), while the meet of two types is not always defined (e.g. $\mathbf{Bool} \sqcap \mathbf{Unit}$).

Deriving the join of two object types (rule TY_JOIN_OBJ) involves constructing the intersection of the state transition systems for each object type. Similarly, deriving the meet (rule TY_MEET_OBJ) involves construction the union of the state transition systems in a similar fashion. As a special case, if we wish to compute the join of two object types with the same protocol, i.e. $O@\overline{S} \sqcup O@\overline{S'}$, it is sufficient to compute the union of the state sets: $O@\overline{S} \sqcup O@\overline{S'} = O@\overline{S''}$ where $\overline{S''} = \overline{S} \cup \overline{S'}$.

The definition of the union and intersection of the state transition systems requires the ability to produce fresh state labels with respect to the state label sets of the input state machines. For this, we define the label composition function $L(S, S')$ and transposition function $L_t(S)$:

Definition 6.3.4 (State label composition)**.** The function $L(\Sigma_1, \Sigma_2, S_1, S_2)$ produces a new state label by combining two state labels from two state sets. It has the following properties:

- The function output is defined when the state labels exist in their respective state sets: $S_1 \in \Sigma_1 \implies S_2 \in \Sigma_2 \implies \exists S_3.\ L(\Sigma_1, \Sigma_2, S_1, S_2) = S_3$.
- Produced labels are fresh relative to the original state sets: $\forall S_1 \in \Sigma_1, S_2 \in \Sigma_2.\ L(\Sigma_1, S_1, \Sigma_2, S_2) \notin \Sigma_1 \cup \Sigma_2$.
- The function is injective with respect to the input state pair: $L(\Sigma_1, \Sigma_2, S_1, S_2) = L(\Sigma_1, \Sigma_2, S_3, S_4) \iff S_1 = S_3 \wedge S_2 = S_4$.

The exact definition of L is irrelevant as long as these properties hold; there are many possible definitions that could optimise for human readability or simplicity.

Where the identity of Σ_1 and Σ_2 are obvious from the context in which L is used (typically in relation to object types with the same subscript), $L(S_1, S_2) = L(\Sigma_1, \Sigma_2, S_1, S_2)$ is used as an abbreviation. ▲

Definition 6.3.5 (State label transposition)**.** The function $L_t(\Sigma_1, \Sigma_2, S)$ produces a new state label. It has the following properties:

- It is defined for all state labels in Σ_2: $S \in \Sigma_2 \implies \exists S'.L_t(\Sigma_1, \Sigma_2, S) = S'$.
- It is injective: $L_t(\Sigma_1, \Sigma_2, S) = L_t(\Sigma_1, \Sigma_2, S') \iff S = S'$.
- Labels are guaranteed to be distinct from those in Σ_1: $S \in \Sigma_2 \implies L_t(\Sigma_1, \Sigma_2, S) \notin \Sigma_1$.

Similar to the label composition function L, the exact definition of L_t is irrelevant as long as these properties hold.

Where the identity of Σ_1 and Σ_2 are obvious from the context in which L_t is used (typically in relation to object types with the same subscript), $L_t(S) = L_t(\Sigma_1, \Sigma_2, S)$ is used as an abbreviation. ▲

The meet of two object types may not be defined, due to the covariant treatment of return types on methods. Consider the two object types $\{S\ \{m : \mathbf{Unit} \Rightarrow S\}\}@S$ and $\{S\ \{m : \mathbf{Bool} \Rightarrow S\}\}@S$. The meet of these two types would require that m return a value that is a subtype of both $\mathbf{Unit}$ or $\mathbf{Bool}$, but no such type exists. Therefore, the meet of these two object types is undefined. The join of two object types is always an object type, even if it is simply equivalent to the empty object, $\{S\{\}\}@S$, due to the two joined object types having no methods in common in their current states.

The join (or meet) of two function types of the same arity is defined as the pair-wise join (or meet) of each effect and the join (or meet) of the return types (rules TY_JOIN_FUN and

$$\frac{}{T \sqcup T = T}\ \textsc{ty\_join\_refl}$$

$$\frac{\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right| \quad \forall i. E_i \sqcup F_i = G_i \quad T \sqcup U = V}{(\overrightarrow{E_i}) \to T \sqcup (\overrightarrow{F_i}) \to U = (\overrightarrow{G_i}) \to V}\ \textsc{ty\_join\_fun}$$

$$\frac{\begin{array}{c} O_1 = (M_1, \Sigma_1, \Delta_1) \quad O_2 = (M_2, \Sigma_2, \Delta_2) \\ M_3 = M_1 \cup M_2 \quad \Delta_3 = \Delta_1 \sqcup \Delta_2 \\ \Sigma_3 = \{L(S_1, S_2) \mid S_1 \in \Sigma_1,\ S_2 \in \Sigma_2\} \quad S_3 = L(S_1, S_2) \end{array}}{O_1 @ S_1 \sqcup O_2 @ S_2 = O_3 @ S_3}\ \textsc{ty\_join\_obj}$$

$$\frac{}{\Delta_1 \sqcup \Delta_2 = \left\{ (L(S_1, S_2), m_1, T_1 \sqcup T_2, L(S_1', S_2')) \mid \begin{array}{c} (S_1, m, T_1, S_1') \in \Delta_1, \\ (S_2, m, T_2, S_2') \in \Delta_2 \end{array} \right\}}\ \textsc{trans\_join}$$

$$\begin{array}{lcllcl} \mathit{kind}(\top) & = & 0 & \mathit{kind}(\mathbf{Unit}) & = & 1 \\ \mathit{kind}(\mathbf{Bool}) & = & 2 & \mathit{kind}((\overrightarrow{E_i}) \to U) & = & 3 \\ \mathit{kind}(O @ \overline{S}) & = & 4 & & & \end{array}$$

$$\frac{\mathit{kind}(T) \neq \mathit{kind}(U)}{T \sqcup U = \top}\ \textsc{ty\_join\_kind\_diff}$$

$$\frac{\left|\overrightarrow{E_i}\right| \neq \left|\overrightarrow{F_j}\right|}{(\overrightarrow{E_i}) \to T \sqcup (\overrightarrow{F_j}) \to U = \top}\ \textsc{ty\_join\_fun\_diff\_arity}$$

$$\frac{\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right| \quad \exists n.\ E_n \sqcup F_n \text{ is undefined}}{(\overrightarrow{E_i}) \to T \sqcup (\overrightarrow{F_i}) \to U = \top}\ \textsc{ty\_join\_fun\_diff\_eff}$$

Figure 6.7: Definition of type join. The label composition function $L(S_1, S_2)$ is described in Definition 6.3.4.

$$\frac{}{T \sqcap T = T}\ \text{TY\_MEET\_REFL} \qquad \frac{}{T \sqcap \top = \top \sqcap T = T}\ \text{TY\_MEET\_TOP}$$

$$\frac{|\overrightarrow{E_i}| = |\overrightarrow{F_i}| \quad V = T \sqcap U \quad G_i = E_i \sqcap F_i}{(\overrightarrow{E_i}) \to T \sqcap (\overrightarrow{F_i}) \to U = (\overrightarrow{G_i}) \to V}\ \text{TY\_MEET\_FUN}$$

$$\frac{\begin{array}{c} O_1 = (M_1, \Sigma_1, \Delta_1) \quad O_2 = (M_2, \Sigma_2, \Delta_2) \quad O_3 = (M_3, \Sigma_3, \Delta_3) \\ M_3 = M_1 \cup M_2 \quad \Sigma_2' = \{L_t(S) \mid S \in \Sigma_2\} \\ \Sigma_3 = \Sigma_1 \cup \Sigma_2' \cup R(\Delta_1, \Delta_2, S_1, S_2) \quad S_3 = L(S_1, S_2) \\ \Delta_3 = \mathit{left}(\Delta_1) \cup \mathit{right}(\Delta_2) \cup \mathit{mid}(\Delta_1, \Delta_2, S_1, S_2) \end{array}}{O_1 @ S_1 \sqcap O_2 @ S_2 = O_3 @ S_3}\ \text{TY\_MEET\_OBJ}$$

$$\frac{\begin{array}{c} (S_1, m, T_1, S_1') \in \Delta_1 \quad (S_1, m, T_2, S_2') \in \Delta_2 \\ S = L(S_1, L_t(S_2)) \quad S \in R(\Delta_1, \Delta_2, S_a, S_b) \\ S' = L(S_1', L_t(S_2')) \quad T_1 \sqcap T_2 = T \end{array}}{(S, m, T, S') \in \mathit{mid}(\Delta_1, \Delta_2, S_a, S_b)}\ \text{TRANS\_MEET\_MID}$$

$$\frac{\begin{array}{c} (S_1, S_2) \in R(\Delta_1, \Delta_2, S_a, S_b) \\ (S_1, m, T, S_1') \in \Delta_1 \\ \nexists U, S_2' \text{ such that } (S_2, m, U, S_2') \in \Delta_2 \\ S = L(S_1, L_t(S_2)) \end{array}}{(S, m, T, S_1') \in \mathit{mid}(\Delta_1, \Delta_2, S_a, S_b)}\ \text{TRANS\_MEET\_LEFT}$$

$$\frac{\begin{array}{c} (S_1, S_2) \in R(\Delta_1, \Delta_2, S_a, S_b) \\ (S_2, m, T, S_2') \in \Delta_2 \\ \nexists U, S_1' \text{ such that } (S_1, m, U, S_1') \in \Delta_1 \\ S = L(S_1, L_t(S_2)) \end{array}}{(S, m, T, S_2') \in \mathit{mid}(\Delta_1, \Delta_2, S_a, S_b)}\ \text{TRANS\_MEET\_RIGHT}$$

$$\frac{(S_1, m, T, S_1') \in \Delta_1}{(S_1, m, T, S_1') \in \mathit{left}(\Delta_1)}\ \text{TRANS\_MEET\_COPY\_L}$$

$$\frac{(S_2, m, T, S_2') \in \Delta_2 \quad S = L_t(S_2) \quad S' = L_t(S_2')}{(S, m, T, S') \in \mathit{right}(\Delta_2)}\ \text{TRANS\_MEET\_COPY\_R}$$

$$\frac{}{L(S_1, L_t(S_2)) \in R(\Delta_1, \Delta_2, S_1, S_2)}\ \text{REACH\_REFL}$$

$$\frac{\begin{array}{cc} (S_a, m, T_a, S_a') \in \Delta_1 & (S_b, m, T_b, S_b') \in \Delta_2 \\ L(S_a, L_t(S_b)) \in R(\Delta_1, \Delta_2, S_1, S_2) & S \in R(\Delta_1, \Delta_2, S_a', S_b') \end{array}}{S \in R(\Delta_1, \Delta_2, S_1, S_2)}\ \text{REACH\_ADJ}$$

Figure 6.8: Definition of type meet. The label composition function $L(S_1, S_2)$ is described in Definition 6.3.4, and the label transposition function $L_t(S_1)$ are described in Definition 6.3.5.

TY_MEET_FUN, respectively). Join and meet for effect types shall be defined later, once the full semantics of effects have been described, though it should be noted at this point that neither is always defined. Consequently, $(E) \to T \sqcup (E') \to T = \top$ when $E \sqcup E'$ is not defined. In the case of the meet of two function types, if any effect type meet is undefined then the meet of the two functions is undefined.

6.3.4 Effect types

An effect type describes how the type of a variable may change in response to evaluating a term, and is an important part of abstracting the behaviour of a function to allow for modular type checking.

For the TS language, we explore the semantics of two different kinds of effect: *update* effects ($T \ggg U$) and *flow* effects ($T \gg U$). In both cases, the type T is referred to as the *input type* or *requirement*, and U as the *output type* or *guarantee*. For notational convenience, the functions $in(E)$ and $out(E)$ shall provide the requirement and guarantee of an effect E respectively.

Given a function $f = \lambda(x : E).t$ and a variable $y : T$ such that we wish to evaluate $f(y)$, we must define what the relationship between T and E should be, such that we may decide what the type of y should be after applying the function.

The semantics of effects can be defined in terms of the following functions and relations:

- $valid(E)$ — a predicate that asserts the requirement and guarantee are compatible, such that the effect is meaningful.
- $E \leq E'$ — a sub-effect relationship. This is necessary to define function subtyping, such that $(x : E) \to T <: (x : E') \to T$ when $E \leq E'$. This makes effects covariant to function subtyping, but as shall be described for both update and flow effects, the effect requirement is contravariant to function subtyping as one might expect. Where $E \leq E'$, we say that E' is a *wider* effect than E, or conversely that E is a *narrower* effect that E'.
- $remap(T, E)$ — a function which defines the type transformation that occurs when applying effect E to type T. This is typically used to describe the type of a passed parameter after applying a function, i.e. the type of parameter y after evaluating $f(y)$.

The sub-effect relationship and $remap$ must be related such that the following property holds:

Definition 6.3.6 (Effect output covariance)**.** Let $valid(E)$, $valid(E')$, and $E \leq E'$. If $remap(T, E')$ is defined, then $remap(T, E)$ is also defined such that $remap(T, E) <: remap(T, E')$. ▲

This property is necessary to ensure that a function f' which is a subtype of f can be safely substituted for f: anything that can be done with y after evaluating $f(y)$ must also be possible after evaluating $f'(y)$. This is proven in Lemma A.8.7.

The definitions of valid, $E \leq E'$ and remap are shown in Figure 6.9, with explanations of the differences between flow and update effects below.

Update effects

For an update effect $T \ggg U$, the requirement T and guarantee U need not be related: $\mathit{valid}(T \ggg U)$ is vacuously true for any T and U (rule VALID_UP_EFF). The sub-effect relationship is structural: $T \ggg U \leq V \ggg W$ if $V <: T$ and $U <: W$. Finally, for any $T' <: T$, the application of an update effect is defined such that $\mathit{remap}(T', T \ggg U) = U$.

Update effects match the semantics used for effects in other typestate systems such as Plural and Plaid. Update effects are simpler but provide less information on how a value is likely to be used during the evaluation of a term — they simply state what the term requires, and what behaviour is guaranteed to be safe after the evaluation of the term.

The definition of $\mathit{remap}(T', T \ggg U)$ for update effects is insensitive to the additional type information that T' may contain over T. As shall be demonstrated later, fewer terms are typeable when update effects are used as compared to flow effects for this reason.

Flow effects

Flow effects are written as $T \overset{\overline{\delta}}{\gg} U$, and represent a set of possible *interactions* with a value, represented by the set $\overline{\delta}$, which is a non-empty subset of the full set of permissible interaction traces for T (represented by $\mathit{Tr}(T)$).

The type T represents a requirement and U represents a guarantee as for update effects, however U is defined directly from the values of T and $\overline{\delta}$. Given a value v of type T and an interaction trace $\delta \in \overline{\delta}$, the function $\mathit{trmap}(T, t)$ (defined in Figure 6.10) specifies the type of v after invoking the sequence of methods that the interaction trace represents. The type U, therefore, is the join of all such resultant types: $U = \bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\}$.

For any valid flow effect $T \overset{\overline{\delta}}{\gg} U$, it is required that $\overline{\delta} \subseteq \mathit{Tr}(T)$. Flow effects with $\mathit{Tr}(T) \subset \overline{\delta}$ are invalid as any trace $\delta \notin \mathit{Tr}(T)$ would contain at least one method call which is not permitted at the point of the call. This is encapsulated in rule VALID_FL_EFF in Figure 6.9.

For notational convenience, $\overline{\delta}$ may be omitted such that $T \gg U = T \overset{\overline{\delta}}{\gg} U$ where $\overline{\delta}$ represents the largest set of traces that would result in a transformation from T to U. The rules for defining this largest set, $\mathit{Tr}(T \gg U)$, are shown in Figure 6.9, which is proven to be a subset

$$\frac{}{\mathit{valid}(T \ggg U)}\ \text{VALID\_UP\_EFF} \qquad \frac{V <: T \quad U <: W}{T \ggg U \leq V \ggg W}\ \text{SUB\_UP\_EFF}$$

$$\frac{\varnothing \subset \overline{\delta} \subseteq \mathit{Tr}(T) \quad U = \{\mathit{trmap}(T,\delta) \mid \delta \in \overline{\delta}\}}{\mathit{valid}(T \overset{\overline{\delta}}{\gg} U)}\ \text{VALID\_FL\_EFF}$$

$$\frac{\mathit{valid}(T \overset{\overline{\delta_1}}{\gg} U) \quad \mathit{valid}(V \overset{\overline{\delta_2}}{\gg} W) \quad V <: T \quad \overline{\delta_1} \subseteq \overline{\delta_2}}{T \gg U \leq V \gg W}\ \text{SUB\_FL\_EFF}$$

$$\frac{E_1 \leq E_2 \quad E_2 \leq E_3}{E_1 \leq E_3}\ \text{SUB\_EFF\_TRANS} \qquad \frac{E_1 \leq E_2 \quad E_2 \leq E_3}{E_1 \equiv E_2}\ \text{EFF\_EQUIV}$$

$$\frac{T <: U}{\mathit{remap}(T, U \ggg V) = V}\ \text{REMAP\_UP\_DEF}$$

$$\frac{T <: U \quad \mathit{valid}(U \overset{\overline{\delta}}{\gg} V)}{\mathit{remap}(T, U \overset{\overline{\delta}}{\gg} V) = \bigsqcup\{\mathit{trmap}(T,\delta) \mid \delta \in \overline{\delta}\}}\ \text{REMAP\_FL\_DEF}$$

$$\frac{}{\epsilon \in \mathit{Tr}(T \gg T)}\ \text{TR\_EFF\_EMPTY}$$

$$\frac{m : T' \Rightarrow \overline{S''} \in O@\overline{S} \quad \delta \in \mathit{Tr}(O@\overline{S''} \gg O@\overline{S'}) \quad T' <: T}{(m,T).\delta \in \mathit{Tr}(O@\overline{S} \gg O@\overline{S'})}\ \text{TR\_EFF\_PREFIX}$$

$$\frac{\delta \in \mathit{Tr}(T \gg U) \quad \mathit{valid}(T \gg V)}{\delta \in \mathit{Tr}(T \gg U \sqcup V)}\ \text{TR\_EFF\_BRANCH}$$

Figure 6.9: Definition of flow and update effect properties

of $\mathit{Tr}(T)$ in Lemma A.7.1. We can demonstrate that for any valid flow effect $T \overset{\overline{\delta}}{\gg} U$ that $\overline{\delta} \subseteq \mathit{Tr}(T \gg U)$ (Lemma A.7.3). For primitive types T, $\mathit{Tr}(T) = \{\epsilon\}$ and therefore the only valid flow effect is $T \overset{\overline{\delta}}{\gg} T$ with $\overline{\delta} = \{\epsilon\}$ (Lemma A.4.7).

The sub-effect relationship for flow effects is defined in terms of effect trace inclusion and a contra-variant subtyping relation between the input types (rule SUB_FL_EFF). As such, if $T \gg U \leq V \gg W$, then $T \gg U$ is more specific about what interactions may occur.

Finally, $\mathit{remap}(T', T \overset{\overline{\delta}}{\gg} U)$ can be defined simply as the application of all traces in $\overline{\delta}$ to the type T' (rule REMAP_FL_DEF in Figure 6.9). This is more sensitive to the extra information provided by T' than for update effects: we are guaranteed that $\mathit{remap}(T', T \overset{\overline{\delta}}{\gg} U) <: U$ (Lemma A.6.1), and preserves all information on the object protocol of T' in the case of object types.

$$\frac{}{T = trmap(T, \epsilon)}\ \text{TR\_MAP\_EMPTY}$$

$$\frac{m : U' \Rightarrow \overline{S'} \in O@\overline{S} \quad U' <: U \quad O@\overline{S''} = trmap(O@\overline{S'}, \delta)}{O@\overline{S''} = trmap(O@\overline{S}, (m, U).\delta)}\ \text{TR\_MAP\_PREFIX}$$

$$\frac{}{\epsilon + \delta = \delta}\ \text{TR\_PLUS\_EMPTY} \qquad \frac{\delta + \delta' = \delta''}{(m, V).\delta + \delta' = (m, V).\delta''}\ \text{TR\_PLUS\_PREFIX}$$

$$\frac{}{len(\epsilon) = 0}\ \text{LEN\_EMPTY} \qquad \frac{len(\delta') = n}{len((m, V).\delta') = n + 1}\ \text{LEN\_PREFIX}$$

Figure 6.10: Definition of trace operations

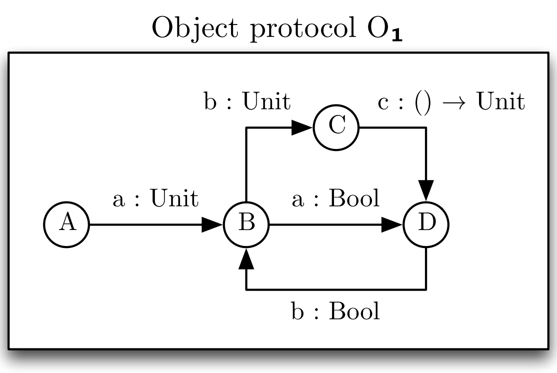


Figure 6.11: A simple object protocol

As an example, consider the object protocol in Figure 6.11, which we shall refer to as O_1, and let $O_2 = \{S_1\{a : \top \Rightarrow S_2\} S_2\{\}\}$. If we have a function f of type $(O_2@S_1 \ggg O_2@S_2) \rightarrow \top$ and a function f' of type $(O_2@S_1 \gg O_2@S_2) \rightarrow \top$, intuitively it should be safe to pass a value x with type $O_1@A$ as a parameter to either function.

By definition, $\mathit{remap}(O_1@A, O_2@S_1 \ggg O_2@S_2) = O_2@S_2$. This would indicate that evaluating $f(x)\,;\,x.b$ is potentially unsafe. In contrast, $\mathit{remap}(O_1@A, O_2@S_1 \gg O_2@S_2) = O_1@B$, meaning that we may reasonably expect to be able to type $f'(x)\,;\,x.b$.

The remap function over flow effects has a number of useful properties, which fit the intuition we may desire for the application of effects:

- $\mathit{remap}(T, T \gg U) = U$ (proven in Lemma A.7.5).
- $\mathit{remap}(T, U \gg V) <: V$ (proven in Lemma A.6.1).
- If $T' <: T$, then $\mathit{remap}(T', U \gg V) <: \mathit{remap}(T, U \gg V)$ (proven in Lemma A.6.2).

These properties demonstrate that in many ways flow effects are more precise than update effects, and generally offer stronger guarantees for the type of a value after function application. However, as the sub-effect relationship is not structural, $T \gg U \leq V \gg W$ cannot be decided by comparing T and V or U and W in isolation. Consequently, function subtyping is no longer structural, which introduces complications to the type inference strategy, as shall be discussed later.

Finally, a consequence of the definition the sub-effect relation for flow effects is that $\top \gg \top$ can be used to describe the effect on a variable which is not used in a term, as
$\mathit{remap}(T, \top \gg \top) = T$ for all T (proven in Lemma A.6.3). This is particularly useful for unused parameters to functions, where no type information is lost in applying the function to variables which will not be used in this manner. There is no equivalent for update effects; there is no meaningful guarantee we can use which will preserve type information in this manner without parametric polymorphism (i.e. the ability to define an update effect $\forall T.\, T \ggg T$).

6.3.5 Strong and weak update

In the TS language, once a variable has been bound to a particular value by either a let-bind or as a parameter to a function, it cannot be changed. Objects values can change state in response to method calls, but a reference always points to the same object.

As such, there is no facility in the language to overwrite or *update* the contents of a reference; it would however be relatively easy to add support for this.

```
\\ o = [ S { m = (true, S2) }
\\       S2 { n = (unit,S) }
\\       S3 { p = (unit,S3) } ]
\\
\\ O1 = { S { m : Bool => S2 }
\\        S2 { n : Unit => S3 }
\\        S3 { p : Unit => S3 } }
\\
\\ p = [ A { m = (false, B) }
\\       B { n = (unit, B) ; p = (unit, B) } ]
\\
\\ O2 = { A { m : Bool => B }
\\         B { n : Unit => B ; p : Unit => B } }

let f = λ(x : O1@S >>> O1@S3).(
  x.m ;
  x := o@S3
) in
let y = p@A in f(y)
```

Listing 6.1: Example of strong update

Weak update of a reference means replacing the contents of a reference with another value of the same type, while *strong update* means replacing the contents of a reference with another of an arbitrary type. The latter is disallowed in most languages as it may be dangerous if the reference is aliased. However, if the reference is known to be unique then the operation should be safe.

Supporting either weak or strong update in the TS language has consequences for the effects that can be ascribed to terms in the language. As an example, consider the code in Listing 6.1 which uses a hypothetical *strong update* operator $:=$ to change the value of reference x.

The function f is declared with an update effect for the parameter x. As can be observed, $O_2@A <: O_1@S$ and therefore the application of f using y should be safe. After the application, the type of y will be $\mathit{remap}(O_2@S, O_1@S \ggg O_1@S_3) = O_1@S_3$.

One may question whether the flow effect $O_1@S \gg O_1@S_3$ could be used for the parameter x. In this case, $\mathit{remap}(O_2@A, O_1@S \gg O@S_3) = O_2@B$ which is problematic as the body of f replaces x with a new value $o@S_3 : O@S_3$. Due to the pass by reference semantics of TS, this new value would be visible through the variable y after the call, where we may incorrectly assume it would be safe to invoke method n after evaluating $f(y)$.

Flow effects may only be used where we can be guaranteed that the *identity* of the value in a reference does not change, such that all operations (in particular, method calls) occur on the same instance. As such, flow effects may not be used to describe the effect on variables which are updated, whether the update is strong or weak. If the reference is not overwritten, then either a flow effect or an update effect may be used, though flow effects are preferred due to the additional precision they offer.

$$\frac{T = T_1 \sqcap T_2 \quad U = U_1 \sqcup U_2}{T_1 \ggg U_1 \sqcup T_2 \ggg U_2 = T \ggg U} \text{ EFF\_JOIN\_UP}$$

$$\frac{T = T_1 \sqcup T_2 \quad U = U_1 \sqcap U_2}{T_1 \ggg U_1 \sqcap T_2 \ggg U_2 = T \ggg U} \text{ EFF\_MEET\_UP}$$

$$\frac{T = T_1 \sqcap T_2 \quad \overline{\delta} = \overline{\delta_1} \cup \overline{\delta_2} \quad U = \{trmap(T, \delta) \mid \delta \in \overline{\delta}\}}{T_1 \overset{\overline{\delta_1}}{\gg} U_1 \sqcup T_2 \overset{\overline{\delta_2}}{\gg} U_2 = T \overset{\overline{\delta}}{\gg} U} \text{ EFF\_JOIN\_FL}$$

$$\frac{T = T_1 \sqcup T_2 \quad \overline{\delta} = \overline{\delta_1} \cap \overline{\delta_2} \quad \overline{\delta} \neq \varnothing \quad U = \bigsqcup\{trmap(T, \delta) \mid \delta \in \overline{\delta}\}}{T_1 \overset{\overline{\delta_1}}{\gg} U_1 \sqcap T_2 \overset{\overline{\delta_2}}{\gg} U_2 = T \overset{\overline{\delta}}{\gg} U} \text{ EFF\_MEET\_FL}$$

Figure 6.12: Definition of join and meet for effect types

This intuition, that flow effects are more precise than update effects, could be formalised as part of the sub-effect relationship: if $valid(T \gg U)$, then it could be said that $T \gg U \leq T \ggg U$. This would allow flow effects to be treated as update effects when necessary, allowing subsumption for function types such that $(T \gg U) \to V <: (T \ggg U) \to V$.

Despite the apparent possibility that strong and weak update could be added to the TS language and for flow and update effects to co-exist within the type system, this possibility was not considered further due to time constraints and has been left for future work.

6.3.6 Meet and join for effects

The join and meet of effect types is formally defined in Figure 6.12, and their basic properties proven in Section A.8.

As the sub-effect relationship for update effects is structural, the join and meet of two update effects is straightforward to define (rules EFF_JOIN_UP and EFF_MEET_UP respectively). The join of two update effects being defined is contingent on the meet of the requirement (input) types being defined, while the meet is dependent on the meet of the guarantee (output) types being defined. Therefore, both join and meet on update effects is partial.

The join of two flow effects $T_1 \overset{\overline{\delta_1}}{\gg} U_1 \sqcup T_2 \overset{\overline{\delta_2}}{\gg} U_2 = T \overset{\overline{\delta}}{\gg} U$, defined in rule EFF_JOIN_FL, is such that $T = T_1 \sqcap T_2$ and $\overline{\delta} = \overline{\delta_1} \cup \overline{\delta_2}$. The meet of two flow effects is similarly structured such that $T_1 \overset{\overline{\delta_1}}{\gg} U_1 \sqcap T_2 \overset{\overline{\delta_2}}{\gg} U_2 = T \overset{\overline{\delta}}{\gg} U$ where $T = T_1 \sqcup T_2$ and $\overline{\delta} = \overline{\delta_1} \cap \overline{\delta_2}$, where this intersection must not be empty.

Where the effects concern object types, such as in $O_1@\overline{S_1} \gg O_1@\overline{S_1'} \sqcup O_2@\overline{S_2} \gg O_2@\overline{S_2'}$, we may observe that each effect represents a finite state machine with defined entry and exit states. The join is similar to the standard intersection operation on finite state machines, while the meet is similar to the union.

6.3.7 Effect combinators

Effects formalise the possible state changes that may occur on a variable in response to evaluating a term. A term such as $t_a ; t_b$ composes two sub-terms which each have a separate effect on the variables they use. It is therefore natural to think of combining effects in ways that match the structure and evaluation order of such terms. Combinators for update effects are not very interesting, as the effects do not describe how a term is used. As such, we shall focus on combinators for flow effects.

Three basic combinators are required for effects in the TS language, which correspond to the basic combinators for regular languages: *concatenation*, *union* and *Kleene star*. Examples of their application to flow effects are shown in Figure 6.13, and each is discussed in more detail below.

Effect concatenation

$E \cdot E'$ is referred to as the *concatenation* of the effects E and E'. Effect concatenation is defined to be left-associative.

The effect $E = E_1 \cdot E_2$, if defined, represents a value which can be used in a way that is compatible with the effect E_1, *then* be used in a way that is compatible with the effect E_2. Therefore, it must be the case that $\mathit{in}(E) <: \mathit{in}(E_1)$ and $\mathit{remap}(\mathit{in}(E), E_1) <: \mathit{in}(E_2)$.

Concatenation for update effects is therefore defined such that $T \ggg U \cdot V \ggg W = T \ggg W$ when $U <: V$, otherwise the concatenation is not defined. For flow effects over primitive types, $T \gg T \cdot U \gg U = V \gg V$ where $V = T \sqcap U$.

Concatenation for flow effects over object types can be defined using the standard algorithm for state machine concatenation [135, Chapter 1, Section 2.3]. In Figure 6.13, the effect $E_1 \cdot E_2$ is derived using this algorithm. The ambiguity in the end state of E_1 and the overlap of the method a between the state S_1 in O_1 and state S_2 in O_2 are the source of the complexity in $E_1 \cdot E_2$. The traces $t_1 = \epsilon$, $t_2 = (a, A_1).\epsilon$ and $t_3 = (a, A_1).(c, C_1).\epsilon$ are all members of $\mathit{Tr}(E_1)$, after which the only trace of E_2, $t_4 = (a, A_2).(b, B_2).\epsilon$ must be permitted.

As the state machine must be deterministic, we cannot distinguish between $t_1 + t_4$ and $t_2 + t_4$ in the initial state of $E_1 \cdot E_2$. As such, the method a must return a value which can be used as an A_1 or an A_2, i.e it is required that a returns a value of type $A_1 \sqcap A_2$. If this type does not exist, then the concatenation of E_1 and E_2 is not defined.

Flow effect concatenation has the following properties:

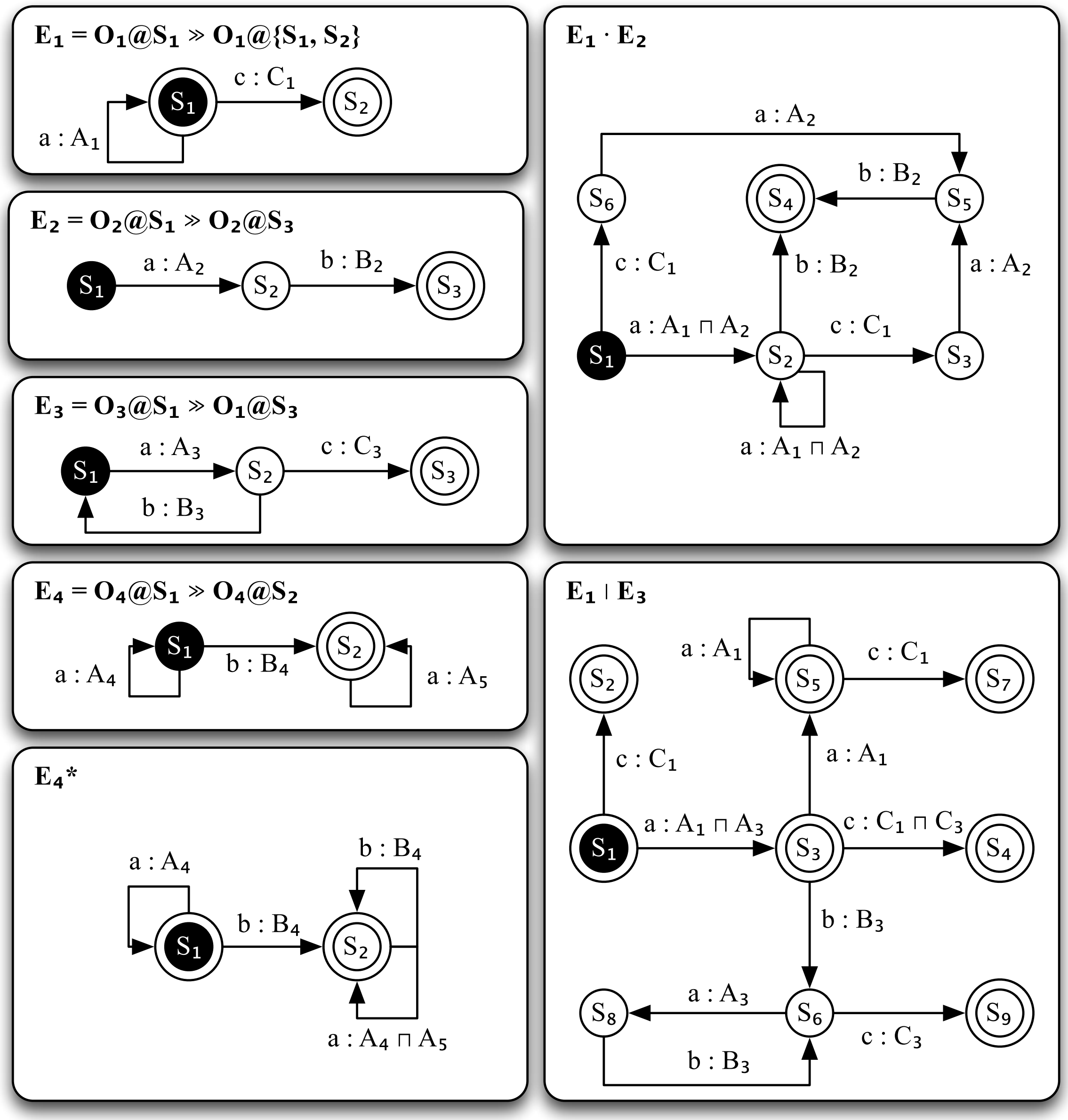


Figure 6.13: Example applications of the effect combinators

- The effect $\top \gg \top$ is the identity element for flow effect concatenation — $\top \gg \top \cdot T \gg U = T \gg U$ and $T \gg U \cdot \top \gg \top = T \gg U$.
- $T \gg U \cdot U \gg V = T \gg V$.
- If $X \gg Y = T \gg U \cdot V \gg W$ is defined, then $T \gg U \leq X \gg Z$ and $V \gg W \leq Z \gg Y$ where $Z = \mathit{remap}(X, T \gg U)$.

Effect union

$E = E_1 \mid E_2$ is referred to as the *union* of the effects E_1 and E_2. $E_1 \mid E_2$, if defined, would allow a value to be used in a way that is compatible with effect E_1 *or* E_2. Therefore, we require that $\mathit{in}(E) <: \mathit{in}(E_1)$, $\mathit{in}(E) <: \mathit{in}(E_2)$ and $\mathit{out}(E) <: \mathit{remap}(\mathit{in}(E), E_1) \sqcup \mathit{remap}(\mathit{in}(E), E_2)$. Effect union must be commutative and associative.

For update effects, union is equivalent to the meet of the two update effect types: $T \ggg U \mid V \ggg W = (T \sqcap V) \mid (U \sqcup W)$.

For flow effects over primitive types, $T \gg T \mid U \gg U = V \gg V$ where $V = T \sqcap U$. This is coincidentally the same definition as for effect concatenation.

Union for flow effects over objects types can be defined using the standard algorithm for state machine union. In Figure 6.13, the effect $E_1 \mid E_3$ is derived using this algorithm. Similar to the effect concatenation case, the ambiguity in the end state of E_1 results in a significant number of possible end states in $E_1 \mid E_3$. The initial state of $E_1 \mid E_3$ is the union of the initial states in E_1 and E_3, therefore both methods a and c are available, and a must return $A_1 \sqcap A_3$. State S_3 in $E_1 \mid E_3$ is the union of S_1 from E_1 and S_2 from E_3, therefore the method c in this state must return $C_1 \sqcap C_3$. If either of these are undefined, then $E_1 \mid E_3$ is undefined.

Flow effect union has the following properties:

- $\top \gg \top \mid T \gg U = T \gg (T \sqcup U)$ — the union of the effect $\top \gg \top$ with any other effect is equivalent to that effect being applied zero or one times.
- $T \gg U \mid T \gg V = T \gg (U \sqcup V)$.
- If $X \gg Y = T \gg U \mid V \gg W$, then $T \gg U \leq X \gg Y_1$ and $V \gg W \leq X \gg Y_2$ where $Y_1 = \mathit{remap}(X, T \gg U)$ and $Y_2 = \mathit{remap}(X, V \gg W)$ such that $Y_1 \sqcup Y_2 = Y$.

The union of a flow effect and an update effect is less straightforward than for concatentation. We must treat the union of these effects as an update effect, such that $T \gg U \sqcap V \ggg W = (T \sqcap V) \ggg (U \sqcup W)$, if defined. Without knowing which of the two effects will be applied, it is safer to treat $T \gg U$ as its update effect equivalent, $T \ggg U$.

Effect repetition

$E' = E*$ is referred to as the *Kleene star* or *repetition* of the effect $T \gg U$. If defined, E' would allow a value to be used in a way that is compatible with E *zero or more times*, in sequence. Therefore, we require that $in(E') <: in(E)$, $out(E') <: in(E)$ and $out(E') <: in(E')$.

For update effects, $(T \ggg U)* = T \ggg (T \sqcup U)$ when $U <: T$, otherwise it is not defined. For flow effects over primitive types, $(T \gg T)* = T \gg T$.

For flow effects over object types, the repetition of an effect can be generated using the standard algorithm for deriving the Kleene star of a finite state machine. In Figure 6.13, the Kleene star of the effect E_4 is generated. The existence of the method a on both the input and output states of E_4 requires that the method return $A_4 \sqcap A_5$ after the first call to b. If this undefined, then E_4* is also undefined.

6.4 Type system

We shall consider two closely related type systems for the TS language: one which uses update effects exclusively for functions, and other which uses flow effects exclusively. As shall be demonstrated, the typing rules are identical with the exception of the typing rule for function definitions, though the system with flow effects allows more terms to be typed and stronger properties can be proven.

As variables are references to values that can change type, it is necessary for the type system to express changes to the execution context. We must then be able to demonstrate that the changes in the context correspond to the changes in the store during evaluation.

Definition 6.4.1 (Contexts)**.** A context is a set of unique variable names mapped to types: $\Gamma = \overline{x_i : T_i}$. It is often convenient to treat Γ as a partial function from variable names to types, such that $dom(\Gamma) = \overline{x_i}$ and $\Gamma(x_i) = T_i$, where $x_i \in dom(\Gamma)$. A total function variant $\hat{\Gamma}(x_i)$ is defined such that $\hat{\Gamma}(x) = \Gamma(x)$ when $x \in dom(\Gamma)$, otherwise $\hat{\Gamma}(x) = \top$. ▲

An ordering is defined on contexts in Figure 6.14, such that when $\Gamma \le \Gamma'$, we say that Γ is *less specific* than Γ'. $\Gamma = \varnothing$ is the smallest (least specific) element. This ordering is a partial order up to equivalence (proof in Section A.3).

Partial join and total meet operators (up to equivalence) are also defined in Figure 6.14, which are used in the typing rule for conditionals. The proof that these operators have the expected properties is given in Section A.3.

Given a definition of a context, we can define *typings* for the terms of the language:

$$\frac{dom(\Gamma) \subset dom(\Gamma') \quad \forall x \in dom(\Gamma).\, \Gamma'(x) <: \Gamma(x)}{\Gamma < \Gamma'} \text{ CTX\_LT\_A}$$

$$\frac{\begin{array}{c} dom(\Gamma) = dom(\Gamma') \quad \forall x \in dom(\Gamma).\, \Gamma'(x) <: \Gamma(x) \\ \exists x \in dom(\Gamma).\, \Gamma'(x) \ll: \Gamma(x) \end{array}}{\Gamma < \Gamma'} \text{ CTX\_LT\_B}$$

$$\frac{dom(\Gamma) = dom(\Gamma') \quad \forall x \in dom(\Gamma).\, \Gamma'(x) \equiv \Gamma(x)}{\Gamma \equiv \Gamma'} \text{ CTX\_EQUIV}$$

$$\frac{\Gamma < \Gamma' \vee \Gamma \equiv \Gamma'}{\Gamma \leq \Gamma'} \text{ CTX\_LEQ}$$

$$\frac{\overline{x_i} = dom(\Gamma) \cup dom(\Gamma') \quad T(x) = \hat{\Gamma}(x) \sqcap \hat{\Gamma}'(x)}{\Gamma \sqcup \Gamma' = \{\, x : T(x) \mid x \in \overline{x_i} \,\}} \text{ CTX\_JOIN}$$

$$\frac{\overline{x_i} = dom(\Gamma) \cap dom(\Gamma') \quad T(x) = \Gamma(x) \sqcup \Gamma'(x)}{\Gamma \sqcap \Gamma' = \{\, x : T(x) \mid x \in \overline{x_i} \,\}} \text{ CTX\_MEET}$$

$$\frac{\Gamma_2 < \Gamma_1}{(\Gamma_1, T_1, \Gamma'_1) < (\Gamma_2, T_2, \Gamma'_2)} \text{ TY\_LT\_1} \qquad \frac{\Gamma_2 \equiv \Gamma_1 \quad T_2 \ll: T_1}{(\Gamma_1, T_1, \Gamma'_1) < (\Gamma_2, T_2, \Gamma'_2)} \text{ TY\_LT\_2}$$

$$\frac{\Gamma_2 \equiv \Gamma_1 \quad T_2 \equiv T_1 \quad \Gamma'_1 < \Gamma'_2}{(\Gamma_1, T_1, \Gamma'_1) < (\Gamma_2, T_2, \Gamma'_2)} \text{ TY\_LT\_3} \qquad \frac{\Gamma_1 \equiv \Gamma_2 \quad T_1 \equiv T_2 \quad \Gamma'_1 \equiv \Gamma'_1}{(\Gamma_1, T_1, \Gamma'_1) \equiv (\Gamma_2, T_2, \Gamma'_2)} \text{ TY\_EQUIV}$$

$$\frac{\tau < \tau' \vee \tau \equiv \tau'}{\tau \leq \tau'} \text{ TY\_LEQ}$$

Figure 6.14: Ordering and equivalance for contexts and typings

Definition 6.4.2 (Typings)**.** A *typing judgement* $\Gamma \rhd t : T \lhd \Gamma'$ defines a member of the *typing relation*. This may be interpreted as " given context Γ, the result of evaluating the term t may be considered to be of type T, after which the context will have changed to Γ' ".

In the terminology of Jim [80] and Wells [149], if $\Gamma \rhd t : T \lhd \Gamma'$ can be derived then $\tau = (\Gamma, T, \Gamma')$ is a *typing* of the term t. ▲

The typing judgements for each kind of term are defined in Figure 6.15, and may be interpreted as follows:

- T_UNIT, T_TRUE and T_FALSE are trivial — the evaluation of a unit or boolean literal does not change the context, and evaluation does not require anything specific of the context. As a result, these terms are typeable with any input context (Lemma A.10.4).
- T_FUN_FL_DEF requires that the body of a function literal is typeable using an input context built from the defined parameters, and that the types of the variables in the output context of this typing correspond to those declared on the effect for each variable. This rule forbids implicit binding of variables from the input context, as the body of the function must be typeable without any reference to the input context. Consequently, function literals can be typed with any input context, and the context is not changed by evaluation.

  T_FUN_FL_DEF defines the derivation for a function which describes the usage of parameters with flow effects, while T_FUN_UP_DEF defines the equivalent derivation for update effects. By omitting T_FUN_FL_DEF or T_FUN_UP_DEF, we produce the two variants of the type system for TS we are interested in studying.
- T_OBJECT requires that the values used for the return values of each method are typeable using an empty context, which forbids implicit binding of variables from the context in a similar manner to function literals. Implicit in this rule is the requirement that state labels be unique, and that each method within a state be unique, ensuring that the protocol is deterministic.
- T_SEQ requires that in order to type $t_a \,;\, t_b$, that there must exist a typing of t_a and t_b such that the output context of the typing of t_a is the input context of t_b. The typing of the overall statement uses the input context of t_a and the type and output context of t_b.
- T_LET is fundamentally similar to T_SEQ— for a term **let** $x = t_x$ **in** t_b, typings must exist for t_x and t_b such that the output context of t_x, with the addition of a binding of x to the type assigned to t_x, is the input context of t_b.
- T_FUN_CALL requires that in order to type $x(\overrightarrow{x_i})$, that the type of x and the types of each x_i must align such that:

$$\frac{}{\Gamma \rhd \texttt{true} : \mathbf{Bool} \lhd \Gamma}\ \text{T\_TRUE} \qquad \frac{}{\Gamma \rhd \texttt{false} : \mathbf{Bool} \lhd \Gamma}\ \text{T\_FALSE}$$

$$\frac{}{\Gamma \rhd \texttt{unit} : \mathbf{Unit} \lhd \Gamma}\ \text{T\_UNIT}$$

$$\frac{\overline{x_i : T_i} \rhd t : V \lhd \overline{x_i : U_i}}{\Gamma \rhd \lambda(\overrightarrow{x_i : T_i \gg U_i}).t : (\overrightarrow{T_i \gg U_i}) \to V \lhd \Gamma}\ \text{T\_FUN\_FL\_DEF}$$

$$\frac{\forall i, j.\ \varnothing \rhd v_{ij} : T_{ij} \lhd \varnothing}{\Gamma \rhd \left[\overline{S_i\{\overline{m_{ij} = (v_{ij}, S_{ij})}\}}\right] @S : \left\{\overline{S_i\{\overline{m_{ij} : T_{ij} \Rightarrow S_{ij}}\}}\right\} @S \lhd \Gamma}\ \text{T\_OBJECT}$$

$$\frac{x \notin \mathit{dom}(\Gamma) \quad \Gamma \rhd t_x : T_x \lhd \Gamma' \quad \Gamma', x : T_x \rhd t_b : T \lhd \Gamma'', x : T'_x}{\Gamma \rhd \mathbf{let}\ x = t_x\ \mathbf{in}\ t_b : T \lhd \Gamma''}\ \text{T\_LET}$$

$$\frac{\Gamma \rhd t_a : T_a \lhd \Gamma' \quad \Gamma' \rhd t_b : T \lhd \Gamma''}{\Gamma \rhd t_a\, ;\, t_b : T \lhd \Gamma''}\ \text{T\_SEQ}$$

$$\frac{\Gamma(x) = (\overrightarrow{E_i}) \to V \quad \forall i.\, T_i <: \mathit{in}(E_i)}{\Gamma, \overline{x_i : T_i} \rhd x(\overrightarrow{x_i}) : V \lhd \Gamma, \overline{x_i : \mathit{remap}(T_i, E_i)}}\ \text{T\_FUN\_CALL}$$

$$\frac{m : T \Rightarrow \overline{S'} \in O@\overline{S}}{\Gamma, x : O@\overline{S} \rhd x.m : T \lhd \Gamma, x : O@\overline{S'}}\ \text{T\_METH\_CALL}$$

$$\frac{\begin{array}{c}\Gamma \rhd t_c : \mathbf{Bool} \lhd \Gamma_1 \\ \Gamma_1 \rhd t_t : T_t \lhd \Gamma_2 \quad \Gamma_1 \rhd t_f : T_f \lhd \Gamma_3\end{array}}{\Gamma \rhd \mathbf{if}\ t_c\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f : T_t \sqcup T_f \lhd \Gamma_2 \sqcap \Gamma_3}\ \text{T\_IF}$$

$$\frac{\begin{array}{c}\Gamma_1 \rhd t_c : \mathbf{Bool} \lhd \Gamma_2 \quad \Gamma_3 \rhd t_b : T_b \lhd \Gamma_4 \\ E_c(x) = \mathit{extract}(x, \Gamma_1, \Gamma_2) \quad E_b(x) = \mathit{extract}(x, \Gamma_3, \Gamma_4) \\ \Gamma' = \{x : \mathit{remap}(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))\ast) \mid x \in \mathit{dom}(\Gamma)\}\end{array}}{\Gamma \rhd \mathbf{while}\ t_c\ \mathbf{do}\ t_b : \mathbf{Unit} \lhd \Gamma'}\ \text{T\_WHILE\_A}$$

$$\frac{\Gamma \rhd t : T' \lhd \Gamma' \quad T' <: T}{\Gamma \rhd t : T \lhd \Gamma'}\ \text{T\_SUB}$$

$$\frac{\Gamma, x : T \rhd t : U \lhd \Gamma', x : V \quad T \gg V \le T' \gg V'}{\Gamma, x : T' \rhd t : U \lhd \Gamma', x : V'}\ \text{T\_WIDEN\_FL\_EFF}$$

Figure 6.15: Type rules

$$\frac{\overline{x_i : T_i} \rhd t : V \lhd \overline{x_i : U_i}}{\Gamma \rhd \lambda(\overrightarrow{x_i : T_i \ggg U_i}).t : (\overrightarrow{T_i \ggg U_i}) \to V \lhd \Gamma} \text{ T\_FUN\_UP\_DEF}$$

$$\frac{\Gamma \rhd t_c : \mathbf{Bool} \lhd \Gamma' \quad \Gamma' \rhd t_b : T \lhd \Gamma}{\Gamma \rhd \mathbf{while}\ t_c\ \mathbf{do}\ t_b : \mathbf{Unit} \lhd \Gamma'} \text{ T\_WHILE\_B}$$

$$\frac{\Gamma, x : T \rhd t : U \lhd \Gamma', x : V \quad T \ggg V \leq T' \ggg V'}{\Gamma, x : T' \rhd t : U \lhd \Gamma', x : V'} \text{ T\_WIDEN\_UP\_EFF}$$

Figure 6.16: Type rule variants for update effects

for flow effects:

$$\mathit{extract}(x, \Gamma_1, \Gamma_2) = \left\{ \begin{array}{ll} \Gamma_1(x) \gg \Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \\ \top \gg \top & \text{otherwise} \end{array} \right\}$$

for update effects:

$$\mathit{extract}(x, \Gamma_1, \Gamma_2) = \left\{ \begin{array}{ll} \Gamma_1(x) \ggg \Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \\ \top \ggg \top & \text{otherwise} \end{array} \right\}$$

Figure 6.17: Definition of *extract* operation

- The correct number of parameters are passed.
- The known type for each x_i in the input context must be a subtype of the required type specified in the function type of x.

The type of each x_i after the function call is defined by *remap*, which is dependent upon the effect type used.

- T_METH_CALL requires that in order to type $x.m$, x must have an type $O@\overline{S}$ such that the method m is available in all of the states $S \in \overline{S}$. The type of the term and the type of x after the call are both derived directly from the definition of the protocol.

- T_IF is somewhat similar to T_SEQ— the output context used to type the condition must be usable as the input context for typings of both the true and false branches. The output context for the entire conditional is the meet of the contexts for the true and false branches.

- T_SUB is the standard subsumption rule — if t can be typed with type T, and $T <: U$, then t can also be typed with U.

- T_WIDEN_FL_EFF provides a means to modify a typing built using the flow effect rules such that a particular variable has a wider effect. An effect $T \gg V$ extracted from a typing $\Gamma, x : T \rhd t : U \lhd \Gamma', x : V$ expresses the type tranformation that may occur on x by evaluating the term t. We may safely replace the effect with one that represents a larger set of possible interactions: $T \gg V \leq T' \gg V'$.

The equivalent for the type system using update effects is given by rule T_WIDEN_UP_EFF. The definition of the sub-effect relationship for update effects is such that this rule allows the equivalent of subsumption in the output context, and a more precise type to be used in the input context.

For both T_WIDEN_FL_EFF and T_WIDEN_UP_EFF, we are guaranteed that $T' <: T$, therefore we are guaranteed that T' may be used in any way that T could be used.

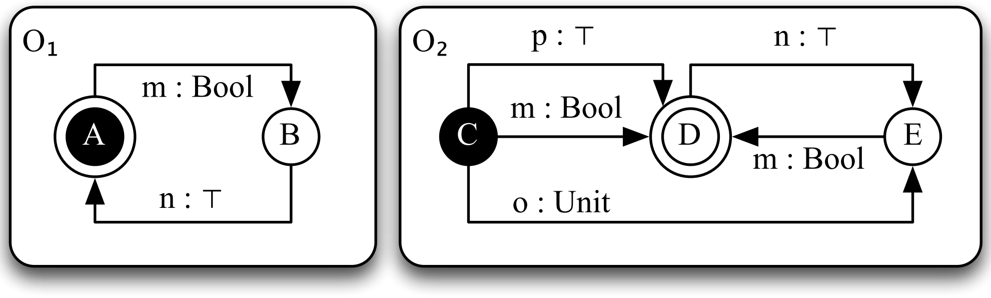
6.4.1 While loops

The rule T_WHILE_A provides typings for while loops and is the most complex rule in the type system, requiring a much more in depth explanation. Given typings of the terms t_c and t_b, we require that the observed effects in these typings ($E_c(x)$ and $E_b(x)$ respectively) can be composed into a larger effect $E_c(x) \cdot (E_b(x) \cdot E_c(x))*$ which represents the evaluation of the loop condition, followed by zero or more evaluations of the loop body and loop condition. Given an input context Γ, a typing may be derived if *remap* may be applied to each variable of Γ using the extracted effects.

The $\mathit{extract}(x, \Gamma, \Gamma')$ function, defined in Figure 6.17, is used to extract the observed effects for the variable x from an input context Γ and an output context Γ'. This function has two alternate definitions to produce either flow or update effects as required by the type system in use. Where a variable does not occur in both contexts, the effect $\top \gg \top$ is produced to represent the fact that the variable is not required for the typing.

The T_WHILE_A rule is perhaps not ideal as it is not algorithmic — the rule does not use Γ or Γ' in the typings of the condition or body sub-terms. As such, a type checker cannot necessarily derive a typing for a while loop in a straightforward manner using this rule.

One may expect that the simpler algorithmic rule T_WHILE_B (shown in Figure 6.16) would have been sufficient. Such a rule certainly works for update effects, but not for flow effects. A crucial property of the type system is that if a term t may be typed such that $\Gamma, x : T \rhd t : U \lhd \Gamma', x : V$, and we have some type $T' <: T$, then a typing must exist for t given input context $\Gamma, x : T'$. This property, referred to as *upgrading* and defined formally in Section 6.4.3, cannot be supported by the above rule. Consider the term $t = \mathbf{while}\ x.m\ \mathbf{do}\ x.n$, and the object protocols shown below:



It is clear that we can type t with $\Gamma = x : O_1@A$ and $\Gamma' = x : O_1@B$ using the proposed rule. It can also be observed that $O_2@C <: O_1@A$, therefore we might expect to find a typing for t with $\Gamma = x : O_2@C$. The sub-terms may be typed such that $x : O_2@C \rhd x.m : \mathbf{Bool} \lhd x : O_2@D$ and $x : O_2@D \rhd x.n : \top \lhd x : O_2@E$, but there is no way to modify this latter typing such that $x : O_2@D \rhd x.n : \top \lhd x : O_2@C$.

Other algorithmic rules have been tried, such as a rule which allows the output context of the body typing to be such that every variable is a subtype of the input context condition typing — this will not work for our example types either, as $O_2@E$ is not a subtype of $O_2@C$.

The following adaptation of T_WHILE_A also does not work:

$$\frac{\begin{array}{c}\Gamma \rhd t_c : \mathbf{Bool} \lhd \Gamma' \quad \Gamma' \rhd t_b : T_b \lhd \Gamma'' \\ E_c(x) = \mathit{extract}(x, \Gamma, \Gamma') \quad E_b(x) = \mathit{extract}(x, \Gamma', \Gamma'') \\ \Gamma''' = \{x : \mathit{remap}(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*) \mid x \in \mathit{dom}(\Gamma)\}\end{array}}{\Gamma \rhd \mathbf{while}\ t_c\ \mathbf{do}\ t_b : \mathbf{Unit} \lhd \Gamma'''}$$

The reason for this is that the effects extracted for the condition and body often describe more than what is actually required — the effect on x extracted from the typing $x : O_2@C \rhd x.m : \mathbf{Bool} \lhd x : O_2@D$ implies that when evaluating the condition, either m or p may be called. The effect combinator $E_c \cdot (E_b \cdot E_c)*$ will generally demand much more from a type than the while loop actually requires.

In reality, we know only m will be called in evaluating the condition but there is no way to represent this as an effect in the object protocol O_2 without changing it. The object protocol O_1 actually represents the minimum requirement for x to type $\mathbf{while}\ x.m\ \mathbf{do}\ x.n$ — if the typings for the condition and body are used based on O_1 and with the rule T_WHILE_A, then any subtype of $O_1@A$ may be used as the type of x in the input to this while loop and a typing will exist. Of course, the selection of O_1 as the minimal object protocol to represent the requirements of the loop is a *type inference* problem, which is specifically looking for a *principal typing* for the condition and body terms. Principal typings are defined and studied later in Section 6.5.

$$\frac{}{\varnothing \vdash \varnothing}\ \text{ST\_EMPTY} \qquad \frac{\forall i. \varnothing \rhd v_i : T_i \lhd \varnothing}{\overline{x_i : T_i} \vdash \overline{x_i \mapsto v_i}}\ \text{ST-VARS} \qquad \frac{\Gamma \vdash \mu \quad \overline{x_i \notin dom(\mu)}}{\Gamma \vdash \mu, \overline{x_i \mapsto v_i}}\ \text{ST-EXTRA}$$

Figure 6.18: Store typing judgements

$$\frac{}{id : Id} \qquad \frac{\varnothing \rhd v : T \lhd \varnothing}{replace(x, v) : Replace(x, T)} \qquad \frac{}{call(x, m) : Call(x, m)}$$

Figure 6.19: Definition of store update and context update compatibility relation, $\upsilon : \Upsilon$

6.4.2 Soundness

In order to demonstrate that this type system prevents the expression of terms which may become stuck, it is necessary to relate contexts to stores. The *well-typed store* relation $\Gamma \vdash \mu$ (see Figure 6.18) dictates when a context is a valid abstraction of a store.

For each reduction rule, we also require a way to relate changes to a store to changes in a context which preserves the well-typed store property. For each reduction rule we may derive a *store update* function for each reduction rule which describes the relationship between μ and μ'. Store update functions are denoted by the meta-variable υ. There are three kinds of store update function:

- id — the identity function: $id(\mu) = \mu$.
- $replace(x, v)$ — replaces the mapping of x in a store to the value v: $replace(x, v)(\mu) = \mu[x \mapsto v]$.
- $call(x, m)$ — calls the method m on the object mapped to x in the store. Unlike id and $replace(x, v)$ this context update function places specific demands on the existing contents of μ. In order for $call(x, m)(\mu)$ to be defined, it must be the case that method m is available in $\mu(x)$. If $\mu(x) = o@S$ where $o = [\ldots S\{\ldots, m = (v, S'), \ldots\}]$, then $call(x, m)(\mu) = \mu[x \mapsto o@S']$.

For each υ there are corresponding *context update* functions, which are denoted by the meta-variable Υ: Id, $Replace(x, T)$ and $Call(x, m)$. Where $\Gamma \vdash \mu$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$, we say that υ and Υ are *compatible*, written $\upsilon : \Upsilon$ — this relationship is defined in Figure 6.19.

We can now state the soundness property for the TS language:

Theorem (Progress and Preservation)**.** *Given a non-value term t such that $\Gamma \rhd t : T \lhd \Gamma'$ and a store μ such that $\Gamma \vdash \mu$, then:*

1. *There exists a term t' and store update function υ such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$.*

2. *There exists an* Υ *such that* $\upsilon : \Upsilon$*, and that* $\Upsilon(\Gamma) \vdash \upsilon(\mu)$.

3. *There exists a* $\Gamma'' \geq \Gamma'$ *such that* $\Upsilon(\Gamma) \rhd t' : T \lhd \Gamma''$.

This soundness property is proven for the flow effect variant of the type system in Theorem A.1.1, and for update effects in Theorem A.1.2. Intuitively, this soundness property means that given a typing of a term with a corresponding well-typed store:

- The term will not become stuck during evaluation.
- The store will remain well-typed during evaluation.
- Both the type of the term and the types of the variables in the output context will monotonically increase in precision.

6.4.3 Properties of typings

Multiple typings are likely to exist for a term, and it is desirable to be able to state whether one typing is *stronger* than another: informally, which has weaker requirements in the input context, and given this input context derives a stronger result type and provides stronger guarantees in the output context. A stronger typing has less extraneous details, and provides a more precise description of the requirements and behaviour of a term.

We may define this formally using a *lexicographic ordering* of the components of the typing. The rules ordering rules for contexts and typings are defined in Figure 6.14, which provide the means to define what the "strongest" typing may be for a particular term.

Definition 6.4.3 (Context ordering)**.** Γ is *less specific* than Γ', written $\Gamma < \Gamma'$, if rule CTX_LT_A or CTX_LT_B is satisfied. Γ is *equivalent to* Γ', written $\Gamma \equiv \Gamma'$, if the rule CTX_EQUIV is satisfied. Naturally, if either $\Gamma < \Gamma'$ or $\Gamma \equiv \Gamma'$, then $\Gamma \leq \Gamma'$ (rule CTX_LEQ). ▲

The context ordering is a partial order up to equivalence, as proven in Lemma A.3.6 and Lemma A.3.7. Given this partial ordering over contexts and the subtyping relation we can define a partial ordering over typings as a lexicographic ordering:

Definition 6.4.4 (Typing ordering)**.** $\tau_1 = (\Gamma_1, T_1, \Gamma'_1)$ is *weaker than* $\tau_2 = (\Gamma_2, T_2, \Gamma'_2)$, written $\tau_1 < \tau_2$, if any of the following conditions hold:

- $\Gamma_2 < \Gamma_1$ (rule TY_LT_1)
- $\Gamma_2 \equiv \Gamma_1 \wedge T_1 \lll: T_2$ (rule TY_LT_2)

- $\Gamma_2 \equiv \Gamma_1 \wedge T_1 \equiv T_2 \wedge \Gamma'_1 < \Gamma'_2$ (rule TY_LT_3)

τ_1 is *equal to* τ_2 if each component is equal (rule TY_EQUIV). The satisfaction of either provides the partial order $\tau_1 \leq \tau_2$ (up to equivalence, defined by rule TY_LEQ). ▲

One important property that should exist for the type system is that if we can derive a typing $\Gamma_1 \ \triangleright t : T \triangleleft \ \Gamma_3$, and we have a *more specific* input context $\Gamma_2 \geq \Gamma_1$, it should be possible to derive a new typing using Γ_2 as the input:

Lemma (Upgrading)**.** *Let* $\Gamma_1 \leq \Gamma_2$ *and t be a term such that* $\Gamma_1 \ \triangleright t : U \triangleleft \ \Gamma_3$. *It follows that there exists a* $\Gamma_4 \geq \Gamma_3$ *such that* $\Gamma_2 \ \triangleright t : U \triangleleft \ \Gamma_4$.

This is proven for flow effects as Lemma A.10.6, and for upgrade effects as Lemma A.10.7. The flow effect proof does in fact provide the exact definition of Γ_4. This lemma fits the intuition that a more specific input context, where at least one variable has a more precise type, should produce a more specific output context for the same term. Such typings are *weaker*, based on the defined partial ordering for typings.

6.5 Principal typings and typing schemes

Out of all the possible typings for a term, it is useful to know if any captures the essence of the behaviour of the term. This typing is the *principal typing*:

Definition 6.5.1 (Principal typings)**.** Let $\overline{\tau}$ by the set of all typings of t. The principal typing (up to equivalence) τ' of t is the *strongest* typing of t, if one exists: $\forall \tau \in \overline{\tau}. \, \tau \leq \tau'$.

If $\tau' = (\Gamma, T, \Gamma')$ is the principal typing of a term t, then Γ is the *principal input context* for t. Given Γ, T is the *principal type* of t and Γ' is the *principal output context*. The type of a variable $x \in dom(\Gamma)$ shall be referred to as a *principal variable input type* of x in t. Similarly, the type of $x \in dom(\Gamma')$ shall be referred to as the *principal variable output type* of x in t. ▲

The intuition behind this definition of a principal typing is that it is a typing which has the weakest possible requirements in the input context, and from this derives the most specific possible type and strongest guarantees in the output context.

A straightforward observation about principal typings is that the domains of the principal input and output contexts must be equal, and contain only the free variables of t. Where flow effects are used, the effect $extract(x, \Gamma, \Gamma')$ is the *principal effect* of x for t. This is defined such that if x is not free in t, then $\top \gg \top$ is the principal effect of x in t, which is to say the term has no effect on x.

Function f

```
\(x, y).(
  x.a;
  if y then ( x.a; x.b )
  else ( x.m; x.n );
  x.z
)
```

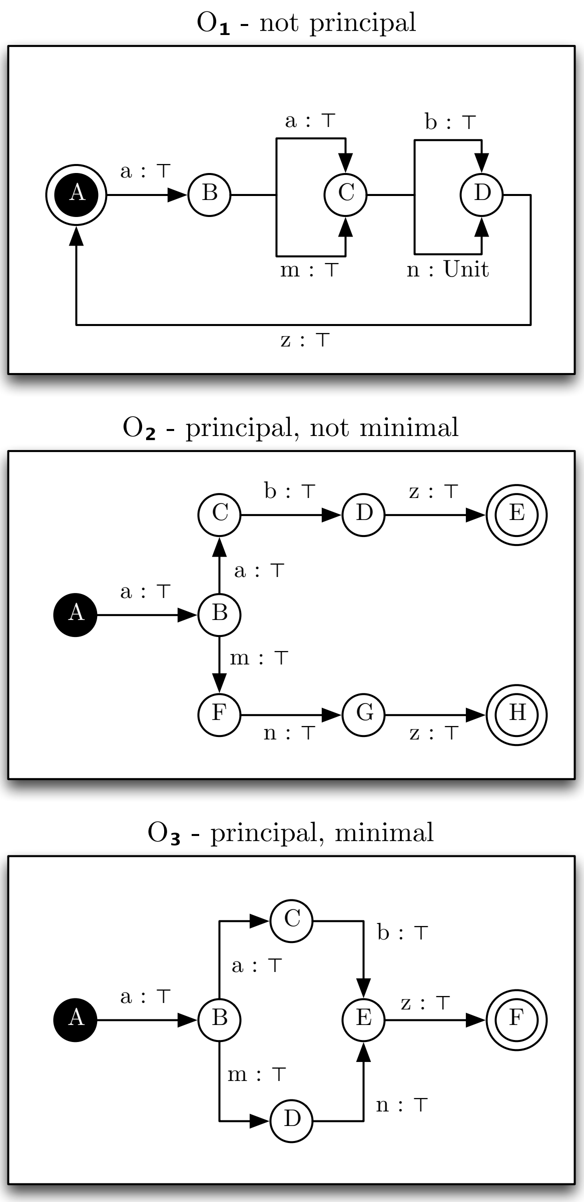


Figure 6.20: Possible types for x in function f

Given a term t which interacts with an object variable o, the principal effect of o in t has the smallest possible trace set, permitting only those sequences of methods calls that could occur in t, with the least specific return types on methods that would allow t to be typeable.

Consider the body term of example function f shown in Figure 6.20. It is clear from the usage of y that its principal variable input type must be $\mathbf{Bool}$ as demanded by the type judgement of an if-then-else statement, while x is an object. We could ascribe any of the object types $O_1@A$, $O_2@A$ and $O_3@A$ to the variable x, and after evaluation the object types would be $O_1@A$, $O_2@\{E, H\}$ and $O_2@F$ respectively. $O_1@A$ is not a principal variable input type for x as it will permit many more sequences of method calls than is strictly required, and also has a lesser return type for the method n than is necessary. $O_2@A$ is the principal variable input type for x, as is the type $O_3@A$, which is isomorphic. It is arguable that $O_3@A$ is a preferable solution to $O_2@A$, as it has a minimal object protocol. However as they are equivalent it is of little consequence which is chosen, and if the latter is preferable for display to a user it can be computed on demand.

The principal typing of the body of function f is $x : O_3@A, y : \mathbf{Bool} \rhd t : \top \lhd x : O_3@F, y : \mathbf{Bool}$. From this, it follows that the principal type of f itself is $(O_3@A \gg O_3@F, \mathbf{Bool} \gg \mathbf{Bool}) \to \top$.

Unfortunately, principal typings do not exist for all typeable terms in TS, regardless of whether update or flow effects are used.

Theorem 6.5.1. *Principal typings do not exist for all terms.*

Proof. by contradiction.

Let $t = f(x)$. Assume a principal typing $\tau_1 = (\Gamma_1, T_1, \Gamma_1')$ exists for t.

Let $\Gamma_2 = \{f : (\mathbf{Bool} \gg \mathbf{Bool}) \to \top, x : \mathbf{Bool}\}$ and $\Gamma_3 = \{f : (\mathbf{Unit} \gg \mathbf{Unit}) \to \top, x : \mathbf{Unit}\}$. It can be observed that $\tau_2 = (\Gamma_2, \top, \Gamma_2)$ and $\tau_3 = (\Gamma_3, \top, \Gamma_3)$ are typings of t.

As τ_1 is the principal typing of t, it follows that $\tau_2 \leq \tau_1$ and $\tau_3 \leq \tau_1$, meaning $\Gamma_1 \leq \Gamma_2$ and $\Gamma_1 \leq \Gamma_3$.

The minimal domain of Γ_1 is $\{f, x\}$ as the term t cannot be typed without both f and x in the input context by rule T_FUN_CALL. It follows that we must have $\Gamma_2(f) <: \Gamma_1(f)$ and $\Gamma_3(f) <: \Gamma_1(f)$.

By SUB_FN, it must be the case that $\Gamma_1(f) = (U \gg V) \to W$ such that $\mathbf{Bool} \gg \mathbf{Bool} \leq U \gg V$ and $\mathbf{Unit} \gg \mathbf{Unit} \leq U \gg V$.

By the definition of the sub-effect relation it must be the case that $U <: \mathbf{Bool}$ and $U <: \mathbf{Unit}$. Therefore, $U <: \mathbf{Bool} \sqcap \mathbf{Unit}$. The meet of $\mathbf{Bool}$ and $\mathbf{Unit}$ is undefined, therefore the existence of τ_3 is a contradiction — there is no principal typing for t. □

Generally, free variables used as functions pose a problem: given a function known to have $i > 0$ parameters, the principal type $(\overrightarrow{T_i \gg U_i}) \to V$ can only exist if a lower bound exists for each T_i. Lower bounds for each T_i only exist where all parameters are bound variables, such as in **let** x = `true` **in** $f(x)$, which has a principal input type for f of $(\mathbf{Bool} \ggg \top) \to \top$ or $(\mathbf{Bool} \gg \mathbf{Bool}) \to \top$ depending on which kind of effects are used.

It is believed that principal typings do exist for all terms which have no free variables that are used as functions:

Hypothesis. *Let t be a term with a set of free variables $\overline{x_i}$, and that there does not exist a sub-term of t of form $y(\overrightarrow{z_i})$ where $y \in \overline{x_i}$. A principal typing exists for t.*

No attempt has yet been made to prove this hypothesis.

6.5.1 Introducing $\bot$ for principal typings

If a least element can be introduced for the set of all types, principal typings may be possible for all terms. This can be achieved where update effects are used exclusively by introducing a `Bottom` (or $\bot$) type to the language: $\forall T. \bot <: T$. This would make the set of all update effects a lattice, up to equivalence, with $\bot \ggg \top$ as the greatest element and $\top \ggg \bot$ as the least element. A function type $(\overrightarrow{\bot \ggg \top}) \to \top$ of arity n is then an upper bound for all functions of the same arity, meaning $f(x)$ has a principal typing $(\Gamma, \top, \Gamma)$ where $\Gamma = f : (\bot \ggg \top) \to \top, x : \bot$. Arguably, such a principal typing is of little value, as it describes an impossible requirement for the variable x — there should be no values of type $\bot$. Additionally, no attempt has yet been made to determine whether the language is still sound when a $\bot$ type is included.

With flow effects, no least effect type can exist as such a least effect would need to contain no traces, but then could not be considered valid. As such, the addition of $\bot$ does not affect the existence of principal typings where flow effects are used.

6.6 Type inference

The lack of principal types in general for the TS language means that an alternative approach is required for capturing the essential requirements and effects of all terms. By using *type variables* to represent types for which we have incomplete information, it may be possible to capture a minimal representation of the requirements and effect of a term.

A notational convention is adopted from this point forward where a previously defined meta-variable with a dot above it indicates that it may contain type variables. For example, $\dot{T}$

is a *type expression* that may contain a type variable, such as $(\overrightarrow{\alpha_i \gg \alpha_i'}) \rightarrow \mathbf{Bool}$. The dot distinguishes this from T, which must be a concrete type with no type variables.

Using type variables, we can define *typing schemes*, adapting the definition and notation used by Simonet [138]:

Definition 6.6.1 (Typing schemes)**.** A typing scheme $\sigma = \forall\overline{\alpha}[C].\dot{\tau}$ is a typing $\dot{\tau}$ with a universally quantified set $\overline{\alpha}$ of *type variables* and a constraint C over those variables. ▲

Figure 6.21 defines the grammar for typing schemes, constraints and type expressions. Given a type variable substitution $\rho = \overline{\alpha_i \mapsto \dot{T}_i}$ (with the application of a substitution defined in Figure 6.22) for each variable in $\overline{\alpha}$ which satisfies a constraint C generated by the constraint typing rules defined in Figure 6.24, applying the substitution to $\dot{\tau}$ produces a normal typing τ. The judgement $\rho \vdash C$ defines what it means for a substitution to satisfy a constraint, and is defined in Figure 6.23.

Finally, if a substitution ρ satisfies the constraint of a typing, and provides a substitution for every universally quantified variable, we say that ρ satisfies the scheme:

$$\frac{\overline{\alpha} \subseteq dom(\rho) \quad \rho \vdash C}{\rho \vdash \forall\overline{\alpha}[C].\dot{\tau}} \text{ SAT\_SCHEME}$$

It remains to show that an appropriate typing scheme can be generated for all terms t, such that all satisfying substitutions produce valid typings — this is the *correctness* property. It would also be desirable to show that all typings of a term t are weaker than a typing which could be produced by a satisfying substitution – this is the *completeness* property.

The grammar for the TS language, as defined in Figure 6.2, includes fixed effect type annotations for function values which must be supplied by the programmer. We define two variants of this term language, $\hat{t}$ in which there are no type annotations on function values (they are of form $\lambda(\overrightarrow{x_i}).t$), and $\dot{t}$ in which function values have type expressions that may contain variables (they are of form $\lambda(\overrightarrow{x_i : \dot{E}}).$). Terms from the original grammar shall be referred to as *annotated* terms, while $\hat{t}$ terms are *bare* terms and $\dot{t}$ terms are *inferred* terms. The function $strip(t) = \hat{t}$ removes all type annotations from a normal term to produce a typeless term: $strip(\lambda(\overrightarrow{x_i : E_i}).t) = \lambda(\overrightarrow{x_i}).t$.

Given a bare term, we generate an inferred term with an associated typing scheme using constraint typing judgements of form $\Delta \vdash \hat{t} \Rightarrow \dot{t} : \dot{T} \mid_{\chi} \mathcal{C}$, adapted from the style of Pierce [118, Chapter 22]. Where $\hat{t}$ is syntactically unmodified in a judgement, we omit the $\Rightarrow \dot{t}$ part of the judgement, such that $\Delta \vdash \hat{t} : \dot{T} \mid_{\chi} \mathcal{C}$, is equivalent to $\Delta \vdash \hat{t} : \hat{t} \mid_{\dot{T}} \chi\mathcal{C}$.

The constraint typing judgements are defined in Figure 6.24. The components of this judgement besides the term are as follows:

σ	::=	$\forall\overline{\alpha}[C].\dot{\tau}$	**typing scheme**
$\dot{\tau}$	::=	$(\dot{\Gamma}, \dot{T}, \dot{\Gamma})$	**typing with variables**
$\dot{\Gamma}$	::=	$\overline{x_i : \dot{T}_i}$	**context with type variables**
Δ	::=	$\overline{x_i : \dot{E}_i}$	**effect context**
C	::=		**type constraint**
		`true`	
	$\mid$	$\dot{T} <: \dot{U}$	subtype constraint
	$\mid$	$valid(\dot{E})$	effect validity
	$\mid$	$C \wedge C'$	conjunction
$\dot{T}, \dot{U}, \dot{V}$	::=		**type expression**
		$\top$	top type
	$\mid$	$\mathbf{Unit}$	unit type
	$\mid$	$\mathbf{Bool}$	boolean type
	$\mid$	$(\overrightarrow{\dot{E}_i}) \to \dot{T}$	function type
	$\mid$	$\left\{\overline{S_i\{\overline{m_{ij} : \dot{T}_{ij} \Rightarrow S_{ij}}\}}\right\} @\overline{S}$	object type
	$\mid$	$\dot{T} \sqcup \dot{U}$	join
	$\mid$	$\dot{T} \sqcap \dot{U}$	meet
	$\mid$	$remap(\dot{T}, \dot{U} \gg \dot{V})$	remap application
	$\mid$	α	type variable
$\dot{E}, \dot{F}$	::=		**effect expression**
		$\dot{T} \gg \dot{U}$	flow effect
	$\mid$	$\dot{T} \ggg \dot{U}$	update effect
	$\mid$	$\dot{T}$	sugar for $\dot{T} \gg \dot{T}$
	$\mid$	$\dot{E} \cdot \dot{F}$	effect concatenation
	$\mid$	$\dot{E} \mid \dot{F}$	effect choice
	$\mid$	$\dot{E}\star$	effect repeat

Figure 6.21: Grammar for typing schemes, constraints, type and effect expressions

$$
\begin{array}{lcl}
\rho(\top) & = & \top \\
\rho(\mathbf{Unit}) & = & \mathbf{Unit} \\
\rho(\mathbf{Bool}) & = & \mathbf{Bool} \\
\rho((\overrightarrow{\dot{E}_i}) \to \dot{V}) & = & (\overrightarrow{\rho(\dot{E}_i)}) \to \rho(\dot{V}) \\
\\
\rho(\{\overline{S_i\{\overline{m_{ij} : \dot{T}_{ij} \Rightarrow S_{ij}}\}}\}@\overline{S}) & = & \{\overline{S_i\{\overline{m_{ij} : \rho(\dot{T}_{ij}) \Rightarrow S_{ij}}\}}\}@\overline{S} \\
\\
\rho(\dot{T} \sqcup \dot{U}) & = & \rho(\dot{T}) \sqcup \rho(\dot{U}) \\
\rho(\dot{T} \sqcap \dot{U}) & = & \rho(\dot{T}) \sqcap \rho(\dot{U}) \\
\\
\rho(\mathit{remap}(\dot{T}, \dot{E})) & = & \mathit{remap}(\rho(\dot{T}), \rho(\dot{E})) \\
\\
\rho(\dot{T} \gg \dot{U}) & = & \rho(\dot{T}) \gg \rho(\dot{U}) \\
\rho(\dot{T} \ggg \dot{U}) & = & \rho(\dot{T}) \ggg \rho(\dot{U}) \\
\\
\rho(\alpha) & = & \left\{ \begin{array}{ll} \dot{T} & \alpha \mapsto \dot{T} \in \rho \\ \alpha & \mathit{otherwise} \end{array} \right\}
\end{array}
$$

Figure 6.22: Definition of type variable substitution

$$\frac{}{\rho \vdash \texttt{true}}\ \text{SAT\_TRUE} \qquad \frac{\rho \vdash C \quad \rho \vdash C'}{\rho \vdash C \wedge C'}\ \text{SAT\_CONJ}$$

$$\frac{\rho(\dot{T}) = T \quad \rho(\dot{U}) = U \quad T <: U}{\rho \vdash \dot{T} <: \dot{U}}\ \text{SAT\_SUB}$$

$$\frac{\rho(\dot{T}) = T \quad \rho(\dot{U}) = U \quad \text{VALID\_FL\_EFF}TU}{\rho \vdash \mathit{valid}(\dot{T} \gg \dot{U})}\ \text{SAT\_VALID\_FL}$$

$$\frac{\rho(\dot{T}) = T \quad \rho(\dot{U}) = U}{\rho \vdash \mathit{valid}(\dot{T} \ggg \dot{U})}\ \text{SAT\_VALID\_UP}$$

Figure 6.23: The constraint satisfaction judgement, $\rho \vdash C$

$$\dfrac{}{\varnothing \vdash \texttt{true} : \mathbf{Bool} \mid_{\varnothing} \texttt{true}}\ \text{TC\_TRUE} \qquad \dfrac{}{\varnothing \vdash \texttt{false} : \mathbf{Bool} \mid_{\varnothing} \texttt{true}}\ \text{TC\_FALSE}$$

$$\dfrac{}{\varnothing \vdash \texttt{unit} : \mathbf{Unit} \mid_{\varnothing} \texttt{true}}\ \text{TC\_UNIT}$$

$$\dfrac{\Delta \vdash \hat{t} \Rightarrow \dot{t} : \dot{T} \mid_{\chi} C \quad dom(\Delta) \subseteq \overline{x_i}}{\varnothing \vdash \lambda(\overrightarrow{x_i}).\hat{t} \Rightarrow \lambda(\overrightarrow{x_i : \Delta(x_i)}).\dot{t} : (\overrightarrow{\Delta(x_i)}) \to \dot{T} \mid_{\chi} C}\ \text{TC\_FUN}$$

$$\dfrac{\begin{array}{c}\forall i, j.\varnothing \vdash \hat{v}_{ij} \Rightarrow \dot{v}_{ij} : \dot{T}_{ij} \mid_{\chi_{ij}} C_{ij} \quad \bigcap(\overline{\chi_{ij}}) = \varnothing \quad \chi = \bigcup(\overline{\chi_{ij}}) \quad C = \bigwedge(\overline{C_{ij}}) \\ \hat{o} = \left[\overline{S_i\{\overline{m_{ij} = (\hat{v}_{ij}, S_{ij})}\}}\right] \quad \dot{o} = \left[\overline{S_i\{\overline{m_{ij} = (\dot{v}_{ij}, S_{ij})}\}}\right]\end{array}}{\varnothing \vdash \hat{o}@S \Rightarrow \dot{o}@S : \left\{\overline{S_i\{\overline{m_{ij} : \dot{T}_{ij} \Rightarrow S_{ij}}\}}\right\} @\overline{S} \mid_{\chi} C}\ \text{TC\_OBJ}$$

$$\dfrac{\chi = \{\alpha\} \quad O = \{S\{m : \alpha \Rightarrow S'\}S'\{\}\}}{x : O@S \gg O@S' \vdash x.m : \alpha \mid_{\chi} \texttt{true}}\ \text{TC\_METH}$$

$$\dfrac{\begin{array}{c}\chi = \overline{\alpha_i}, \overline{\alpha_i'}, \overline{\alpha_i''}, \alpha \quad \Delta = \overline{x_i : \dot{E}_i}, x : (\overrightarrow{\alpha_i \gg \alpha_i'}) \to \alpha \\ \dot{E}_i = \alpha_i'' \gg remap(\alpha_i'', \alpha_i \gg \alpha_i') \quad C = \bigwedge(\overline{valid(\alpha_i \gg \alpha_i') \wedge \alpha_i'' <: \alpha_i})\end{array}}{\Delta \vdash x(\overrightarrow{x_i}) : \alpha \mid_{\chi} C}\ \text{TC\_CALL\_FL}$$

$$\dfrac{\begin{array}{c}C = \bigwedge(\overline{\alpha_i'' <: \alpha_i}) \quad \chi = \overline{\alpha_i}, \overline{\alpha_i'}, \overline{\alpha_i''}, \alpha \\ \Delta = \overline{x_i : \alpha_i'' \ggg \alpha_i'}, x : (\overrightarrow{\alpha_i \ggg \alpha_i'}) \to \alpha\end{array}}{\Delta \vdash x(\overrightarrow{x_i}) : \alpha \mid_{\chi} C}\ \text{TC\_CALL\_UP}$$

Figure 6.24: Constraint typing rules (Page 1 of 2)

- $\Delta = \overline{x_i : \dot{E}_i}$ is an *effect context* which defines the effect on each variable used in t. Δ
- $\chi = \overline{\alpha}$ is the set of type variables generated.
- $\dot{T}$ is the type of the term, which is potentially a type variable.
- $\mathcal{C}$ is a constraint, or conjunction of constraints on the type variables in χ.

It is often convenient to treat an effect context as if it is a total function from variables to effects: $\Delta(x) = \top \gg \top$ if an explicit effect mapping is not defined for x in Δ. Input and output contexts can be extracted from an effect context $\Delta = \overline{x_i : \dot{E}_i}$ using the functions $in(\Delta) = \overline{x_i : in(\dot{E}_i)}$ and $out(\Delta) = \overline{x_i : out(\dot{E}_i)}$. Where it is necessary to describe removing an effect for variable x from an effect context, the notation Δ/x is used, defined as follows:

$$\Delta/x = \left\{ \begin{array}{ll} \Delta & \text{when } x \notin dom(\Delta) \\ \Delta' & \text{when } \Delta = \Delta', x : \dot{E} \end{array} \right\}$$

$$\frac{\begin{array}{c}\Delta_x \vdash \hat{t}_x \Rightarrow \dot{t}_x : \dot{T}_x \,|_{\chi_x}\, C_x \quad \Delta_b \vdash \hat{t}_b \Rightarrow \dot{t}_b : \dot{T} \,|_{\chi_b}\, C_b \\ \chi_x \cap \chi_b = \varnothing \quad \chi = \chi_x \cup \chi_b \quad \Delta = \Delta_x \cdot (\Delta_b / x) \\ C = \bigwedge\{\mathit{valid}(\Delta(y)) \mid y \in \mathit{dom}(\Delta)\} \wedge C_x \wedge C_b \wedge \dot{T}_x <: \mathit{in}(\Delta_b(x))\end{array}}{\Delta \vdash \mathbf{let}\ x = \hat{t}_x\ \mathbf{in}\ \hat{t}_b \Rightarrow \mathbf{let}\ x = \dot{t}_x\ \mathbf{in}\ \dot{t}_b : \dot{T} \,|_{\chi}\, C}\ \textsc{TC\_LET}$$

$$\frac{\begin{array}{c}\Delta_1 \vdash \hat{t}_1 \Rightarrow \dot{t}_1 : \dot{T}_1 \,|_{\chi_1}\, C_1 \quad \Delta_2 \vdash \hat{t}_2 \Rightarrow \dot{t}_2 : \dot{T}_2 \,|_{\chi_2}\, C_2 \\ \chi_1 \cap \chi_2 = \varnothing \quad \chi = \chi_1 \cup \chi_2 \quad \Delta = \Delta_1 \cdot \Delta_2 \\ C = \bigwedge\{\mathit{valid}(\Delta(x)) \mid x \in \mathit{dom}(\Delta)\} \wedge C_1 \wedge C_2\end{array}}{\Delta \vdash \hat{t}_1; \hat{t}_2 \Rightarrow \dot{t}_1; \dot{t}_2 : \dot{T}_2 \,|_{\chi}\, C}\ \textsc{TC\_SEQ}$$

$$\frac{\begin{array}{c}\Delta_c \vdash \hat{t}_c \Rightarrow \dot{t}_c : \dot{T}_c \,|_{\chi_c}\, C_c \quad \Delta_t \vdash \hat{t}_t \Rightarrow \dot{t}_t : \dot{T}_t \,|_{\chi_t}\, C_t \quad \Delta_f \vdash \hat{t}_f \Rightarrow \dot{t}_f : \dot{T}_f \,|_{\chi_f}\, C_f \\ \chi_c \cap \chi_t \cap \chi_f = \varnothing \quad \chi = \chi_c \cup \chi_t \cup \chi_f \quad \Delta = \Delta_c \cdot (\Delta_t \mid \Delta_f) \\ C = \bigwedge\{\mathit{valid}(\Delta(x)) \mid x \in \mathit{dom}(\Delta)\} \wedge C_c \wedge C_t \wedge C_f \wedge \dot{T}_c <: \mathbf{Bool}\end{array}}{\Delta \vdash \mathbf{if}\ \hat{t}_c\ \mathbf{then}\ \hat{t}_t\ \mathbf{else}\ \hat{t}_f \Rightarrow \mathbf{if}\ \dot{t}_c\ \mathbf{then}\ \dot{t}_t\ \mathbf{else}\ \dot{t}_f : \dot{T}_t \sqcup \dot{T}_f \,|_{\chi}\, C}\ \textsc{TC\_IF}$$

$$\frac{\begin{array}{c}\Delta_c \vdash \hat{t}_c \Rightarrow \dot{t}_c : \dot{T}_c \,|_{\chi_c}\, C_c \quad \Delta_b \vdash \hat{t}_b \Rightarrow \dot{t}_b : \dot{T}_b \,|_{\chi_b}\, C_b \\ \chi_c \cap \chi_b = \varnothing \quad \chi = \chi_c \cup \chi_b \quad \Delta = \Delta_c \cdot (\Delta_b \cdot \Delta_c)\ast \\ C = \bigwedge\{\mathit{valid}(\Delta(x)) \mid x \in \mathit{dom}(\Delta)\} \wedge C_c \wedge C_b \wedge \dot{T}_c <: \mathbf{Bool}\end{array}}{\Delta \vdash \mathbf{while}\ \hat{t}_c\ \mathbf{do}\ \hat{t}_b \Rightarrow \mathbf{while}\ \dot{t}_c\ \mathbf{do}\ \dot{t}_b : \mathbf{Unit} \,|_{\chi}\, C}\ \textsc{TC-WHILE}$$

Figure 6.24: Constraint typing rules (Page 2 of 2)

The constraint typing derivations are algorithmic and compositional, through the use of effect combinators. Similar to the typing judgements for the type system of TS, we must choose to interpret function definitions as generating either flow or update effects for their parameters, and use either TC_CALL_FL or TC_CALL_UP to generate constraint typings for function calls. Similarly, with rule TC_FUN any effects that are extracted from the function body can be subsumed to be update effects if desired.

A typing scheme can be constructed directly from a constraint typing: for a constraint typing $\Delta \vdash \dot{t} : \dot{T} \mid_{\chi} \mathcal{C}$ where $\chi = \overline{\alpha_i}$, we may derive the typing scheme $\forall \overline{\alpha_i}[C].\,(in(\Delta), \dot{T}, out(\Delta))$.

As an example, let us consider again the term $f(x)$. By rule TC_CALL_FL, the constraint typing for this term is:

$$\begin{gathered}
\Delta \vdash f(x) : \alpha_3 \mid_{\chi} C \text{ where} \\
\Delta = f : (\alpha_1 \gg \alpha_2) \to \alpha_3,\ x : \alpha_4 \gg \mathit{remap}(\alpha_4, \alpha_1 \gg \alpha_2) \\
\chi = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \text{ and } C = \mathit{valid}(\alpha_1 \gg \alpha_2) \wedge \alpha_4 <: \alpha_1
\end{gathered}$$

One substitution which satisfies the constraint C is:

$$\sigma = \{\alpha_1 \mapsto \top, \alpha_2 \mapsto \top, \alpha_3 \mapsto \mathbf{Bool}, \alpha_4 \mapsto \mathbf{Unit}\}$$

From this, we may derive the typing:

$$\begin{gathered}
\Gamma \vartriangleright f(x) : \mathbf{Bool} \vartriangleleft \Gamma \\
\text{where } \Gamma = \{f : (\top \gg \top) \to \mathbf{Bool}, x : \mathbf{Unit}\}
\end{gathered}$$

Similarly, if rule TC_CALL_UP were used, we could derive the constraint typing:

$$\begin{gathered}
\Delta \vdash f(x) : \alpha_3 \mid_{\chi} C \text{ where} \\
\Delta = f : (\alpha_1 \ggg \alpha_2) \to \alpha_3,\ x : \alpha_4 \ggg \alpha_2 \\
\chi = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \text{ and } C = \alpha_4 <: \alpha_1
\end{gathered}$$

6.6.1 Correctness

The correctness property for constraint typings is as follows:

Theorem (Constraint typing correctness)**.** *Let* $\Delta \vdash \hat{t} \Rightarrow \dot{t} : \dot{T} \mid_{\chi} C$ *and* ρ *be a substitution such that* $\rho \vdash C$ *and* $\chi \subseteq \mathit{dom}(\rho)$*. It follows that* $in(\rho(\Delta)) \vartriangleright \rho(\dot{t}) : \rho(\dot{T}) \vartriangleleft out(\rho(\Delta))$.

This is proven as Theorem A.2.1, meaning all satisfying substitutions for typing schemes generated this way produce valid typings.

No attempt has yet been made to prove the completeness of constraint typing, which would be phrased as follows:

Hypothesis (Constraint typing completeness)**.** *Let* $\Gamma \rhd t : T \lhd \Gamma'$, $strip(t) = \hat{t}$ *and* $\Delta \vdash \hat{t} : \dot{t} \mid_{\dot{T}} \chi C$. *There exists a satisfying substitution* ρ *such that* $(\Gamma, T, \Gamma') \leq (in(\rho(\Delta)), \rho(\dot{T}), out(\rho(\Delta)))$.

6.6.2 Constraint typing simplification

Effect expressions are the main source of complexity in deciding whether a particular typing scheme can be satisfied. The production of a valid type for a type expression containing variables requires that a substitution ρ provides types that ensure type expressions such as $\dot{T} \sqcap \dot{U}$ or $remap(\dot{T}, \dot{U} \gg \dot{V})$ are defined after substitution. In analysing the structure of such type expressions, we may derive additional subtyping constraints over the type variables, that may allow an equivalent but simpler typing scheme to be derived.

Simplifying meet expressions

Meet type expressions are typically generated as a consequence of function type joins, where the effect input type is contravariant. Meet type expressions are conditionally defined, and therefore their satisfaction places some strong constraints on the satisfiability of the typing scheme.

Consider the expression $T \sqcap \alpha$. In order for a valid type to be generated from this expression, it must be the case that either $\alpha = \top$, or α is *structurally similar* to T.

Where $T = \mathbf{Bool}$ or $T = \mathbf{Unit}$, we may infer that $T <: \alpha$ as $\mathbf{Bool}$ and $\mathbf{Unit}$ do not have any subtypes. Consequently, $T \sqcap \alpha = T$ regardless of the choice of α.

If $T = (\overrightarrow{U_i \gg V_i}) \to W$, then for fresh $\overline{\alpha_i}$, $\overline{\alpha_i'}$ and α' we know that $(\overrightarrow{\alpha_i \gg \alpha_i'}) \to \alpha' <: \alpha$ such that $U_i \gg V_i \sqcup \alpha_i \gg \alpha_i'$ must be defined, and $W \sqcap \alpha'$ must be defined. This provides less direct information than in the case of $\mathbf{Bool}$ or $\mathbf{Unit}$, though further analysis of the meet expressions it generates for effect inputs and return type may allow for further simplifications.

If T is an object type, then little extra information can be extracted in a general way. If T allows a method call to m returning a value of type U, then if the substitution for α also allows a call to m returning a value of type V, then $U \sqcap V$ must also be defined.

In general, $O_1@\overline{S_1} \sqcap O_2@\overline{S_2}$ will require all methods along the shared *method sequences* of $O_1@\overline{S_1}$ and $O_2@\overline{S_2}$ to have return types for which a meet exists.

Definition 6.6.2 (Method sequences)**.** A *method sequence* is an interaction trace with the return types omitted. The set of method sequences $seq(T)$ of a type T are given by the following rules:

$$\frac{}{\epsilon \in seq(T)} \qquad \frac{s \in seq(T') \quad m : V \Rightarrow T' \in T}{m.s \in seq(T)}$$

The shared method sequences of two types T and T' are therefore given by $seq(T) \cap seq(T')$.

▲

There is no straightforward way to extract or represent all the additional sub-constraints for $O@\overline{S} \sqcap \alpha$ as the structure of α is unknown, therefore $seq(\alpha)$ is unknown.

Simplifying effect expressions

The predicate $\rho \vdash valid(\dot{E})$ requires that:

- All constituent type expressions produce valid types after substitution
- The effect expression can then be collapsed into a single effect of form $\rho(\dot{E}) = T \gg U$ or $T \ggg U$
- The resulting effect is valid: $valid(\rho(\dot{E}))$.

Effect combinator expressions can often be readily simplified, particularly where they do not contain *variable effects*, i.e. those of form $\alpha \gg \alpha'$ or $\alpha \ggg \alpha'$. Based upon the rules for effect combinators in Section 6.3.7, we can decide whether two effects can be composed based upon their structure. This will produce an effect of form $\dot{T} \gg \dot{U}$ or $\dot{T} \ggg \dot{U}$ which will often contain type expressions that provide further constraints by analysing any produced meet expressions.

Expressions involving update effects With an effect concatenation expression where at least one of the two effects is an update effect, we may simplify the effect expression and extract additional constraints in the following ways, regardless of whether either effect is a variable effect:

- $\dot{T} \ggg \dot{U} \cdot \dot{V} \ggg \dot{W} = \dot{T} \ggg \dot{W}$ when $\dot{U} <: \dot{V}$, otherwise it is undefined.
- $\dot{T} \ggg \dot{U} \cdot \dot{V} \gg \dot{W}$ may be simplified to $\dot{T} \ggg remap(\dot{U}, \dot{V} \gg \dot{W})$ when $\dot{U} <: \dot{V}$. It may be tempting to simplify this further to $\dot{T} \ggg \dot{W}$, as $remap(U, V \gg W) <: W$ by Lemma A.6.1, but this potentially loses type information.
- $\dot{T} \gg \dot{U} \cdot \dot{V} \ggg \dot{W}$ may be simplified to $\dot{T} \ggg \dot{W}$ when $\dot{U} <: \dot{V}$. Any extra information that would be derived by applying $\dot{T} \gg \dot{U}$ to some input type is irrelevant as it will be lost by the subsequent application of $\dot{V} \ggg \dot{W}$.

With an effect choice expression where at least one of the two effects is an update effect, we may perform the following simplifications:

- $\dot{T} \ggg \dot{U} \mid \dot{V} \ggg \dot{W} = (\dot{T} \sqcap \dot{V}) \ggg (\dot{U} \sqcup \dot{W})$. It may be possible to further simplify the meet expression on the input of this effect, with the generation of additional constraints.
- $\dot{T} \ggg \dot{U} \mid \dot{V} \gg \dot{W}$ (or the inverse, as choice is commutative) could be simplified to $(\dot{T} \sqcap \dot{V}) \ggg (\dot{U} \sqcup \dot{W})$ as in the previous case, but this may result in the loss of some type information when the effect is applied: given input type $T <: \dot{T} \sqcap \dot{V}$, $\mathit{remap}(T, \dot{T} \ggg \dot{U} \mid \dot{V} \gg \dot{W}) = \mathit{remap}(T, \dot{T} \ggg \dot{U}) \sqcup \mathit{remap}(T, \dot{V} \gg \dot{W})$, which is equivalent to $\dot{U} \sqcup \mathit{remap}(T, \dot{V} \gg \dot{W})$. This is guaranteed to be a subtype of $\dot{U} \sqcup \dot{W}$.

  As such, it is preferable to avoid directly simplifying effect choice expressions that mix update and flow effects. Where such a choice effect expression appears in a more complex expression involving concatenation, such as in $E_1 \cdot (E_2 \mid E_3)$, the expression can be rewritten as $(E_1 \cdot E_2) \mid (E_1 \cdot E_3)$. Where E_1 is an update effect, this will allow simplification without loss of type information, as $E_1 \cdot E_2$ and $E_1 \cdot E_3$ will simplify to update effects if defined.

  The same rewrite can also occur where the effect expression occurs on the left of the concatenation: $(E_1 \mid E_2) \cdot E_3 = (E_1 \cdot E_3) \mid (E_2 \cdot E_3)$.

The Kleene star of an update effect $\dot{T} \ggg \dot{U}$ is straightforward to simplify: $(\dot{T} \ggg \dot{U})* = \dot{T} \ggg (\dot{T} \sqcup \dot{U})$ when $\dot{U} <: \dot{T}$.

Flow effects With flow effect concatenation or choice, we may simplify effect expressions where one of the two effects is known to be a primitive flow effect. Let $\dot{T}$ be a primitive type expression. Concatenation may be simplified such that $\dot{T} \gg \dot{T} \cdot \dot{U} \gg \dot{V} = \dot{W} \gg \dot{W}$ where $\dot{W} = \dot{T} \sqcap \dot{U}$ when $\dot{U} = \dot{V}$. The same is true of choice: $\dot{T} \gg \dot{T} \mid \dot{U} \gg \dot{V} = \dot{W} \gg \dot{W}$. For Kleene star, $(\dot{T} \gg \dot{T})* = \dot{T} \gg \dot{T}$.

For the expressions $E \cdot \alpha \gg \alpha'$ and $E \mid \alpha \gg \alpha'$ where E is not a primitive flow effect, no simplification can be performed. Similarly, the kleene star of a variable flow effect $(\alpha \gg \alpha')*$ cannot be simplified.

6.6.3 Constraint solving

Once a typing scheme has been simplified as far as possible, it remains to determine whether the constraint of the typing scheme has a solution. Where the constraint is a conjunction of subtyping inequalities, and the types are structural, Simonet's Dalton solver [138] can be used to determine whether the constraint is satisfiable.

When update effects are used, the validity constraints for effects are all trivially satisfied and may be removed from the constraint, leaving only inequalities. Object types may be treated as *atomic* for the purposes of Dalton, with join and meet expressions involving object types resolved outwith the main algorithm. Where simplifying such join or meet expressions produces additional constraints, these can be added to the constraint set for the algorithm. Once no more constraints are generated in this manner, Dalton will indicate whether the constraint is satisfiable or not. If it is, the algorithm provides a type variable substitution which produces a minimal equivalent constraint, eliminating any variables which have only one type expression as a solution.

This approach has been implemented, and has been found to be successful in all cases tried, though no proof has yet been attempted to show that simplified typing schemes are equivalent to those generated, and that the use of Dalton with objects treated as atoms is sound.

Flow effect constraint solving

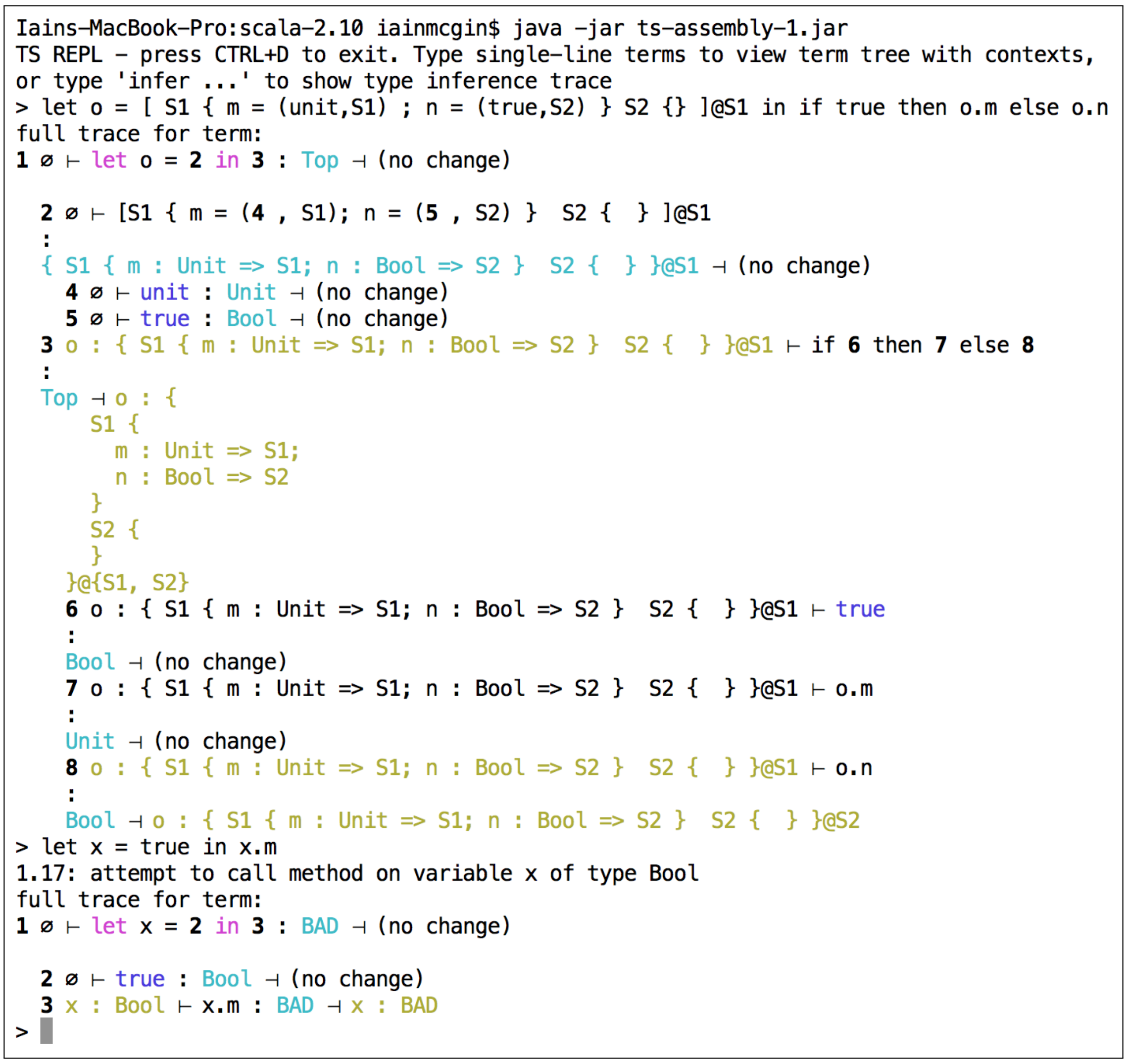
When flow effects are used, the type expressions generated are not structural, and therefore Dalton may not be reliably used: constraints such as $remap(\alpha_1, \alpha_2 \gg \alpha_3) <: \alpha_4$ can be generated when free function variables exist in the term. Such constraints cannot be structurally decomposed or simplified without a substitution for α_2 and α_3, and therefore Simonet's algorithm cannot be used.

The additional complexity that variable flow effects introduce poses a challenge for constraint solving which has not yet been studied in detail.

6.7 Implementation

Experimenting with implementing the type checker and various approaches to type inference proved to be very important to the process of understanding the problem space. The implementation is approximately 10000 lines of Scala code, half of which is a port of the Simonet's Dalton constraint solver [138] from ML to Scala, and a further quarter implements the necessary state machine operations: union, intersection, building simulation relations, and $remap$ for flow effects.

A read-eval-print-loop (REPL) was implemented using the Kiama library [139], into which a term can be entered and the state of the type checker can be printed in a legible manner, as shown in Figure 6.25. Each sub-term is given a unique numeric identifier, with the root term labelled `1`, and the typing for each term printed as



```
Iains-MacBook-Pro:scala-2.10 iainmcgin$ java -jar ts-assembly-1.jar
TS REPL - press CTRL+D to exit. Type single-line terms to view term tree with contexts,
or type 'infer ...' to show type inference trace
> let o = [ S1 { m = (unit,S1) ; n = (true,S2) } S2 {} ]@S1 in if true then o.m else o.n
full trace for term:
1 ∅ ⊢ let o = 2 in 3 : Top ⊣ (no change)

 2 ∅ ⊢ [S1 { m = (4 , S1); n = (5 , S2) }  S2 {  } ]@S1
 :
 { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S1 ⊣ (no change)
  4 ∅ ⊢ unit : Unit ⊣ (no change)
  5 ∅ ⊢ true : Bool ⊣ (no change)
 3 o : { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S1 ⊢ if 6 then 7 else 8
 :
 Top ⊣ o : {
   S1 {
    m : Unit => S1;
    n : Bool => S2
   }
   S2 {
   }
  }@{S1, S2}
  6 o : { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S1 ⊢ true
  :
  Bool ⊣ (no change)
  7 o : { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S1 ⊢ o.m
  :
  Unit ⊣ (no change)
  8 o : { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S1 ⊢ o.n
  :
  Bool ⊣ o : { S1 { m : Unit => S1; n : Bool => S2 }  S2 {  } }@S2
> let x = true in x.m
1.17: attempt to call method on variable x of type Bool
full trace for term:
1 ∅ ⊢ let x = 2 in 3 : BAD ⊣ (no change)

 2 ∅ ⊢ true : Bool ⊣ (no change)
 3 x : Bool ⊢ x.m : BAD ⊣ x : BAD
> █
```

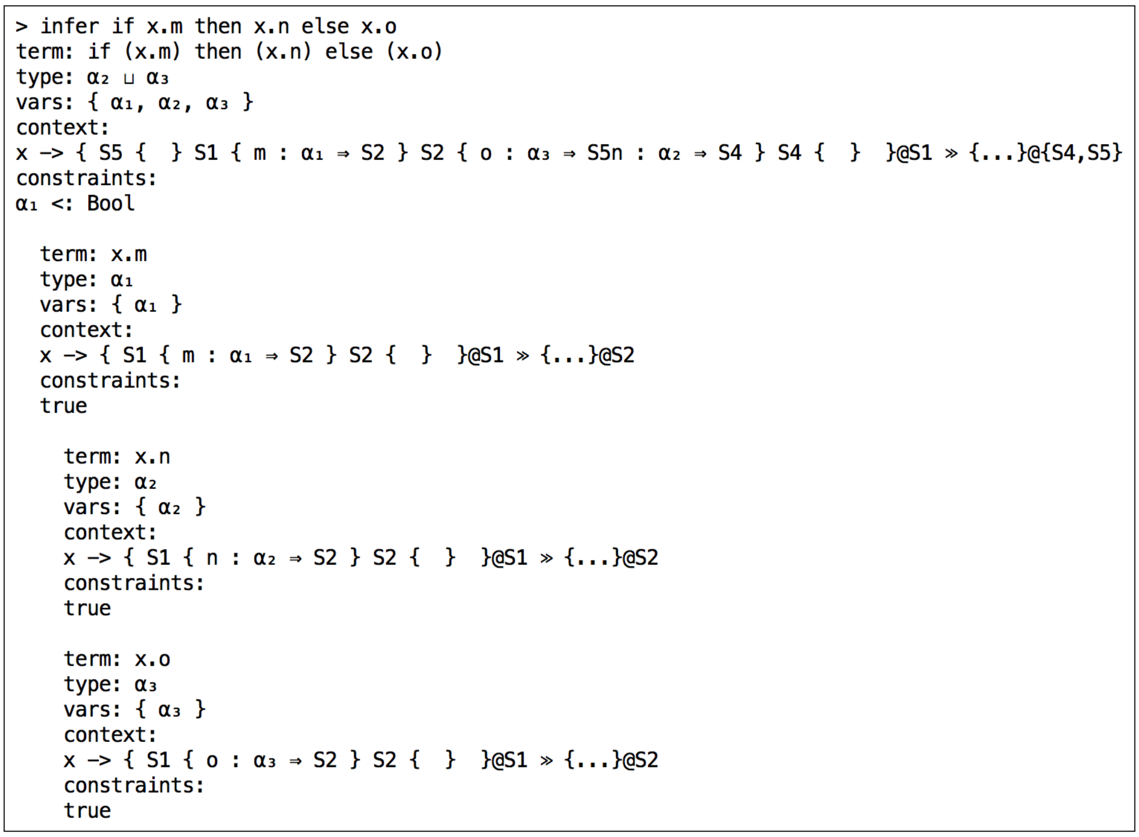
Figure 6.25: Example of type checking output in the TS implementation

---

$$\overline{x_i : T_i} \vdash t : U \dashv \overline{x_i : V_i}$$

The ⊢ and ⊣ symbols are used in place of ▷ and ◁ as they render more consistently in a terminal. Where the input context is unaltered in a typing of a term, the output context is rendered as no change to save printing redundant information. Where the type of a variable changes as part of a typing, the variable in each context is rendered in a different colour.

A trace of generated constraint typings can also be represented, as shown in Figure 6.26. This output shows the constraint typing generated for the root term, and the constraint typings for each sub-term from which this is generated. The constraint set of these constraint typings can then be passed to the port of the Dalton solver to check a solution exists.



```
> infer if x.m then x.n else x.o
term: if (x.m) then (x.n) else (x.o)
type: α2 ⊔ α3
vars: { α1, α2, α3 }
context:
x -> { S5 {  } S1 { m : α1 ⇒ S2 } S2 { o : α3 ⇒ S5n : α2 ⇒ S4 } S4 {  }  }@S1 » {...}@{S4,S5}
constraints:
α1 <: Bool

  term: x.m
  type: α1
  vars: { α1 }
  context:
  x -> { S1 { m : α1 ⇒ S2 } S2 {  }  }@S1 » {...}@S2
  constraints:
  true

    term: x.n
    type: α2
    vars: { α2 }
    context:
    x -> { S1 { n : α2 ⇒ S2 } S2 {  }  }@S1 » {...}@S2
    constraints:
    true

    term: x.o
    type: α3
    vars: { α3 }
    context:
    x -> { S1 { o : α3 ⇒ S2 } S2 {  }  }@S1 » {...}@S2
    constraints:
    true
```

Figure 6.26: Example of constraint typing generation in the TS implementation

6.8 Conclusion

This chapter has demonstrated a small, convenient type system in which to explore the inherent complexity of typestate and typestate inference. Flow effects represent an advancement of the state of the art, allowing more precise interpretation of function effects where we can be guaranteed that the identity and uniqueness of a parameter is preserved, and it appears that flow effects and update effects can co-exist in a type system, though this idea was not pursued any further in the type systems of TS, where either flow effects or update effects were used exclusively for function types.

The absence of principal types in the TS language, particularly where flow effects are used, demonstrates a fundamental limit to the representation of the requirements and effect of terms without type variables. Typing schemes serve this purpose better, and the algorithm presented for generating typing schemes is sound but not yet known to be complete. The constraint generated for such typing schemes where update effects are used can be solved using a structural subtyping solver, and the Scala implementation demonstrates that this approach is promising though not yet proven. Where flow effects are used, the constraints are not structural and therefore require a different approach to solving, which has not yet been investigated.

Overall, type inference for typestate languages does not appear to be impossible; indeed, significant progress has been made in the context of the TS language. While the constraint solving aspect of the algorithm is not proven, the implementation and lack of any specific counter-examples to the approach provide confidence that, with further work, the approach can be fully formalised and proven. It is not known how well the techniques presented in this chapter will generalise to a larger, more practical language, but the initial results are encouraging, and represent a useful contribution to the theory of typestate.

Chapter 7

Challenges for a typestate-oriented future

The expression and enforcement of typestate constraints are both complex topics, which are made more complex in the context of a legacy language's syntax and semantics. Attempting to provide a practical implementation of typestate in an existing language is difficult primarily due to the need to interact with legacy code, and working within the constraints of a type system which was not designed with typestate in mind.

Extracting the necessary information on aliasing requirements and effects in legacy code is intractable in a practical setting — at best, only partial information can be derived through expensive whole program analyses. As a compromise, dynamic checking can be used to enforce requirements on the boundaries between the legacy code and new code which is written with typestate in mind. Even with such a compromise, the lack of type inference, and the differences between subtyping with and without typestate, make it unlikely that any such hybrid system would be accepted as a practical solution which provides sufficient value.

With a clean slate, the essence of typestate can be more clearly articulated. This chapter attempts to explore the opportunities and challenges that remain in defining a full general purpose language with deeply integrated support for typestate and alias control. This constitutes a manifesto for such future research, with motivating examples throughout, rather than a formal treatment of a language as in Chapter 6. A hypothetical language *Chimera*, based upon the syntax of Scala, shall be partially defined in order to produce concrete examples, and the challenges for defining its semantics formally shall be explored.

7.1 Desirable features for a typestate oriented language

I believe a new typestate-oriented language should attempt to achieve the following goals:

- Be statically typed, with a type system that will not prove overly restrictive or frustrating to competent software engineers.
- Provide a lightweight syntax inspired by the best features of other successful, contemporary programming languages.
- Provide a coherent strategy for writing parallel code to exploit multi-core processors, and in general for writing distributed systems.

A language which could perhaps be augmented to satisfy these requirements is OCaml. The strict functional ML core provides a straightforward syntax and semantics which can be exploited to write concise, parallel code, while the object sub-language and reference types provide the scope to explore the challenges of typestate and alias permissions.

While OCaml is popular, it has failed to achieve the widespread popularity of languages like Java or C#. The Scala language, with a syntax that is more familiar to users of such languages and with easier integration with legacy Java/C#, is more popular for this reason and would also serve as a viable basis for a future typestate oriented language. The type system of Scala is however much more complex and less widely studied than that of OCaml (and other ML flavours).

I am unaware of any other languages which come closer to satisfying the requirements as specified, or which could provide a better foundation to work with. Given the additional challenge of addressing contemporary requirements for parallel and distributed systems, a new language which borrows heavily from the syntax and semantics of other successful languages may be the best approach to realising the goals of a typestate oriented language, much as Scala succeeded in its goals of producing a viable object-functional language by borrowing from both Java and Odersky's knowledge of functional languages.

I am hopeful that as the construction of parallel and distributed systems become a standard and essential skill set in industry, that more attention will be paid to the concept and benefits of typestate in such a setting, and as a result future languages will be invented or augmented to support it. Such languages may not be statically typed, and instead take the approach of Plaid where typestate is enforced dynamically to lower the impact of the type system's complexity.

Each of the specific requirements listed above shall be discussed in more detail below.

7.1.1 Type system

The complexity of the type system of a language is strongly tied to its practicality for programmers at a certain level of experience or education. Most complexity is unavoidable if certain features such as subtyping or parametric polymorphism are desired. Some complexity, however, derives from attempts to expose runtime performance concerns directly through the type system, resulting in special cases in the treatment of some terms or values in a language. Java's differentiation of primitive types such as integers or arrays from object types is an example of such a special case — this exists for performance reasons, as objects carry a significant overhead in memory usage compared to primitive types.

This early choice in the design of Java had consequences when parametric polymorphism was introduced. The designers of Java decided, for the purposes of backwards compatibility, that type parameters would be *erased* after the type checking of a program. As such, a consistent way of handling values of the erased type is required, which is problematic as primitive type fields and object references are handled differently. The decision was made therefore to only allow object types to be used for type parameters, disallowing a type such as `List<int>`. Instead, each primitive (or *unboxed*) type has a *boxed* alternative, which is its object type equivalent, and implicit conversion is provided between the boxed and unboxed forms. This "auto-boxing" and "auto-unboxing" has a runtime cost, for the creation of an object or an indirect field lookup respectively. Had the primitive types never existed, the compiler may have been able to make more intelligent decisions about how to handle this situation, cleanly separating the concerns of the type system from that of code optimisation.

Javascript provides a good case study for avoiding premature optimisation — the language has a relatively simple specification, and is not overly concerned with performance as it was designed to be easy to interpret, rather than easy to compile into efficient native code. Given the same program written in both C and Javascript, the execution time of the Javascript on a traditional interpreter compared to the C program compiled with a good optimising compiler was several orders of magnitude slower. However, good just-in-time compilers for Javascript which perform complex whole-program analyses are now standard and produce code that is roughly a factor of three slower than the optimised C program [52]. By comparison, Java programs running on the standard Oracle JVM are typically a factor of two slower than C.

Pure object oriented languages require all values to be objects, and treat such values consistently. Such languages are experiencing a resurgence in popularity, with Ruby and Scala as the most recent widely-used examples. Both languages inherit ideas from early pure OO languages such as Eiffel and Smalltalk. The consistent treatment of values simplifies the type system, and also means that built in and user-defined types are not fundamentally different.

Structural subtyping I believe that structural subtyping is preferable to nominal subtyping in an object oriented language. For one type to nominally be a subtype of another in a language such as Java, it must also be a structural subtype — it must have the same available methods, with parameters and return types fitting the necessary contra- and co-variant relationship to the parent, and so on. Nominal subtyping is typically required for performance reasons, as the explicit relationship allows the compiler to ensure that objects instances have the same memory layout for fast lookup of field and method pointer locations. Other languages such as Google Go [121] and OCaml offer structural subtyping without a significant performance hit.

As discussed in Chapter 6, a form of structural subtyping is possible in the presence of typestate and further reduces the burden to explicitly declare the relationship between types, as is required by the nominal subtyping in Java. Bounded parametric polymorphism for object interfaces is also clearly important to reuse and accurate specification, especially in a statically typed language.

No *null* references In Scala *null* references are supported as it was designed to interoperate with Java, but the use of *null* is strongly discouraged, in favour of *option types*. Null values can be convenient but are dangerous — null is a value which does not provide any of the guarantees that other values of the type do.

In new languages where backwards compatibility with previous poor language choices is not necessary, null should be avoided entirely in favour of real union or option types. In a typestate language, option types can be provided as a type with two states, `SOME` and `NONE`, where the `get()` method to extract the real value can only be called when the object is known to be in the `SOME` state. A hypothetical type expressed in a Hanoi-like syntax is shown in Listing 7.1, where `NONE` is represented by the root state.

With suitable language support, option types can be straightforward to work with and ensure that false assumptions about references are not made, with static enforcement. Where such static guarantees about freedom from null dereferencing can be provided, there is no need for a runtime check before every dereference as is required in languages like Java.

7.1.2 Lightweight syntax

Current popular object oriented languages have all made efforts to reduce the syntactic burden on the programmer. This is typically achieved in two ways: by providing more useful syntactic sugar for common tasks, and eliminating syntax that exists only for the benefit of the compiler.

```
type Option[T] {

  def isSome() : Bool
    :: true -> SOME
    :: false -> ROOT

  state SOME {
    def get() : T
  }
}
```

Listing 7.1: Option types in Chimera

Syntactic Sugar and DSLs

Over two decades of experience in working with object oriented code has led to the identification of common, low-level patterns. Providing direct support for expressing such patterns in a language reduces the amount that a programmer must type, and often clarifies the intent of the code, making the language more pleasant to work with. As an example, iterating over and manipulating data structures is a very common task, and was typically conducted in imperative code using loop constructs and local variables. An example of code with this structure is shown in Figure 7.1, where the function derives the average age of all males in a collection of people.

The Java code, written in a style which was common in Java 1.4, uses an iterator to visit and process each node. Java 1.5 introduced a specialisation of the for loop to make this pattern more compact: `for(Person p : list) { ... }`.

Python's list comprehensions go much further, providing a mechanism to simultaneously filter and transform a collection into another collection, as demonstrated in the definition of the `maleAges` variable.

The Language Integrated Query (LINQ) syntactic sugar in C# [93] is even more powerful, providing an SQL-like syntax for performing combined filter, map and reduce operations on collections of data, as shown in the definition of the `averageAgeByGender` variable. C# makes use of *higher order functions* as part of LINQ — the `Average` method on the group g is an example of this in the code above, as it takes a function literal which extracts the age from each `Person` object in a group.

The Scala code leverages higher order functions and a convenient syntax for function literals to filter, map and reduce the collection to derive the result.

The rigid syntax in most languages prevents the programmer from adding their own syntactic sugar for operations they perform frequently. Research into supporting the embedding of programmer-defined domain specific languages into general purpose languages has become a very active area. Scala aimed to demonstrate that higher order functions combined with a

```java
public Collection<String> extractEmptyStrings(Collection<String> strs) {
  ArrayList<String> nonEmptyStrs = new ArrayList<String>();
  Iterator<String> iter = strs.iterator();
  while(iter.hasNext()) {
    String str = iter.next();
    if(str.length() == 0) continue;
    nonEmptyStrs.add(str);
  }

  return nonEmptyStrs;
}
```

Listing 7.2: An example of redundant type annotations in Java

flexible syntax for method invocation could provide a means to support programmer-defined DSLs. This has allowed libraries to provide convenient mechanisms for data-parallelism and actor based concurrency, without any changes to the core language.

I believe that Scala's successes in this area, and the syntactic convenience that can be derived from them, indicate that higher order functions and closures which allow implicit capture are essential features for any new language.

Unnecessary syntax

Many languages require programmers to write code which is "obvious" in most situations. The simplest example of this is the semi-colon in imperative languages derived from C. In the vast majority of cases, the end of one statement and the beginning of another is completely obvious to the programmer. The semicolon exists for the sake of the parser, which often has only fixed look-ahead and no ability to backtrack and attempt alternative interpretations of a collection of tokens. This may have been justifiable in an era of limited memory and processing power, but with the several orders of magnitude increase in both over the last two decades this is no longer the case. Regardless, reasonable conventions involving the use of whitespace that match the behaviour of programmers in structuring their own code, as found in Python and Scala, work well in practice and do not require a complex parser.

A much more important class of unhelpful syntax, in some contexts, is that of type annotations. These are a common source of frustration for programmers in many languages — in Java, for instance, it is very common to have to write code in a manner where type information is repeated frequently, as shown in Listing 7.2.

The verbosity of Java is what has driven many programmers to other Java-compatible languages that do not require so many type annotations. Groovy is a prominent example of a dynamically typed language for the JVM, while Scala is perhaps the most popular statically typed alternative to Java on the JVM.

Java:

```java
public float getAverageMaleAge(List<Person> list) {
  Iterator<Person> iter = list.iterator();
  int totalMen = 0;
  int totalAge = 0;
  while(iter.hasNext()) {
    Person p = iter.next();
    if(p.male) {
      totalMen++;
      totalAge += p.age;
    }
  }

  if(totalMen == 0) return -1;
  return ((float)totalAge) / totalMen;
}
```

Python:

```python
def getAverageMaleAge(list):
  maleAges = [p.age for p in list if p.male]
  if maleAges:
    return sum(maleAges) / len(maleAges)
  else:
    return -1
```

C#:

```csharp
public float getAverageMaleAge(List<Person> list) {
  var averageAgeByGender =
    from p in list
    group p by p.isMale() into g
    select new {
      male = g.Key(),
      avg = g.Average(p => (float)p.getAge())
    };

  return averageAgeByGender.ToDictionary(x => x.male)[true]
}
```

Scala:

```scala
def getAverageMaleAge(people : Seq[Person]) : Double = {
    val maleAges = people.filter(_.male).map(_.age)
    if(maleAges.isEmpty) return -1
    return maleAges.reduce(_ + _) / (maleAges.length.toDouble)
}
```

Figure 7.1: Calculating the average age of all males in a collection, in various languages

Dynamically typed languages eliminate type annotations entirely at the expense of runtime overhead and practically no compile-time checking of whether code is correct. Gradual typing [136, 137] allows for some compile-time guarantees to be provided for code where type annotations are provided — the Plaid language relies upon this strategy. In a statically typed language, the only option to eliminate type annotations is to use type inference, if an appropriate algorithm exists for the type system. Scala provides a contemporary example of this, and uses a local type inference algorithm [120] to eliminate type annotations in the majority of cases where they would be required.

It is often possible to write code in Scala that is reminiscent of Python or Ruby in its lack of type annotations, with the additional static safety guarantees that Scala provides. As the discussion of type inference for the TS language in Chapter 6 entails, type inference may be feasible for a typestate-oriented language with a feature set approximating that of Java, or perhaps even Scala.

7.1.3 Parallel and distributed systems

Concurrency, parallelism and distributed systems generally should also be considered seriously in the design of any new language. More than just low-level primitives such as threads and atomic compare-and-swap operations are required; programmers require higher level concepts and tools in order to exploit the capabilities of many-core processors and large distributed systems.

Functional programming has become recognised as a possible means of abstracting data-parallel operations, through the pervasive use of parallel implementations of `map` and `reduce`. Scala provides data structures optimised for data-parallel operations. Such operations look identical to their sequential counterparts — they are simply different implementations of the same interface. These are convenient, and I believe any future language should include a similar mechanism for providing easy data-parallelism.

The Actor model for distributed systems has become popular as a result of the successes of the Erlang language, with implementations built around this idea appearing for many popular languages. The use of named *mailboxes* in the Actor model, where any actor with a handle to a mailbox can send a message to it, is more coarse than the duplex channels of π-calculus inspired languages. A channel, where the endpoints are linearly controlled, could allow for session-type-like protocol definitions. If the endpoints are permitted to be aliased, however, then a mailbox-like protocol where messages of a fixed type may only be sent in one direction can be permitted. Given that it should be possible to send a channel endpoint over a channel (classic *delegation* in session types), it should be possible to establish a two-party restricted channel *through* a mailbox. As such, in a language which integrates alias control and typestate, it should be possible to provide a system that is flexible, type safe, and offers the best of both the actor model and channels.

7.2 The Chimera language

Chimera is a hypothetical typestate-oriented language inspired by Scala which aims to serve the requirements outlined above, and provide context for the discussion of the theoretical challenges in formally defining such a language.

The syntax of Chimera will not be formally specified, but instead introduced by example amongst the discussion of the language's features. Some ambiguity is necessary in places where the theoretical underpinnings of the language are not fully defined or known to be sound.

The Chimera language is a pure object oriented language — all values are objects, and all variables are references to objects which are allocated on a heap. There is no `null` value in Chimera. Every object value has an associated object protocol and a current state. Every reference has an associated aliasing annotation which defines whether it is unique or not. Functions and methods on objects have effects on their parameters which are interpreted in the same manner as in the TS language. Subtyping in Chimera is behavioural, using the same definition based on traces as the TS language.

The object protocols in Chimera are hierarchical finite state machines in the same manner as Hanoi, where the root state is given the standard name `ROOT`.

7.2.1 Chimera types

The type of a reference in Chimera is a triple composed of the object protocol of the referenced value, the set of possible states that the value is guaranteed to be in, and the *permission* to the object through the reference.

It is my belief that the system of fractional permission types used in Plural and Plaid is overly complex. A simpler system where a reference can be *unique*, *shared* or *opaque* is likely to be sufficient. An opaque reference is one which points to *something*, but cannot be used in any way.

In this system, methods can be annotated with the level of permission that is required, much as in Plural. State changes are only permitted on methods which require a unique permission. Unique and shared references can be conceptually *split* into multiple shared references. This is effectively the alias control system used in Fugue.

I do not believe the addition of *full* and *pure* references or *immutable* references as in Plural adds sufficient value for the additional complexity it brings; if more types of permission were needed, then a programmer-defined mechanism inspired by Militão's views [96] would be preferable.

Let `Int` be the object protocol for integer values. As in most languages, we may dictate that integer values are immutable, therefore the protocol only has a single state: `ROOT`, and all methods on integers are defined in this state. If we have a unique reference to an integer value, then the type of this reference is written as $!Int@ROOT$ — the ! indicates that the reference is unique, while ~ indicates that the reference is shared, and $\#$ indicates a reference is opaque. This compact syntax is employed in an effort to reduce the amount of typing, and space, that type annotations will take when they are required.

Stating that an object is in the root state provides very little information — the methods available in the root state are available in all states. As such, Chimera adopts the convention that if the state set is not specified as part of a type annotation, then object is effectively in state $\{ROOT\}$. As such, ~*Int* is equivalent to ~$Int@ROOT$.

Methods must be annotated with their requirements — as integer values are intended to be immutable, all methods on `Int` would be annotated to indicate they can be invoked through a shared reference, while a state-changing method such as `next` on an `Iterator` would be annotated to require a unique reference.

Methods which can be invoked through a shared reference can of course be invoked through a unique reference, therefore the set of methods available in $O@\overline{S}$ through a shared reference is a subset of the methods available through a unique reference. Therefore, $!O@\overline{S}$ <: ~$O@\overline{S}$.

7.2.2 Methods and functions

Chimera supports object-functional programming in the style of Scala. Functions may be passed as parameters, be partially applied, and may implicitly capture references from the context in which they are defined. Functions are essentially objects with a single method, `apply(...)`, and have private fields for each implicitly captured reference. As such, all function calls in Chimera are in fact method calls, and all code is executed in the context of an object. Any object which defines an `apply` method may be considered to be a function.

Method definitions can be annotated with their *permission* requirement in the same way as reference types, with a symbol prefix to their name: `!next` would be a method which requires a unique permission, while `~getName()` is a method which only requires a shared reference. A convention is adopted in Chimera that if the permission annotation is not provided, then one of two defaults are adopted:

1. If the method does not change the state of the object, then the method will be treated as requiring a shared reference.

2. If the method may could the state of the object, then the method will be treated as requiring a unique reference.

Parameters on methods may be annotated to declare how they change the properties of an alias, in addition to any state change the method applies to the object value the method is associated with. A method can either *borrow* or *steal* a parameter reference. A borrowed reference may be used temporarily but it cannot be stored in a field of the object, or passed to any other method which may steal the reference. A stolen reference may be stored or passed to another method which steals the reference. In the latter case, the stolen reference may no longer be used in the original context.

For instance, consider the behaviour of a method which adds an element to a collection: the reference provided to the method is stored in the data structure. It follows that such a function must *steal* the reference it is provided. However, such a data structure may only require a *shared* reference. In the context where the method is called, the original reference can conceptually be *split* into two parts, and one part provided to method. The consequence of this is that if the original reference were unique, then it becomes shared; if the original reference were shared, then it remains shared. If however the function were specified to steal a unique reference, then the original reference becomes an *opaque* reference.

Function objects

A simple example of a class whose instances may be treated as a functions is shown in Listing 7.3. The `apply` method is defined to take a shared reference to an Integer i as a parameter, will not change the state of this parameter and will return a value of type $\mathbf{Unit}$. Additionally, the function promises not to make a copy of the reference to i through the `borrow` annotation on the parameter.

The body of `apply` demonstrates syntactic sugar for method invocation — the expression `acc + i` is an invocation of the method '+' on the `acc` field: `acc.+(i)`.

The full signature of `apply` has been specified in this example, but much of what has been written can be potentially inferred or be provided by implied defaults. Where the state of a parameter does not change, the effect may be omitted such that $t : T \gg T$ is equivalent to $t : T$. As the state of the parameter i does not change, we may specify it as `borrow i : ~Int`.

Finally, it may be inferred that the parameter i is borrowed by observing its usage — it is not stored in any fields, and not passed to any other functions which would copy the reference.

```
type Int {
  def +(borrow other : ~Int) : ~Int
  def -(borrow other : ~Int) : ~Int
  // ...
}

class Accumulator {
  private var acc : ~Int = 0

  def apply(borrow i : ~Int >> ~Int) : ~Unit = {
    acc = acc + i
  }

  def getTotal() : ~Int = acc
}

// ...

val acc = new Accumulator()
acc(1)
acc(2)
println(acc.getTotal())
```

Listing 7.3: Method definition in Chimera

Similarly, the requirements of the method + would have also provided the required type for i. As such, the signature of the method apply could simply have been `def apply(i) = ...`. For the purposes of documentation however, we may wish to preserve at least the type, permission and reference copying requirements for i.

7.2.3 Function literals

Function literals are syntactic sugar for creating objects with apply methods, and function types are similarly sugar for object types. Listing 7.4 shows some examples of function literals. A function literal defines the names of its parameters between brackets, optionally declaring the effect type for those parameters, and follows this with an expression body.

The function literal assigned to `returnsConstant` implicitly captures the reference x into the context of the body expression. For such a simple function and reference type this is of little consequence, though permitting this in general is a complex topic which is discussed in Section 7.3.3.

The function literal assigned to `closeIfOpen` takes a single parameter of an unspecified type. We may hope to be able to infer what the type of s may be in this situation, using similar techniques to those in the TS language. This may indeed be possible, though there are two rather different answers that may be derived depending on whether s is a unique or shared reference. If s is unique, then we may derive the following typing scheme:

```
val x = 1
val returnsConstant = () => x

val closeIfOpen = (s) => if(s.isOpen()) s.close() else unit

// invalid
val addBroken = (x, y) => x.add(y)

// addWorking : [T](O@S >> O@S', Int) -> T
// where O = { S { add : Int -> T => S' } S' {} }
val addWorking = (x, ~y : ~Int) => x.add(y)

val factorial = (x) => {
  var tot = 1
  var last = x
  while(last > 1) {
    tot = tot * last
    last = last - 1
  }

  tot
}
```

Listing 7.4: Example function literals

$$\mathit{closeIfOpen} : \forall\alpha.(!O@S_1 \gg !O@\{S_2, S_3\}) \to (\alpha \sqcup \mathbf{Unit}) \text{ where}$$
$$O = \{S_1\{\mathit{isOpen} : () \to \mathbf{Bool} \Rightarrow S_2\}S_2\{\mathit{close} : () \to \alpha \Rightarrow S_3\}S_3\{\}\}$$

However, if s is a shared reference, then s cannot change state. In this case, we may derive the following typing scheme:

$$\mathit{closeIfOpen} : \forall\alpha.\, {\sim}O \to (\alpha \sqcup \mathbf{Unit}) \text{ where}$$
$$O = \{\mathit{isOpen} : () \to \mathbf{Bool} \Rightarrow \mathit{ROOT};\, \mathit{close} : () \to \alpha \Rightarrow \mathit{ROOT}\}$$

Given any choice for the type α, the type of closeIfOpen where s is unique is always a subtype of the case where s is shared. A legitimate choice may be to always assume a reference is shared if this is not specified, as a unique reference can always be passed when a shared reference is requested. However the types place such radically different requirements on s that it would seem prudent to require the programmer to specify which of the two they desire explicitly, through a type annotation.

The function literal assigned to addBroken takes two parameters, x and y, and invokes the method add on the x, using y as an argument. Inferring the type of such a trivial function is not straightforward. It is clear that x is an object which must have the method add available in the current state, and that y must be a subtype of the required type of the first parameter to x. As is the case with functions in the TS language, insufficient information is provided to decide what the effect type of the method add should be. Furthermore, it is also unclear whether `add` is likely to copy the reference to y, or simply use it temporarily.

If, however, y was defined to be a *shared* reference which is *borrowed*, this would provide sufficient information to know that `add` may not change the state of y, eliminating the ambiguity in the possible effects that `add` may have. This is shown in `addWorking`, where an additional type annotation on y provides sufficient information to infer all that we need to know about x.

7.2.4 Defining object types and classes

The state-oriented syntactic structure of Hanoi can be adapted to provide more than just the specification of typestate constraints — implementations can be provided within the structure. In Chimera, state labels for an object type are fully capitalised, for example `EMPTY` or `CAN_READ`.

Consider the `Iterator` type defined and implemented in Listing 7.5. A *type declaration* gives a name to an object protocol, and is effectively an interface specification. Both fields and methods may be defined on types. A *class* declaration provides an implementation of an ad-hoc type. An object instance produced from a class' constructor has an object protocol derived from the class' definition and a state as defined on the constructor signature. An object value may be considered to be of any type which it is compatible with.

The type `Iterator[T]` specifies the interface for iterator-like objects, with four states in a hierarchy: `ROOT`, `NEXT_AVAIL`, `MIDDLE` and `CAN_REMOVE`. Conditional transitions can be specified using essentially the same syntax as Hanoi, but as annotations directly on the declaration of a method. Such method declarations could be overridden in natural ways, such as in the case of `hasNext` on Line 13, where the return type and irrelevant transitions need not be respecified.

More interestingly, methods are declared in the scope of the state they should exist in, and method implementations can be overridden in child states. This design should provide a very intuitive relationship between states and their available methods — methods that cannot be called simply do not exist in the state in question, unlike in Hanoi where the methods do exist at all times on the object, but are enabled and disabled in accordance with the typestate constraints. It is also conceivable that states could have their own fields, or override parent fields in a similar manner.

It is not necessary to explicitly specify that `ArrayIterator` is an `Iterator`, or the mapping between the state labels. The type system can derive on demand that `ArrayIterator[T]@N` is a subtype of `Iterator[T]@NEXT_AVAIL`.

```
type Iterator[+T] {
  def hasNext() : Bool
    :: true -> NEXT_AVAIL
    :: false -> <self>

  state NEXT_AVAIL {
    def next() : T -> CAN_REMOVE
    state MIDDLE { def remove() : Unit -> NEXT_AVAIL }
  }

  state CAN_REMOVE {
    def remove() : Unit -> ROOT
    def hasNext() :: true -> MIDDLE
  }
}

class ArrayIterator[+T] {
  private var values : Array[T]
  private var position : Int

  new(arr) -> ROOT = {
    this.values = arr
    this.position = 0
  }

  def hasNext()
    :: true -> N
    :: false -> <self>
    = position < (values.size - 1)

  state N {
    def next() -> ROOT = {
      val nextVal = values.get(position)
      position += 1
      return nextVal
    }

    state M {
      def remove() -> N = values.removeIndex(position - 1)
    }
  }

  state R {
    def hasNext() :: true -> M
    def remove() -> ROOT = values.removeIndex(position - 1)
  }
}
```

Listing 7.5: A Hanoi-like syntax for specifying and implementing an iterator

7.2.5 Type parameters

As the `Iterator` example demonstrates, both types and classes can be parameterised by other reference types, and a *variance* annotation is included. $+T$ indicates the type parameter is *covariant*, $-T$ indicates it is *contravariant*, and no annotation indicates it is *invariant*. Let $U' <: U$, and $\neg(U <: U')$. Then the following table demonstrates the subtyping relationship between $T[U]$ and $T[U']$:

Type parameter	$T[U'] <: T[U]$	$T[U] <: T[U']$
+	Yes	No
−	No	Yes
None	No	No

The reference type used as a parameter *includes* the permission to the type, therefore $T[!O@\overline{S}]$ and $T[\sim O@\overline{S}]$ are different types. This is useful as it allows for a collection of unique references to be differentiated from a collection of shared references, which enables some interesting possibilities for *mutable collections*, discussed in Section 7.3.1. If a specifically unique or reference type is required, the parameter may be specified with the necessary prefix: `type O[!T]`.

It is also likely that methods may wish to return parameterised types with the aliasing annotation modified, such as returning a shared reference to a value stored in a data structure: `def get(key : K) : ~V`. While it would be possible for a collection with type parameter `~T` to specify that a method returns a `!T`, implementing such a method would require some means of generating a `!T` which is not always possible.

One scenario where this may be possible is if the type parameter `T` provides a method which returns a `!T`, such as through a `clone()` method which creates a deep copy of the value. In order for a type to require that values of a specified type parameter offers such a method, a *type bound* would be required: `type O[T <: ~Cloneable[T]]`. The ability to specify such type bounds is standard in Java and Scala, and Chimera would require such a feature to satisfy the modelling needs of programmers.

7.2.6 Public fields and global objects

Allowing public mutable fields on an object is not generally considered good design, as it breaches the encapsulation of the object. It is particularly problematic in a language with alias control, as a public field is implicitly shared by all contexts that have access to the field's containing object. Public immutable fields, in contrast, are relatively harmless and are effectively constants.

```
type Person {
  def name : ~String
  def age : ~Int
  def age_= : ~Int
}

def makeOlder(p : ~Person) : ~Unit = { p.age = p.age + 1 }
def isAdult(p : ~Person) : ~Bool = p.age > 18
```

Listing 7.6: Example of virtual fields

In Scala, a package is effectively an object, which allows other types, objects and functions (which are actually fields and methods of the *package object*) to be defined without the need for any special semantics. Packages are *singleton* objects, and Scala also provides a convenient syntax for defining singleton objects generally. Singletons are both architecturally useful and dangerous, and unarguably common in real software systems.

As such, it is important that Chimera has a sound strategy for providing singletons, and it is likely that in order to provide pure object semantics that any package system for Chimera would need to be based on singleton objects, as it is in Scala.

Public fields could be disallowed in Chimera in favour of *properties*, which are a pair of *get* and *set* methods. Syntactic sugar for properties is provided in Scala, which allows methods to be declared which take no parameters, and do not require parentheses to be invoked. C# provides similar syntactic sugar.

Chimera could adopt Scala's convention for properties. Consider the following example in Listing 7.6, which shows a `Person` type with two properties: `name` and `age`. The `age` property allows overwriting through the method `age_=`, which is Scala's convention for defining a method which writes to a property, while the `name` property is immutable. Both `name` and `age` can be used as though they were fields in the methods `makeOlder` and `isAdult`.

Though in principle reading or writing a property could change the state of an object, in order to avoid confusion it would be safer to insist that property methods cannot change the state of the object, such that their semantics are as close to that of real fields as possible. Regardless, properties on shared objects such as packages could only be accessed if the property methods do not change the state of the object.

7.2.7 Field overriding

One interesting possibility for Chimera would be to allow the set of fields, and the type properties of these fields, to vary by state just as methods do. The main source of complexity in allowing this is in defining how the fields of a state should be initialised as part of a state

```
class MutableOption[T] {

  new() -> ROOT = {}
  new(steal t : T) -> SOME(t) = { }

  def isSome() : ~Bool = false
  def set(steal t : T) : Unit -> SOME = { }

  state SOME {
    private val t : T

    entry(steal t : T) = {
      this.t = T
    }

    def isSome() = true
    def get() : T = t
    def clear() -> ROOT = { }
  }
}
```

Listing 7.7: Example of state entry methods

transition. One possibility would be to define *entry* methods for each state, which would be responsible for correctly initialising all of the fields related to a state. The entry method for a state is invoked *after* the evaluation of a method that declares a transition. If the entry method takes parameters, then these parameters must be specified as part of the transition declaration, and may be taken from the set of variables in the *output context* of the body of the function. The entry methods cannot be directly invoked — this is to avoid confusion over allowing an object to temporarily transition between states during a method call.

An example of this is shown in Listing 7.7, which demonstrates an implementation of a mutable `Option` type. The `SOME` state contains a field for storing the value of the option type; this field could not be declared in the root of the object as the field must have a value, and `null` does not exist in the language (see Section 7.1.1). The methods `set` and `clear` transition the instance between the `ROOT` and `SOME` states, with the entry method of `SOME` simply assigning a provided value of type T into the field.

A likely scenario is that common fields will be defined in a parent state and that child states will wish to refine the type information of their parent's fields. The example in Listing 7.8 demonstrates with an implementation of an iterator-like stream for reading from a random-access binary file handle. The `ROOT` state of `FileStream` wraps a potentially closed file handle, while in the `OPEN` and `READABLE` states the file handle is known to definitely be open. The methods `reopen`, `close` and `next` change the state of the `FileStream` instance.

The `reopen` method is potentially controversial — the type system would need to be sensitive to the control flow in the method in order to determine that when the method returns true the file instance is in fact open.

In general, a child state must override the type of a parent's field with a subtype. Allowing arbitrary overriding of a field's type would violate the invariants that inherited methods would expect — `reopen` expects the `file` variable to be at least a `!FileHandle@ROOT`, and changing this to a `!String` in a child state would result in undefined behaviour when this method is executed.

7.2.8 Inheritance

Inheritance and subtyping are orthogonal features of object oriented languages; we may define subtyping structurally, and allow inheritance purely for code reuse. Java avoids the issue of *diamonds* [87] (also known as *fork-join inheritance*) by only permitting single inheritance, while Scala uses linearised mix-in inheritance. In both, subtyping is nominal. Google Go, which uses structural subtyping, provides no mechanism for inheritance at all. Go mandates the use of *composition* for code reuse rather than inheritance, which is the recommended approach in Gamma *et al.* 's seminal work on design patterns [54].

It is unclear what inheritance option would work well for Chimera. If a mix-in based approach were adopted, and mix-ins were allowed to define their own state machines, then it is unclear how the state machines of each mix-in should be composed. If mix-ins were restricted to be stateless, then each state of class could be permitted to select its own set of mix-ins. Without further investigation, it is unclear if this is particularly useful.

Chimera could forego inheritance altogether as Go has; compositional reuse of code can work well in practice, and would avoid adding additional complexity to what is already likely to be a very complex language. However, single inheritance may be a viable option, which would provide some means of implicit reuse. Inheriting from another class would also inherit its state hierarchy. An abstract example of this is shown in Listing 7.9. The class `B` extends `A`, and adds a new state `Y` which is a sub-state of `X`. The method `x()` is overridden in `X` and `Y`. This is not conceptually difficult, and the overriding rules for transitions adopted from Hanoi ensure that `B@X` is still a subtype of `A@X`, as is `B@Y`.

7.2.9 Dynamic state tests

As the typestate of an object in a language such as Java is virtual and only discernible through *dynamic state test methods* such as `hasNext`, it was necessary to model conditional transitions in Hanoi and for an analysis to be sensitive to the return values of methods. In a new language with the freedom to define any necessary constructs, it may be better to provide

```
type FileHandle {
  def isOpen() : ~Bool
    :: true -> OPEN
    :: false -> ROOT

  def close() : ~Bool -> ROOT

  state OPEN {
    def length() : ~Int
    def read(index : Int) : ~Int
  }
}

class FileStream {
  private val file : !FileHandle@ROOT

  new(file : !FileHandle@ROOT) -> ROOT(file) = {}
  new(file : !FileHandle@OPEN) -> READABLE(file) = {}
  entry(steal file : !FileHandle@ROOT) = { this.file = file }

  def reopen() : ~Bool
      :: true -> OPEN(file)
      :: false -> ROOT(file) = {
    if(file.isOpen()) {
      file.rewind()
      return true
    }

    return false
  }

  def close() : Bool -> ROOT = file.close()

  state OPEN {
    private val file : !FileHandle@OPEN
    private var index : ~Int

    entry(file : FileHandle@OPEN) = {
      this.file = file
      this.index = 0
    }

    def hasNext() : ~Bool = index < file.length()

    state READABLE {
      def next() : Int -> OPEN(file) = {
        val nextVal = file.read(index)
        index += 1
        return nextVal
      }
    }
  }
}
```

Listing 7.8: An example of refining the type of a parent state's field

```

class A {
  def a() : !A = new A()

  state X {
    def x() : ~Int = 0
  }
}

class B extends A {
  state X {

    def x() -> Y : ~Int = 1

    state Y {
      def x() -> X = 0
      def y() -> X : ~String = "hello"
    }
  }
}
```

Listing 7.9: An example of single inheritance in Chimera

---

dynamic state testing as a fundamental operation, akin to pattern matching in functional languages, or an equivalent of the `instanceof` operator in Java. The `hasNext` operation on `Iterator` could be removed and instead code iteration could be written in the recursive form shown in Listing 7.10.

While this may be more convenient in some scenarios, it is likely to have at an unpredictable runtime cost due to structural subtyping, with some unintuitive properties. A type which is a subtype of `!Iterator[~Int]@ROOT` can be passed into `sum` and a simulation relation must be built to determine which of the `Iterator` states it could be considered to be in. There may be more than one state which is a simulation of any given state in `Iterator`, and therefore the cases in the `stateOf` construct must be considered in order and the first match accepted. This has the consequence that if the cases in `sum` were reversed, the method would always return 0, as `ROOT` has no methods and all object types are a simulation of an empty state.

7.2.10 Processes and channels

Synchronous duplex directional channels such as those found in the π-calculus [97] are a good fit for a language like Chimera — a program consists of one or more sequential processes which communicate via channels with protocols. The two endpoints of a channel can be represented by objects that exist in the context of each process.

Session types provide a foundation for the theory of typed communication over such chan-

```
type Iterator[+T] {
  state HAS_NEXT {
    next() : T -> ROOT
  }
}

def sum(iter : !Iterator[~Int]@ROOT) : ~Int = {
  stateOf(iter) {
    case HAS_NEXT => {
      val i = iter.next()
      return i + sum(iter)
    }
    case ROOT => 0
  }
}
```

Listing 7.10: Iterator-like behaviour without the need for a `hasNext` method or conditional transitions

---

nels. Consider the session type in Figure 7.2 which defines the protocol that an automated teller machine (ATM) may have with a bank. The card details and the pin entered by the user are first sent, to which the bank can respond with either `welcome` or `error` (the curly braces represent external choice), with a string describing the problem (incorrect details, unknown account, etc.) in the latter case. If the login is successful, then the user can request the account balance or withdraw some funds (with angle brackets representing internal choice). Withdrawing funds can fail in a similar manner to the login procedure if an attempt is made to withdraw more funds than are available, but this error does not terminate the session — this must be done explicitly by sending a logout request.

This protocol can be represented using a typestate-constrained object in a straightforward manner, and a possible encoding of this in Chimera is also shown in Figure 7.2. The `sendCredentials` method takes the card details and pin as parameters, and the return value indicates the success or failure of this request. Scala-like case classes are used like the possible values of an enumeration, but with the added benefit that they can carry the data that is associated with outcome easily. The methods available in the `ACTIVE` state are equivalent to the labelled choices that are available in the session type at this point.

A channel's definition is independent of whether the processes exist on the same machine, or on different machines across a network. Channels are inherently unreliable — the protocol defines what *should* happen, but failures can occur for a number of reasons:

- The receiving process has crashed, due to an unhandled exception of some form, and therefore would never respond to the message.
- The transport used to implement the channel has failed, e.g. a TCP connection which could not be re-established within a reasonable period of time.

Session type:

```
LOGIN = !String . !String . {
  welcome : ACTIVE,
  error : ?String . exit
}
ACTIVE = <
  balance : ?Int . ACTIVE,
  withdraw : !Int . {
    ok : ACTIVE,
    error : ?String . ACTIVE
  }
  logout : exit
>
```

Chimera Channel type:

```
sealed abstract class LoginResponse
case object LoginWelcome extends LoginResponse
case class LoginFailure(msg : ~String) extends LoginResponse

sealed abstract class WithdrawResponse
case object WithdrawOK extends WithdrawResponse
case class WithdrawError(msg : ~String) extends WithdrawResponse

type BankChannel {
  state LOGIN {
    def sendCredentials(cardInfo : ~String, pin : ~String)
      : LoginResponse
      :: LoginWelcome -> ACTIVE
      :: LoginFailure -> END
  }

  state ACTIVE {
    def requestBalance() : ~Int
    def withdraw(amount : ~Int) : WithdrawResponse -> ACTIVE
    def logout() : Unit -> END
  }

  state END {}
}
```

Figure 7.2: The client session type for a simple ATM, with an analogous Chimera channel type

- The remote process has gone rogue, sending invalid data for the current expected state of the channel.

All of these exceptional circumstances are best represented as thrown exceptions, which the process can handle and attempt to recover from if possible (i.e. attempt to set up a new channel, potentially to a replacement process). Whether those exceptions should result in a state change in the object is unclear — if a state change were specified, then the alias control semantics of Chimera would require a unique reference be used to interact with the channel. This may prove overly restrictive in practice; there may be some situations where a protocol is desired that is effectively stateless, which would allow more than one component in a process is able to use the channel. If exceptions do not trigger state transitions, then the channel's state will not be a faithfully represented by the protocol — an exception is thrown to indicate the channel is broken, but the state of the object would still allow methods to be called.

Some careful consideration of the memory model for Chimera is also required. The simplest option would be to require that each Chimera process exist in a separate memory space — no shared data would exist between processes. Passing data over a channel, when both processes are local, could be implemented as a deep copy of the object and a passing of a reference to the copy. If the type were in fact immutable, such as an `~Int`, then the object need not even be copied. Analysis of user-defined types to determine when they are effectively immutable in this manner could be used to allow this optimisation for arbitrary types.

7.3 Theoretical challenges and opportunities

Formalising a language like Chimera poses significant theoretical challenges, as it mixes structural subtyping, parametric polymorphism, alias control and typestate. Attempts have been made to provide a formal core for the Scala language [5,112], though much of the complexity in such formalisms is as a result of *path-dependent types*, which allow a restricted form of dependent typing and are necessary for mix-in inheritance and parametric polymorphism. As I am not proposing that Chimera attempt to provide mix-in inheritance, such a formalism may be a poor choice as a basis for the definition of Chimera.

It may be possible to formalise a language such as Chimera using Abadi and Cardelli's seminal work [1] as a foundation.

Beyond this, there are many interesting things that one could potentially do in a language with deeply integrated support for aliasing and typestate, and also some cases which will require careful consideration in the theoretical work to retain soundness; each of these are covered separately below.

7.3.1 Mutable collections

The ability to determine whether a reference is unique or not offers the interesting possibility of allowing the *type parameter* of an object to change in response to a method call, in addition to the object's state. This is particularly interesting in the case of collection types — a higher order function like `map` could be permitted to change the type parameter. An example of this for a mutable linked list node is shown in Listing 7.11. Here, the `map` function can take a function which mutates the element type to some new type U, and consequently changes the type parameter of the node to U. This must be performed along the entire chain, as shown in the override of `map` for the `PREFIX` state.

In order for this to occur, a unique reference is required to the node type in order to allow the type parameter to be safely changed. This raises the question as to whether we also need to specify a separate `map` function for when $f : T => T$ — if the function does not change the type parameter, then map can be permitted through a shared reference as it does not change the type parameter.

The ability to do either offers the possibility of efficient operations on data structures which do not require a deep copies, much as linear types in functional languages permit. Of course, deep copy based implementations can be permitted, but they would at least be *optional*.

Allowing the type parameter to change does not appear to be any different to allowing the state to change — both require uniqueness, and would likely be handled in the same way in the type system. Nonetheless, further investigation is required to determine whether any unanticipated complexities exist in allowing this to occur.

7.3.2 Borrow and steal for return values

The *borrow* and *steal* annotations on parameters are an essential part of making the alias control system in Chimera practical — shared references are very restrictive and without the ability to borrow unique references, they can never leave the local scope of a method or object, even temporarily.

Return values pose a challenge in a similar manner. Consider a Chimera type which represents an array:

```
type Array[T] {
  def set(index : ~Int, t : T) : Unit
  def get(index : ~Int) : T
  def length() : ~Int
}
```

```
class Node[T] {
  private var t : T

  new(t : T) = { this.t = t }
  entry(t : T) = { this.t = t }

  def !map[U](f : T => U) : Unit -> ROOT[U](u) = {
    val u = f(t)
  }

  def !prefix(tail : !Node[T]) -> PREFIX(t, tail) = { }

  state PREFIX {
    private var tail : !Node[T]

    entry(head : T, tail : !Node[T]) {
      super(head)
      this.tail = tail
    }

    def !map(f : T => U) -> PREFIX[U](u, uTail) = {
      val u = f(u)
      val uTail = tail.map(f)
    }
  }
}
```

Listing 7.11: A mutable linked list

An implementation of `Array` will store a value of type T for each valid index. The return values of `get` pose a problem — if we have an instance of `Array[!String]`, if the value returned by `get` is stored in another field or passed to a function which steals it, then the invariant of the array instance will have been violated.

As such, different semantics for `get` are required dependent upon whether the returned value is borrowed or stolen. In the case of an `!Array[!String]` where a return value is stolen, the array would effectively become an `!Array[~String]` or `!Array[#String]`, depending on whether the the full permission is stolen or not. It is unclear whether such an automatic transformation of type parameters could realistically be achieved, or is even desirable due to the confusion it may cause.

There is also the issue of whether state change should be permitted on the borrowed reference: if we have an instance of `!Array[!Stack[~Int]@NOT_EMPTY]`, retrieve one of the references in the array and invoke `pop()` which would change the state of the reference. It is unclear whether this state change could be reintegrated into the type parameter of the array instance, or when this should occur. Consider the example in Listing 7.12, which demonstrates this in a function which pops the top element off of the first stack, and the pushes this onto all of the stacks. The `firstStack` reference ceases to be relevant after the call to `pop` on Line 4. After this call, the stack would be in state `ROOT` as it may

```
 1  def copyFirstToAll(borrow a : !Array[!Stack[~Int]@NOT_EMPTY]) : Unit = {
 2    if(a.length() < 1) return;
 3    val firstStack = a.get(0)
 4    val topElem = firstStack.pop()
 5    var i = 0
 6    while(i < a.length()) {
 7      val stack = a.get(i);
 8      stack.push(topElem);
 9      i = i + 1;
10    }
11  }
```

Listing 7.12: Temporary state change of borrowed return values

---

be potentially empty. The best type we could assign to the array at this point would be `!Array[!Stack[~Int]@ROOT]`.

After this, the calls to `push` on each stack on Line 7 would not change the type of the array, as changing the state of one stack from `ROOT` to `NOT_EMPTY` does not change the upper bound on the state of the stacks which the type parameter represents. This may be disappointing, as to a programmer it is clear that `push` will be invoked on every element of the array and therefore one may reasonably expect that the type of the array should be `!Array[!Stack[~Int]@NOT_EMPTY]` again after the completion of the loop. Such reasoning is unlikely to be possible in general.

As such, the type on the parameter a is actually invalid, as the type parameter on a is not accurate.

An additional problem exists in the code: the variables `firstStack` and `stack` are aliases of one another during the first iteration of the loop, but it is unlikely that the type system would be able to derive this association. As such, if the `firstStack` variable were to be used again during or after the loop, it would be possible for the two references to have an inconsistent view of the state of the same object, as both references would be believed to be unique.

Given the above difficulties, the only sound option may be to require that return values are explicitly annotated as offering a borrowed or stolen reference, and that the state of borrowed references cannot be changed. In the case of `Array`, variants of the `get` method could be provided to cater for each possible scenario:

```
def peek(borrow index : ~Int) : borrow T
def share(borrow index : ~Int) : steal ~T -> ROOT[~T]
def take(borrow index : ~Int) : steal T -> ROOT[#T]
```

The `peek` method would allow the client to use the value from a given index temporarily, and in a manner which does not change its state. The `share` method copies the reference in the array such that it will become shared, resulting in a change in the type parameter of the

```
def countMinors(borrow people : !Array[!Person]) : ~Int = {
  var i = 0
  var minorCount = 0
  while(i < people.length()) {
    if(people.peek(i).getAge() < ADULT_AGE) {
      minorCount += 1
    }
  }

  return minorCount
}
```

Listing 7.13: An example of using peek

array. The `take` method consumes the reference in the array entirely, which would allow a unique reference to be extracted from an array in a permanent fashion. Both `share` and `take` would require a unique reference to be invoked, as they change the type of the array.

The `share` method is of little consequence if the type T is already a shared reference type, and this is perhaps the mostly likely scenario for the use of such a data structure. The `peek` method is particularly useful for collections of unique references, where a value is extracted only for temporary use, such as in the `countMinors` method shown in Listing 7.13.

The `take` method is much more destructive and unlikely to be useful in general, as taking the value at a single index requires that all other indices be treated as opaque references, making them useless. This is potentially symptomatic of a poor design — an alternative data structure with potentially better characteristics is a partial map:

```
type PartialMap[K, V] {
  def swap(steal k : K, steal v : V) : steal Option[V]
  def peek(borrow k : K) : borrow Option[V]
  def take(borrow k : K) : steal Option[V]
}
```

The semantics of this partial map would be such that `take` would remove the mapping for the value k, if it exists, returning an `Option@NONE` if the mapping is undefined. If the value does exist, then the mapping can be stolen without any need change to the type of the map.

The impact of borrow and steal annotations on return values will require further investigation in order to decide what can realistically be achieved, and also what is necessary. Alternative interface designs for traditional data structures may be more usable in a typestate-oriented language; it is important to not just attempt to produce APIs that fit the preconceptions that programmers may have, but instead to design the best possible APIs for the chosen characteristics of the language.

7.3.3 Function literals with implicit binding

Function literals, combined with higher order functions, are incredibly useful and make the concise expression of complex logic possible. The ability to pass a function as a parameter is not particularly new, and can be emulated in languages which do not directly support this by passing *method objects* — objects which contain the one function intended to be called. The `Callable` interface in Java is a class example of this.

Many object oriented languages offer convenient syntax to define function literals which *implicitly bind* values from the context of their definition. In Scala we may define a function which adds a number to every element in a sequence by exploiting this feature:

```
def addX(x : Int, vals : Seq[Int]) : Seq[Int] = vals.map(_ + x)
```

In this code, $\_ + x$ defines a function literal which takes one parameter, inferred to be of type `Int`. This may be written more explicitly as $(y : Int)$ => $y + x$. The value of x is implicitly bound into the scope of this function — it is not passed explicitly as a parameter to the function on every invocation.

The ability to implicitly bind values is incredibly useful, especially when defining such small function literals for use with higher order functions such as `map`.

Implicit binding does not add significant complexity to the type system of a language which is not concerned with alias control or state change. A typestate-oriented language is however concerned with these things, and there are two important aspects of implicit binding that must be formalised and studied:

- Is it possible to distinguish between borrowing and stealing an implicitly bound reference?
- Can effects on implicitly bound references be tracked and interpreted?

Each of these questions are discussed in more detail below.

Borrow / Steal for implicit bindings

As described earlier, function literals are syntactic sugar for the creation of objects with `apply` methods, and implicitly bound references become private fields on such objects. A safe choice would be to treat such references as stolen, but this is often overly restrictive, particularly where a unique reference is implicitly bound.

Function literals can be considered to be *short lived* or *long lived*. Short lived function literals are those which exist to fulfil a temporary function before the reference to it is lost and it is eligible for garbage collection. Long lived function literals are those which escape the scope of their definition, and as such the implicitly bound references they carry also escape.

The vast majority of function literals are short lived, such as in `list.map(_ + 1)` or `list.reduce(_ * _)`. A short lived function may capture a unique reference and use it in a manner which changes its state, but after the function has ceased to serve a purpose a programmer may expect that the original reference should still be unique. For instance, we may implement copying a list as follows:

```
type List[T] {
  ...
  def foreach(borrow f : T => ~Unit) : ~Unit
}

def copy(borrow from : ~List[~Int]) : !List[~Int] = {
  val l = new List[~Int]()
  from.foreach(x => l.add(x))
  return l
}
```

The reference l starts as a unique reference, which is then implicitly bound into the function literal `x => l.add(x)`. The `foreach` function is defined such that it borrows the function literal, therefore we may infer that l will not escape the scope of `copy` as part of the function literal. As the function literal is no longer used after the call to `foreach`, it is not unreasonable to expect that l is still a unique reference when it is returned as the result of the function.

This reasoning represents an extension of the proposed inference of borrow and steal annotations for method parameters, though it is unclear how far this can or should be generalised. For instance, consider the example in Listing 7.14 which splits a list into two pieces. The function `partition` borrows the parameter f, which is a unique reference to a function. This function is provided to the constructor of `Partitioner`, which steals the reference, storing it in a private field. However the `Partitioner` reference is short-lived, existing only locally within the `partition` method after which it would be garbage collected. The parameter f is only temporarily an opaque reference, but it may be reasonable to assume that it can be treated as a unique reference after the return of `partition`. This reasoning is more complex than in the previous example, and further investigation would be required in order to determine whether this can be proven sound in general.

7.3.4 Implicit effects

As in the description of functions in the TS language, the application of a function may have an effect on its bound variables. The same is potentially true of an implicitly bound variable,

```
class Partitioner[T] {
  private val f : ~(T => ~Bool)
  private val trueVals  : #List[T]
  private val falseVals : #List[T]

  new Partitioner(steal f : ~(T => ~Bool)) => ACTIVE = {
    this.f = f
    trueVals = new List[T]()
    falseVals = new List[T]()
  }

  state ACTIVE {
    private val trueVals : !List[T]
    private val falseVals : !List[T]

    def add(t : T) : ~Unit =
    if(f(t)) trueVals.add(t) else falseVals.add(t)

    def extractResult() : !Pair[!List[T], !List[T]] => ROOT = {
      return new Pair(trueVals, falseVals)
    }
  }
}

def partition(borrow list : List[~T], borrow f : !(~T => ~Bool))
    : !Pair[!List[~Int], !List[~Int]]  = {

  val partitioner = new Partitioner(f)
  list.foreach(x => partitioner.add(f))
  return partitioner.extractResult()
}
```

Listing 7.14: A more complex example of scope analysis concerning the parameter *f*

```
type Stack[T] {

  def push(t : T) : ~Unit -> NOT_EMPTY

  def isEmpty() : ~Bool
    :: true  -> ROOT
    :: false -> NOT_EMPTY

  state NOT_EMPTY {
    def pop() : T -> ROOT
  }
}

type List[T] {
  def ~foreach(!f : T => Unit) : Unit
}

def addAll(s : !Stack[Int]@ROOT, l : ~List[Int]) =
  l.foreach(elem => s.push(elem))
```

Listing 7.15: An example of modifying an implicitly bound parameter

but it not obvious how the effect should be represented or interpreted. Consider the example in Listing 7.15, which defines a protocol for an unbounded stack and a method which adds all the elements of a list to a stack.

The stack is defined such that the method `push` is always available and ensures that the stack is in state `NOT_EMPTY`, and method `pop` is available only in the `NOT_EMPTY` state and returns the top element of the stack and returns the object to state ROOT.

The function `addAll` takes a unique reference to a stack and a shared reference to a list. Through implicit binding, the function literal `elem => s.push(elem)` adds each element of the list to the stack. Intuitively, we know that if the list is non-empty then the stack will be in state `NOT_EMPTY` after `addAll` executes. If the list is empty, then the stack will remain unchanged.

However, the process by which the type system could derive this is unclear. The effect on s occurs within the function literal, and the code as presented provides no direct information as to what this effect is. Furthermore, it is also not apparent that the function literal will be invoked, or how many times it may be invoked. This information is required in order for the effect on s to be derived, and applied to the context as part of interpreting the effect of calling the method l.

One way to do this would be to abstract the *implicit effects* of invoking a function with a variable. The augmented notation for a function type $(\overline{T_i \gg U_i}) \xrightarrow{\Delta} V$ could mean the same as in the TS language, but with the addition of an *implicit effect* represented by Δ. The method `foreach` could then be specified as having type:

$$\mathit{foreach} : (T \xrightarrow{\Delta} \mathbf{Unit}) \xrightarrow{\Delta*} \mathbf{Unit}$$

Which specifies that the implicit effect of the function parameter will be applied zero or more times (the Kleene star of the effect). The exact nature of that implicit effect is irrelevant to the implementation of `foreach`; it is only relevant to the context where the function literal is created.

In the example of Listing 7.15, it can be inferred that the implicit effect of the function literal passed to `foreach` is

$$\Delta = \{s : \mathit{Stack}[\mathit{Int}]@\mathit{ROOT} \gg \mathit{Stack}[\mathit{Int}]@\mathit{NOT\_EMPTY}\}$$
$$\texttt{elem => s.push(elem)} : \mathit{Int} \xrightarrow{\Delta} \mathbf{Unit}$$

There we could derive that the implicit effect of applying the function `foreach` is

$$\Delta* = \{s : \mathit{Stack}[\mathit{Int}]@\mathit{ROOT} \gg \mathit{Stack}[\mathit{Int}]@\{\mathit{ROOT}, \mathit{NOT\_EMPTY}\}\}$$

As `NOT_EMPTY` is a descendant state of `ROOT`, it follows that
$\mathit{Stack}[\mathit{Int}]@\{\mathit{ROOT}, \mathit{NOT\_EMPTY}\} = \mathit{Stack}[\mathit{Int}]@\mathit{ROOT}$. We would therefore in theory be able to correctly derive the state change applied to s, despite those state changes occurring in a separate context.

The composition of such implicit effects would rely on the same effect combinators as in the TS language, providing regular language approximations of the implicit effects of higher order functions such as `foreach` or more complex functions which have multiple function parameters. For example, conditional execution could in fact be implemented as a method `branch` on boolean values:

$$\mathit{branch} : \forall\alpha.(() \xrightarrow{\Delta} \alpha, () \xrightarrow{\Delta'} \alpha) \xrightarrow{\Delta|\Delta'} \alpha$$

It is unclear how likely this approach will work in general. The notation used above is inspired by Nielson's treatment of *type and effect systems* [109], though the effects in Nielson's analyses are typically concerned with behavioural aspects of the program that are orthogonal to the context and term type, and are useful for the runtime system — memory allocation information, variable usage, binding time and so on. In the work of Wright [154], effect inference is used to determine the possible type of values in mutable ML references, showing in principle that effect analyses can be used for purposes which are not completely orthogonal to the typing of terms. Through further investigation and experimentation, it may be possible to devise a mechanism which allows for state change on implicitly bound variables.

7.4 Conclusion

This chapter has presented some of the open questions and challenges for a practical, statically-typed language with support for typestate and contemporary features. Through further research it may be discovered that some features, such as functions with implicit capture, do not mix well with typestate. This would help to clearly define the limitations of typestate, which is just as important as defining its strengths.

The exploration of using alias control and the ability to express type change for more than just classic typestate problems, such as in the representation of mutable collections, demonstrates that the techniques required to support static typestate checking may also have wider applications. Clearly demonstrating the utility of these other features may help to promote the adoption of languages with typestate, as the additional complexity required supports more than just one use case.

Overall, this chapter demonstrates that while a strong case is developing for typestate as an essential feature for contemporary object oriented languages, and that hope exists for typestate to co-exist with the features that are now expected in contemporary languages such as Scala, a significant amount of research remains to be done in order to present a fully convincing case for this.

Chapter 8

Conclusion

The goals of this thesis were broad; it attempted to provide a rare combination of theory, practice and empirical evaluation that is so often absent from programming language research. As I hope this work has demonstrated, typestate has great potential as a realistic foundation for the theory of object oriented programming, capable of being truly representative of the kinds of constraints that exist in real software systems.

Typestate modelling simply formalises the kinds of constraints that already exist in object oriented software, and such models make the prospect of automated enforcement of typestate constraints possible. The Hanoi language presented in Chapter 3 was demonstrated to be capable of modelling the most common typestate constraints, and Chapter 4 demonstrated that many options exist for the implementation of dynamic checking of Hanoi models in existing languages.

The user study in Chapter 5 has demonstrated through a rarely conducted empirical study that programmers are able to comprehend typestate models. Additionally, it demonstrated that Hanoi's state-oriented notation was preferable to method-oriented approach, which is used in the seminal work of Aldrich *et al.* on Plural. I believe this study demonstrated the value in user studies which compare notations, despite the inherent difficulties involved. As a research community, I feel we should postulate less and measure more, and this was my attempt to honour this sentiment.

The theoretical work in Chapter 6 aimed to provide a simple setting in which the fundamental problem of typestate inference could be studied. The $remap$ mechanism for interpreting effects in the presence of subtyping is novel, and proved to be important component of constructing a type system which is not overly restrictive. Additionally, the presentation of principal types in this language isolates the fundamental requirement that the effect of a function on its parameter must be known, and that it is possible to define and infer principal typing schemes. The language, while not practical, provides a useful theoretical foundation in which future experimentation with alias control, implicit effects and other such challenges can be considered.

Finally, Chapter 7 outlined the possibilities for a statically typed, expressive typestate language inspired by Scala. While this presentation is light on technical details, I believe it does present a useful discourse on why such a language might be desirable, and the technical challenges which must be overcome in order to make this possible. The unification of the ideas of session types and typestate has potential to providing a safe, flexible environment for the construction of parallel and distributed systems.

8.1 Limitations

There are a number of limitations to the research presented herein. Firstly, the work on the Hanoi language has not yet been connected to the theoretical work on the TS language. The Hanoi language is focused on syntactic convenience for describing real typestate constraints. In Section 3.5.2 a method of reducing Hanoi models to flat finite state machines is presented, which provides the means to relate Hanoi models to TS object protocols. The primary difference between the finite state machines in Hanoi and those in TS object protocols is that TS does not represent or support conditional transitions, either based on return values or exceptions, which are vitally important to the representation of real world typestate constraints. A simplified form of Hanoi without conditional transitions can be used to represent TS object protocols, which would benefit the readability of the type annotations for TS and allow de-duplication of common transitions using state hierarchies. However, TS itself must be extended to support conditional transitions and exceptions in order for the connection between Hanoi and TS to be meaningful.

Both Hanoi and TS also side-step the issue of alias control by requiring unique references. This is convenient for the theoretical work but problematic in practice — at a minimum, it must be possible to specify which methods can be safely invoked through a shared reference. This is very likely to add a significant amount of complexity to the type system. The nascent type inference algorithm presented as part of the TS language requires further study, particularly to determine whether inference is still feasible when conditional transitions and aliasing annotations are included in the formalism. Without the ability to accomodate both, the practical utility of the type inference algorithm is greatly diminished.

The TS language studies the "interface" perspective of interacting with stateful objects, and omits the "implementation" aspect of defining objects with real functionality. Doing so requires the ability to verify that objects conform to their defined protocols, and preferably a mechanism to infer the protocol of an object from its implementation. Studying this problem in detail, and finding an approach which is compatible with the existing work on the TS language, will be an essential step towards providing a more complete and realistic language.

Finally, the user study conducted on Hanoi has a variety of limitations. It was not conducted with enough participants to gain statistically significant results on a number of its hypotheses, diminishing the ability to make any firm claims based on its results. It is also not clear whether the results can be generalised — can it really be said whether a state-oriented approach would be preferable to users in any language other than Hanoi? Does evidence of the ability of programmers to understand Hanoi models in the context of short code examples predict the ability to understand typestate in real programs? A broader study of typestate would be required to provide a more definitive answer to this question. Such a study should also attempt to study programmers as they actually use the language, and build a body of qualitative evidence over a much longer period of time on the attitudes and experiences of programmers using typestate.

8.2 Future Work

This thesis has uncovered many possible directions for further research.

Typestate modelling While the Hanoi language has been demonstrated to be capable of modelling the most common typestate constraints, there are still some areas for potential improvement:

- The ability to specify conditions based on parameter values, while rarely required, would generalise the mechanism for expressing conditional transitions. In particular, the pattern of passing boolean flags to methods to select behaviour is common in some APIs, and may have an impact on the selection of successor states.
- Modelling “views” on an object, as presented by Militão [96] , is likely to be valuable when modelling objects which support communication and sharing between components in a system, and provide better opportunities for customising the semantics of such sharing than the parallel state machines of Plural or Plaid.

  Implementing views requires much deeper integration into the type system and runtime system of a language — “splitting” a reference is not typically an explicit operation, and the runtime system of Java or C# does not typically store the additional meta-data on each reference that would be required to distinguish which view a reference relates to.

Synthesis of typestate and session types In Section 7.2.10 the idea of using typestate constrained objects to represent the endpoints of a channel within a hypothetical language was investigated. Without further work to formalise such a language and use it to try and build real distributed systems, it will not be known where the tradeoffs are in representing session types in such a manner.

With the increasing popularity of functional programming languages as a possible solution to efficient and safe parallelism and concurrency in many-core and distributed systems, the broader utility of typestate in representing stateful contracts to mutable objects may yet become irrelevant. Much like garbage collection has replaced manual memory management in many software systems, it may yet prove simply more convenient to allow a sophisticated runtime system to optimise away the notional inefficiencies of immutable data types where possible, providing a simpler and easier to work with language semantics that unburdens the programmer. If such functional languages were to replace object oriented languages in mainstream programming, then session types would ultimately become the enduring representation of stateful communication, and typestate a footnote in the development of the topic.

However, given the continuing prevalence of object oriented and imperative languages in mainstream industrial programming, I believe such a transition to pure functional languages is not yet within sight. It may never happen for cultural reasons — the decision as to which language is taught to new students is typically made based on pragmatism and the desire to equip students with immediately useful skills [43]. As such, imperative and object oriented languages remain the most popular choices [65], which only serves to reinforce the use of such languages. Additionally, there is not yet any functional language which has the explicit backing and marketing might of a major commercial entity, which is often required for disruptive change in the mix of languages used in industry: Microsoft pushing C# as the primary language of the .NET Framework and its subsequent demand in the jobs market is evidence of this [123].

As such, I believe imperative and object oriented languages will continue to dominate, and slowly adopt useful features and patterns from functional languages identified in the literature as Scala has done. Efforts are still being made to advance the state of the art in imperative languages such as Go and object-functional languages like Scala. With the focus on many-core and distributed systems, the introduction of typestate-like mechanisms for representing and controlling state is a natural fit for such impure languages and provides more flexibility to define constraints on objects which are not just communication channels. For this reason, I believe typestate will “win” over session types in the mainstream, though session types will continue to be useful for the study of the formal underpinnings of distributed systems, and be influential on the evolution of typestate.

Dynamic Checking The dynamic checker for Hanoi has not been profiled or performance optimised, and therefore there are likely to be many ways in which its runtime overhead could be reduced. The use of static analysis to aid the dynamic analysis, such as peformed for tracematches, could also provide a substantial performance improvement.

If sensitivity to method return values could be added to tracematches (and the respective dynamic and static aspects of their implementations), then they would provide sufficient expressivity to enforce the majority of typestate constraints. Furthermore, Hanoi models could be translated into tracematches, in order to directly leverage the extensive performance optimisation research [7, 19–21, 31, 48] that has occurred in this area.

User study The user study presented in Chapter 5 provided some important information on the practicality of typestate and the relative usability of the DSL and annotation model variants. However, statistical significance could not be established for a number of the quantitative aspects of the experiment due to the small sample size used. It is also likely that the think-aloud aspect of the experiment interfered with the timing data collected. As such, a follow-up study which focuses purely on the quantitative aspects, with a greater sample size, would provide more confidence in the results of the experiment.

Effectively the same experiment, in terms of the models and questions presented and the quantitative metrics gathered, could be repeated in an unsupervised setting with a much larger sample size. This could be conducted over the Internet with at least 50 participants. The risk in conducting such an unsupervised experiment is that the timing data is likely to be less reliable, as participants may be distracted or interrupted during the experiment in ways that cannot be controlled for. The large sample size may help to mitigate the impact of such disruptions.

A user study focused on writing Hanoi models, specifically the qualitative aspects of this task, may provide useful insight. While writing models is likely to be less frequent than reading and interpreting them, it is still important to determine whether an adequately trained engineer can construct correct models, and what the most common errors they make during this process are. This could guide modifications to the language, or provide ideas for tools that would help with the construction of typestate models.

Typestate inference The formalism and typestate inference algorithm presented in Chapter 6 provides a foundation for future work on typestate inference. Alias control must be added to the language and experimented with to determine the practicality of some of the features presented in the Chimera language. Exploring the limitations of type inference with implicit binding is also an important area worthy of further research.

A number of important proofs to provide confidence that the type inference approach presented for the TS language remain to be completed. Where flow effects are concerned, further work remains to determine whether a safe approach can be found to solving constraints with variable flow effects, or whether a different representation for constraints is possible which makes solving easier.

Adding the ability to clone and overwrite a reference in the TS language would help to make it more representative of a typical imperative language, and would force aliasing to be considered in the type system. Investigating type inference in the presence of aliasing information represents the next significant challenge for this work. Finally, allowing for the definition of objects with fields and method implementations would result in a language which could provide a foundation for the study of the features described for the Chimera language.

8.3 Availability of code

The Java implementation of Hanoi, which includes support for dynamic proxies and the aspect generator, can be found at `https://bitbucket.org/iainmcgin/hanoi`.

The Scala implementation of the TS language type checker and type inference algorithm can be found at `http://github.com/iainmcgin/ts-lang`.

8.4 Closing remarks

This thesis has demonstrated through the review of existing work and new contributions that typestate can be practical, and that it is a justifiable addition to object oriented languages used for producing systems with non-trivial cost of failure. Value can be added to existing languages through practical dynamic checking based upon the techniques presented in Chapter 4. Such techniques will ensure programs fail immediately upon typestate violations with meaningful, contextual error messages that are likely to prove useful in diagnosing and fixing bugs of this nature.

If a language like Chimera were to be formalised and implemented, then the safety guarantees of static checking could be realised for typestate without a significant burden on the programmer. There are still many open questions and obstacles to be overcome, but they do not seem insurmountable based on the results presented in this thesis. Typestate has a promising future, and I look forward to seeing it unfold.

Appendix A

TS language proofs

A.1 Soundness

Theorem A.1.1 (Progress and preservation, using flow effects)**.** *Given a non-value term t such that $\Gamma \vartriangleright t : T \vartriangleleft \Gamma'$ and a store μ such that $\Gamma \vdash \mu$, then:*

1. *There exists a term t' and store update function υ such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$.*
2. *There exists an Υ such that $\upsilon : \Upsilon$, and that $\Upsilon(\Gamma) \vdash \upsilon(\mu)$.*
3. *There exists a $\Gamma'' \geq \Gamma'$ such that $\Upsilon(\Gamma) \vartriangleright t' : T \vartriangleleft \Gamma''$.*

Proof. by induction on the typing derivation of t. As t is not a value, $\Gamma \vartriangleright t : T \vartriangleleft \Gamma'$ must have been derived by one of the following rules:

- T_LET. It follows that:

  $t = (\textbf{let}\ x = t_v\ \textbf{in}\ t_b) \quad \Gamma \vartriangleright t_v : T_v \vartriangleleft \Gamma_1 \quad \Gamma_1, x : T_v \vartriangleright t_b : T \vartriangleleft \Gamma', x : T'_v$

  There are two cases to consider:

  - t_v is a value. Let $t' = t_b$, $\upsilon = \mathit{replace}(x, t_v)$ and $\Upsilon = \mathit{Replace}(x, T_v)$. By definition, $\upsilon : \Upsilon$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$. Reduction can occur by R_LET_VALUE such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. By Lemma A.10.3, $\Gamma = \Gamma_1$, therefore by Lemma A.1.3, $\Upsilon(\Gamma) = \Gamma_1, x : T_v$. Let $\Gamma'' = \Gamma', x : T'_v$. Trivially, $\Gamma'' \geq \Gamma'$ and $\Upsilon(\Gamma) \vartriangleright t_b : T \vartriangleleft \Gamma''$.
  - t_v is not a value. By induction there exists a t'_v, υ, Υ and Γ'_1 such that:

    $t_v \mid \mu \longrightarrow t'_v \mid \upsilon(\mu) \quad \upsilon : \Upsilon \quad \Upsilon(\Gamma) \vdash \upsilon(\mu) \quad \Gamma'_1 \geq \Gamma_1 \quad \Upsilon(\Gamma) \vartriangleright t'_v : T_v \vartriangleleft \Gamma'_1$

    Let $t' = (\textbf{let}\ x = t'_v\ \textbf{in}\ t_b)$. Reduction can occur by R_LET_TERM such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. As $\Gamma'_1 \geq \Gamma_1$, by the upgrade lemma (A.10.6) there exists a $\Gamma'' \geq \Gamma'$ and $T''_v <: T'_v$. such that $\Gamma'_1, x : T_v \vartriangleright t_b : T \vartriangleleft \Gamma'', x : T''_v$.

    By T_LET, $\Upsilon(\Gamma) \vartriangleright t' : T \vartriangleleft \Gamma''$.

- T_SEQ. It follows that $t = (t_l\,; t_r)$ where $\Gamma \triangleright t_l : T_l \triangleleft \Gamma_{mid}$ and $\Gamma_{mid} \triangleright t_r : T \triangleleft \Gamma'$. There are two cases to consider:

  - t_l is a value. Let $t' = t_r$, $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon : \Upsilon$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$. Reduction can occur by R_SEQ_LEFT_VALUE such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$.

    By Lemma A.10.3, $\Gamma = \Gamma_{mid}$. Let $\Gamma'' = \Gamma'$. Trivially, $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$ and $\Gamma'' \geq \Gamma'$.

  - If t_l is not a value. By induction there exists a t'_v, υ, Υ and Γ'_{mid} such that:
    $t_l \mid \mu \longrightarrow t'_l \mid \upsilon(\mu) \qquad \upsilon : \Upsilon \quad \Upsilon(\Gamma) \vdash \upsilon(\mu) \quad \Gamma'_{mid} \geq \Gamma_{mid}$
    $\Upsilon(\Gamma) \triangleright t'_l : T_v \triangleleft \Gamma'_{mid}$
    Let $t' = t'_l\,; t_r$. Reduction can occur by R_SEQ_LEFT_TERM such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu')$. By the upgrade lemma (A.10.6) , there exists a $\Gamma'' \geq \Gamma'$ such that $\Gamma'_{mid} \triangleright t_r : T \triangleleft \Gamma''$. By T_SEQ, $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$.

- T_FUN_CALL. It follows that $t = x(\overrightarrow{x_i})$, $\Gamma = \Gamma_1, \overline{x_i : T_i}$, $\Gamma(x) = (\overrightarrow{U_i \gg V_i}) \to T$, $\forall i.\, T_i <: U_i$ and $\Gamma' = \Gamma_1, \overline{x_i : remap(T_i, U_i \gg V_i)}$.

  As $\Gamma \vdash \mu$, it follows that $\varnothing \triangleright \mu(x) : \Gamma(x) \triangleleft \varnothing$. It therefore must be the case that $\mu(x) = \lambda(\overrightarrow{x_i : U'_i \gg V'_i}).t_b$ such that $\varnothing \triangleright \mu(x) : (\overrightarrow{U_i \gg V_i}) \to T \triangleleft \varnothing$.

  By Lemma A.10.1, $U'_i \gg V'_i \leq U_i \gg V_i$ for each i and $\overline{y_i : U'_i} \triangleright t_b : T \triangleleft \overline{y_i : V'_i}$.

  Let $t' = t_b\{\overline{{}^{x_i}/_{y_i}}\}$, $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon : \Upsilon$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$. Reduction can occur by R_FUN_CALL such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. By the substitution lemma (A.10.5), $\overline{x_i : U'_i} \triangleright t' : T \triangleleft \overline{x_i : V'_i}$.

  By repeated application of T_WIDEN_FL_EFF, $\overline{x_i : U_i} \triangleright t' : T \triangleleft \overline{x_i : V_i}$. Let $\Gamma'' = \Gamma_1, \overline{x_i : V'_i}$. By the weakening lemma (A.1.5) , $\Gamma \triangleright t' : T \triangleleft \Gamma''$, which is equivalent to $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$. By Lemma A.8.7, $\forall i.remap(T_i, U'_i \gg V'_i) <: remap(T_i, U_i \gg V_i)$. Therefore, $\Gamma'' \geq \Gamma'$.

- T_METH_CALL. It follows that $t = x.m$ with $\Gamma = \Gamma_1, x : O_1@\overline{S_1}$ and $\Gamma' = \Gamma_1, x : O_1@\overline{S_2}$, where $m : T \Rightarrow \overline{S_2} \in O_1@\overline{S_2}$.

  As $\Gamma \vdash \mu$, we must have $\mu(x) = o@S$, where $o = [\ldots S\{\ldots m = (v, S')\}S'\{\ldots\}]$ such that $\varnothing \triangleright \mu(x) : O@S \triangleleft \varnothing$, where $O@S <: O_1@\overline{S_1}$. Additionally, $\varnothing \triangleright v : T' \triangleleft \varnothing$, where $T' <: T$ by SUB_OBJ.

  Let $t' = v$, $\upsilon = call(x, m)$, $\Upsilon = Call(x, m)$ and $\Gamma'' = \Gamma'$. By definition, $\upsilon : \Upsilon$, $\upsilon(\mu)$ and $\Upsilon(\Gamma) = \Gamma'$. Reduction can occur by R_METH_CALL such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. By the weakening lemma (A.1.5) and T_SUB, $\Upsilon(\Gamma) \triangleright v : T \triangleleft \Gamma''$. Reflexively, $\Gamma'' \geq \Gamma'$.

- T_IF. It follows that $t = \textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f$ with $\Gamma \triangleright t_c : \textbf{Bool} \triangleleft \Gamma_1$ and $\Gamma_1 \triangleright t_t : T_t \triangleleft \Gamma_2$ and $\Gamma_1 \triangleright t_f : T_f \triangleleft \Gamma_3$ where $\Gamma' = \Gamma_2 \sqcap \Gamma_3$ and $T = T_t \sqcup T_f$.

  There are three possibilities for reduction:

  - $t_c = \texttt{true}$. Let $t' = t_t$, $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon : \Upsilon$, $\upsilon(\mu) = \mu$ and $\Upsilon(\Gamma) = \Gamma$. Reduction can occur by R_IF_TRUE such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. Trivially, $\Upsilon(\Gamma) \vdash \upsilon(\mu)$.

    By Lemma A.10.3, $\Upsilon(\Gamma) = \Gamma_1$. Let $\Gamma'' = \Gamma_2$. Directly, $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$. By Lemma A.3.15, $\Gamma'' \geq \Gamma'$.

  - $t_c = \texttt{false}$. Let $t' = t_f$ and $\upsilon = id$, $\Upsilon = Id$ and $\Gamma'' = \Gamma_3$. By similar reasoning to the case for $t_c = \texttt{true}$, $t \mid \mu \longrightarrow t' \mid \mu'$ where $\Upsilon(\Gamma) \vdash \upsilon(\mu)$ and
    $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$.

  - t_c is not a value. By induction there exists a t_c', υ, Υ and Γ_1' such that:

    $$t_c \mid \mu \longrightarrow t_c' \mid \upsilon(\mu) \quad \upsilon : \Upsilon \quad \Upsilon(\Gamma) \vdash \upsilon(\mu) \quad \Gamma_1' \geq \Gamma_1 \quad \Upsilon(\Gamma) \triangleright t_c' : \textbf{Bool} \triangleleft \Gamma_1'$$

    By the upgrade lemma (A.10.6) , there exists a $\Gamma_2' \geq \Gamma_2$ such that $\Gamma_1' \triangleright t_t : T_t \triangleleft \Gamma_2'$ and a $\Gamma_3' \geq \Gamma_3$ such that $\Gamma_1' \triangleright t_f : T_f \triangleleft \Gamma_3'$.

    Let $t' = \textbf{if } t_c' \textbf{ then } t_t \textbf{ else } t_f$. By R_IF_TERM, $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. Let $\Gamma'' = \Gamma_2' \sqcap \Gamma_3'$. By Lemma A.3.19, $\Gamma'' \geq \Gamma'$. By T_IF, $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$.

- T_WHILE_A. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$. Contexts Γ_1 through Γ_4 exist where $dom(\Gamma_1) = dom(\Gamma_2)$ and $dom(\Gamma_3) = dom(\Gamma_4)$, and there exists a type T_b such that $\Gamma_1 \triangleright t_c : \textbf{Bool} \triangleleft \Gamma_2$ and $\Gamma_3 \triangleright t_b : T_b \triangleleft \Gamma_4$.

  Let $E_c(x) = extract(x, \Gamma_1, \Gamma_2)$ and $E_b(x) = extract(x, \Gamma_3, \Gamma_4)$.

  Finally, $\Gamma' = \{x : remap(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*) \mid x \in dom(\Gamma)\}$ and $U = \textbf{Unit}$.

  Let $t_t = t_b \,;\, t$, $t_f = \texttt{unit}$, and $t' = (\textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f)$. Let $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon(\mu) = \mu$ and $\Upsilon(\Gamma) = \Gamma$. By R_WHILE, $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. Trivially, $\Upsilon(\Gamma) \vdash \upsilon(\mu)$.

  By definition, $\Gamma'(x) = remap(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*)$. Let
  $f(x) = remap(\Gamma(x), E_c(x))$. By the asserted properties of effect combinators in Definition A.10.1, $\Gamma'(x) = remap(f(x), (E_b(x) \cdot E_c(x))*)$.
  Let $g(x) = remap(f(x), E_b(x) \cdot E_c(x))$. Also by Definition A.10.1,
  $\Gamma'(x) = f(x) \sqcup remap(g(x), (E_b(x) \cdot E_c(x))*)$. By Theorem A.9.1, $f(x) <: \Gamma'(x)$.

  In order for $f(x)$ to be defined for each $x \in dom(\Gamma_1)$, it must be the case that $\Gamma \geq \Gamma_1$. By the upgrade lemma (A.10.6) , it follows that there exists a $\Gamma_5 \geq \Gamma_2$ such that $\Gamma \triangleright t_c : \textbf{Bool} \triangleleft \Gamma_5$, and that $\forall x \in dom(\Gamma_2).\ \Gamma_5(x) <: f(x)$.

In order for $g(x)$ to be defined, it must be the case that $f(x) <: \Gamma_3(x)$. Transitively, $\Gamma_5(x) <: \Gamma_3(x)$, therefore $\Gamma_5 \geq \Gamma_3$. Additionally, as $f(x) <: \Gamma'(x)$, transitively $\Gamma_5(x) <: \Gamma'(x)$. Therefore, $\Gamma_5 \geq \Gamma'$.

By the upgrade lemma (A.10.6) , it follows that there exists a $\Gamma_6 \geq \Gamma_4$ such that $\Gamma_5 \rhd t_b : T_b \lhd \Gamma_6$, and that $\forall x \in \mathit{dom}(\Gamma_4)$. $\Gamma_6(x) <: \mathit{remap}(\Gamma_5(x), E_b(x))$. By Lemma A.6.2, $\mathit{remap}(\Gamma_5(x), E_b(x) \gg\!<) : g(x)$. Transitively, $\Gamma_6(x) <: g(x)$.

Let $\Gamma_7 = \{x : \mathit{remap}(\Gamma_6(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*)\}$.

By Lemma A.6.2, $\Gamma_7(x) <: \mathit{remap}(g(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*)$. Transitively, $\Gamma_7(x) <: \Gamma'(x)$. Therefore $\Gamma_7 \geq \Gamma'$.

By T_SEQ, $\Gamma_5 \rhd t_t : \mathbf{Unit} \lhd \Gamma_7$. By Lemma A.10.4, $\Gamma_5 \rhd t_f : \mathbf{Unit} \lhd \Gamma_5$.
Let $\Gamma'' = \Gamma_5 \sqcap \Gamma_7$. By T_IF, $\Upsilon(\Gamma) \rhd t' : \mathbf{Unit} \lhd \Gamma''$.

As $\Gamma' \leq \Gamma_5$ and $\Gamma' \leq \Gamma_7$, it follows by Lemma A.3.16 that $\Gamma' \leq \Gamma''$, which is equivalent to $\Gamma'' \geq \Gamma'$.

- T_SUB. It follows that there exists a $T' <: T$ such that $\Gamma \rhd t : T' \lhd \Gamma'$. By induction, there exists a t', υ and Υ such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$, $\upsilon : \Upsilon$, $\Upsilon(\Gamma) \vdash \upsilon(\mu)$ and there exists a $\Gamma'' \geq \Gamma'$ such that $\Upsilon(\Gamma) \rhd t' : T' \lhd \Gamma''$. By T_SUB, $\Upsilon(\Gamma) \rhd t' : T \lhd \Gamma''$.

- T_WIDEN_FL_EFF.

  $\Gamma = \Gamma_1, x : U' \quad \Gamma' = \Gamma_2, x : V' \quad \Gamma'_1 = \Gamma_1, x : U \quad \Gamma'_2 = \Gamma_2, x : V$
  $\Gamma'_1 \rhd t : T \lhd \Gamma'_2 \quad U \gg V \leq U' \gg V'$

  As $\Gamma \vdash \mu$ and $\Gamma \geq \Gamma'_1$, it follows that $\Gamma'_1 \vdash \mu$.

  By induction, there exists a t', υ, Υ and Γ'_3 such that:

  $$t \mid \mu \longrightarrow t' \mid \upsilon(\mu) \quad \upsilon : \Upsilon \quad \Upsilon(\Gamma'_1) \vdash \upsilon(\mu) \quad \Gamma'_3 \geq \Gamma'_2 \quad \Upsilon(\Gamma'_1) \rhd t' : T \lhd \Gamma'_3$$

  By Lemma A.1.4 as $\Gamma \geq \Gamma'_1$ and $\Upsilon(\Gamma'_1) \vdash \upsilon(\mu')$, it follows that $\Upsilon(\Gamma) \vdash \upsilon(\mu')$. By Lemma A.1.3, $\Upsilon(\Gamma) \geq \Upsilon(\Gamma'_1)$. By the upgrade lemma (A.10.6) , it follows that $\Upsilon(\Gamma) \rhd t' : T \lhd \Gamma''$, where $\Gamma'' = \{\mathit{remap}(\Upsilon(\Gamma)(x), \Upsilon(\Gamma'_1)(x) \gg \Gamma'_3(x)) \mid x \in \mathit{dom}(\Gamma'_3)\}$ and $\Gamma'' \geq \Gamma'_3$.

  As $\Gamma'' \geq \Gamma'_3$, it follows that there exists a Γ_4 and Γ_3 such that $\Gamma'' = \Gamma_4, x : V''$ and $\Gamma'_3 = \Gamma_3, x : V_3$ where $\Gamma_4 \geq \Gamma_3$ and $V'' <: V_3$. Transitively, $\Gamma_4 \geq \Gamma_2$.

  Let $\Upsilon(\Gamma'_1)(x) = W$ and $\Upsilon(\Gamma)(x) = W'$. As x was an existing variable in Γ'_1, we can observe that either $\Upsilon = \mathit{id}$ or $\Upsilon = \mathit{Call}(y, m)$ for some y and m — the only situation in which $\Upsilon = \mathit{Replace}(y, m)$ is where $y \notin \mathit{dom}(\Gamma)$ (see the reasoning for the T_LET rule). Therefore there are two cases to consider:

  - $\Upsilon = \mathit{id}$ or $\Upsilon = \mathit{Call}(y, m)$ where $y \neq x$. Consequently, $W = U$, $W' = U'$ and $V'' = \mathit{remap}(U', U \gg V)$. As $U \gg V \leq U' \gg V'$, By Lemma A.8.7,

$remap(U', U \gg V) <: remap(U', U' \gg V')$. By Lemma A.7.5, $remap(U', U' \gg V') = V'$, therefore $V'' <: V$.

- $\Upsilon = call(x, m)$ such that there exists an m, X and X' where $m : X \Rightarrow W \in U$ and $m : X' \Rightarrow W' \in U'$. As $U' <: U$, by Lemma A.5.4 it follows that $X' <: X$.

  In this case, the set of all possible traces that may have been executed on U (represented by $U \gg V$) is potentially reduced by the selection of m as a method to call. The traces which may be invoked after m are represented by $W \gg V_3$. The set of traces $\{(m, Y).\delta \mid X <: Y, \delta \in Tr(W \gg V_3)\}$ is necessarily a subset of $Tr(U \gg V_3)$, which is in turn a subset of $Tr(U \gg V)$.

  The same reasoning applies to the selection of m as the method to call on U': the traces which may be invoked after m are represented by $W' \gg V''$, and the set of traces $\{(m, Y').\delta \mid X' <: Y', \delta \in Tr(W' \gg V'')\}$ is necessarily a subset of $Tr(U' \gg V'')$, which is in turn a subset of $Tr(U' \gg V')$. Consequently, $U' \gg V'' \leq U' \gg V'$.

  By Lemma A.7.5 and Lemma A.8.8,
  $V'' = remap(U', U' \gg V'') <: remap(U', U' \gg V') = V'$.

As $\Gamma_4 \geq \Gamma_3$ and $V'' <: V$, it follows that $\Gamma'' \geq \Gamma'$.

□

Theorem A.1.2 (Progress and preservation, using update effects)**.** *Given a non-value term* t *such that* $\Gamma \triangleright t : T \triangleleft \Gamma'$ *and a store* μ *such that* $\Gamma \vdash \mu$*, then:*

1. *There exists a term* t' *and store update function* v *such that* $t \mid \mu \longrightarrow t' \mid v(\mu)$*.*
2. *There exists an* Υ *such that* $v : \Upsilon$*, and that* $\Upsilon(\Gamma) \vdash v(\mu)$*.*
3. *There exists a* $\Gamma'' \geq \Gamma'$ *such that* $\Upsilon(\Gamma) \triangleright t' : T \triangleleft \Gamma''$*.*

Proof. by induction on the typing derivation of t. The proof is very similar to that of Theorem A.1.1, using Lemma A.10.7 instead of Lemma A.10.6 for upgrading derived typings. The following proof steps are notably different:

- T_FUN_CALL. It follows that $t = x(\overrightarrow{x_i})$, $\Gamma = \Gamma_1, \overline{x_i : T_i}$, $\Gamma(x) = (\overrightarrow{U_i \ggg V_i}) \to T$, $\forall i. T_i <: U_i$ and $\Gamma' = \Gamma_1, \overline{x_i : remap(T_i, U_i \ggg V_i)}$.

  By REMAP_UP_DEF, as $T_i <: U_i$, $remap(T_i, U_i \ggg V_i) = V_i$ for each i. Therefore, $\Gamma' = \Gamma_1, \overline{x_i : V_i}$.

  As $\Gamma \vdash \mu$, it follows that $\varnothing \triangleright \mu(x) : \Gamma(x) \triangleleft \varnothing$. It therefore must be the case that $\mu(x) = \lambda(\overrightarrow{x_i : U_i' \ggg V_i'}).t_b$ such that $\varnothing \triangleright \mu(x) : (\overrightarrow{U_i \ggg V_i}) \to T \triangleleft \varnothing$.

By Lemma A.10.1, $U'_i \ggg V'_i \leq U_i \ggg V_i$ for each i. Therefore t_b is typeable such that $\overline{y_i : U'_i} \rhd t_b : T \lhd \overline{y_i : V'_i}$.

Let $t' = t_b\{\overline{{}^{x_i}/_{y_i}}\}$, $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon : \Upsilon$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$. Reduction can occur by R_FUN_CALL such that $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. By Lemma A.10.5, $\overline{x_i : U'_i} \rhd t' : T \lhd \overline{x_i : V'_i}$.

By SUB_UP_EFF, $U_i <: U'_i$ for each i. Therefore by SUB_TRANS, $T_i <: U'_i$, meaning that $\overline{x_i : T_i} \geq \overline{x_i : U'_i}$. By the upgrade lemma (A.10.7) there exists a $\Gamma_2 \geq \overline{x_i : V'_i}$ such that $\overline{x_i : T_i} \rhd t' : T \lhd \Gamma_2$.

Let $\Gamma'' = \Gamma_1, \Gamma_2$. By the weakening lemma (A.1.5) , $\Gamma \rhd t' : T \lhd \Gamma''$, which is equivalent to $\Upsilon(\Gamma) \rhd t' : T \lhd \Gamma''$.

By SUB_UP_EFF, $V'_i <: V_i$. As $\Gamma_2 \geq \overline{x_i : V'_i}$, by SUB_TRANS $\Gamma_2 \geq \overline{x_i : V_i}$. Therefore, $\Gamma'' \geq \Gamma'$.

- T_WHILE_A. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$. Contexts Γ_1 through Γ_4 exist where $dom(\Gamma_1) = dom(\Gamma_2)$ and $dom(\Gamma_3) = dom(\Gamma_4)$, and there exists a type T_b such that $\Gamma_1 \rhd t_c : \textbf{Bool} \lhd \Gamma_2$ and $\Gamma_3 \rhd t_b : T_b \lhd \Gamma_4$.

  Let $E_c(x) = extract(x, \Gamma_1, \Gamma_2)$ and $E_b(x) = extract(x, \Gamma_3, \Gamma_4)$, which are both update effects. Let $E(x) = E_c(x) \cdot (E_b(x) \cdot E_c(x))\star$. Finally, $\Gamma' = \{x : remap(\Gamma(x), E(x)) \mid x \in dom(\Gamma)\}$ and $U = \textbf{Unit}$.

  Let $t_t = t_b\,;\, t$, $t_f = \texttt{unit}$, and $t' = (\textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f)$. Let $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon(\mu) = \mu$ and $\Upsilon(\Gamma) = \Gamma$. By R_WHILE, $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$ (1). Trivially, $\Upsilon(\Gamma) \vdash \upsilon(\mu)$ (2).

  As Γ' is defined, $E(x_i)$ is defined for each $x_i \in dom(\Gamma)$. The inner concatenation expression simplifies such that $E_b(x_i) \cdot E_c(x_i) = in(E_b(x_i)) \ggg out(E_c(x_i))$ and $out(()E_b(x_i)) <: in(E_c(x_i))$. The Kleene star expression simplifies such that $(E_b(x_i) \cdot E_c(x_i))\star = in(E_b(x_i)) \ggg (in(E_b(x_i)) \sqcup out(E_c(x_i)))$. The outer concatenation simplifies such that $E(x) = in(E_c(x_i)) \ggg (in(E_b(x_i)) \sqcup out(E_c(x_i)))$. and $out(E_c(x_i)) <: in(E_b(x_i))$. This final subtyping relation provides one final simplification, such that $E(x) \equiv in(E_c(x_i)) \ggg out(E_c(x_i))$.

  As Γ' is defined, it follows that $\Gamma(x_i) <: in(E(x_i))$ for all $x_i \in dom(\Gamma)$, and $\Gamma'(x_i) = out(E(x_i)) \equiv out(E_c(x_i))$. Therefore $\Gamma \geq \Gamma_1$.

  By the upgrade lemma (A.10.7) , there exists a $\Gamma_5 \geq \Gamma_2$ such that $\Gamma \rhd t_c : \textbf{Bool} \lhd \Gamma_5$.

  As $out(E_c(x_i)) <: in(E_b(x_i))$, $\Gamma_2 \geq \Gamma_3$. Transitively, $\Gamma_5 \geq \Gamma_3$. Therefore by the upgrade lemma (A.10.7) , there exists a $\Gamma_6 \geq \Gamma_4$ such that $\Gamma_5 \rhd t_b : T_b \lhd \Gamma_6$.

  As $out(()E_b(x_i)) <: in(E_c(x_i))$, $\Gamma_4 \geq \Gamma_1$. Transitively, $\Gamma_6 \geq \Gamma_1$. It follows by T_WHILE_A that $\Gamma_6 \rhd t : \textbf{Unit} \lhd \Gamma'$.

By T_UNIT, $\Gamma_5 \rhd t_f : \mathbf{Unit} \lhd \Gamma_5$. As $\Gamma'(x_i) = out(E_c(x_i))$ for each $x_i \in dom(\Gamma)$, it follows that $\Gamma_5 \geq \Gamma'$. By T_WIDEN_UP_EFF, $\Gamma_5 \rhd t_f : \mathbf{Unit} \lhd \Gamma'$.

Therefore by T_IF, $\Gamma \rhd t' : \mathbf{Unit} \lhd \Gamma'$, which is equivalent to $\Upsilon(\Gamma) \rhd t' : \mathbf{Unit} \lhd \Gamma'$. Reflexively, $\Gamma' \geq \Gamma'$ (3).

- T_WHILE_B. It follows that $t = \mathbf{while}\ t_c\ \mathbf{do}\ t_b$ and $T = \mathbf{Unit}$, and there exists a $\Gamma_1 \geq \Gamma$ such that $\Gamma \rhd t_c : \mathbf{Bool} \lhd \Gamma'$ and $\Gamma' \rhd t_b : T_b \lhd \Gamma_1$.

  Let $t_t = t_b\,;\,t$, $t_f = \texttt{unit}$, and $t' = (\mathbf{if}\ t_c\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f)$. Let $\upsilon = id$ and $\Upsilon = Id$. By definition, $\upsilon(\mu) = \mu$ and $\Upsilon(\Gamma) = \Gamma$. By R_WHILE, $t \mid \mu \longrightarrow t' \mid \upsilon(\mu)$. Trivially, $\Upsilon(\Gamma) \vdash \upsilon(\mu)$.

  As $\Gamma_1 \geq \Gamma$, by the upgrade lemma (A.10.7) it follows that there exists a $\Gamma_2 \geq \Gamma'$ such that $\Gamma_1 \rhd t : \mathbf{Unit} \lhd \Gamma_2$.

  By T_SEQ, $\Gamma' \rhd t_t : \mathbf{Unit} \lhd \Gamma_2$. By T_UNIT, $\Gamma' \rhd t_f : \mathbf{Unit} \lhd \Gamma'$.

  Let $\Gamma'' = \Gamma'$. Reflexively, $\Gamma'' \geq \Gamma'$. As $\Gamma_2 \geq \Gamma'$, it follows that $\Gamma_2 \sqcap \Gamma' = \Gamma'$. Therefore by T_IF, $\Gamma \rhd t' : \mathbf{Unit} \lhd \Gamma'$, which is equivalent to $\Upsilon(\Gamma) \rhd t' : \mathbf{Unit} \lhd \Gamma''$.

- T_WIDEN_UP_EFF.

  $$\begin{array}{llll} \Gamma = \Gamma_1, x : U' & \Gamma' = \Gamma_2, x : V' & \Gamma_1' = \Gamma_1, x : U & \Gamma_2' = \Gamma_2, x : V \\ \Gamma_1' \rhd t : T \lhd \Gamma_2' & U \ggg V \leq U' \ggg V' & & \end{array}$$

  As $\Gamma \vdash \mu$ and $\Gamma \geq \Gamma_1'$, it follows that $\Gamma_1' \vdash \mu$.

  By induction, there exists a t', υ, Υ and Γ_3' such that:

  $$t \mid \mu \longrightarrow t' \mid \upsilon(\mu) \quad \upsilon : \Upsilon \quad \Upsilon(\Gamma_1') \vdash \upsilon(\mu) \quad \Gamma_3' \geq \Gamma_2' \quad \Upsilon(\Gamma_1') \rhd t' : T \lhd \Gamma_3'$$

  By Lemma A.1.4 as $\Gamma \geq \Gamma_1'$ and $\Upsilon(\Gamma_1') \vdash \upsilon(\mu')$, it follows that $\Upsilon(\Gamma) \vdash \upsilon(\mu')$. By Lemma A.1.3, $\Upsilon(\Gamma) \geq \Upsilon(\Gamma_1')$. It follows by the upgrade lemma (A.10.7) that there exists a $\Gamma'' \geq \Gamma_3'$ such that $\Upsilon(\Gamma) \rhd t' : T \lhd \Gamma''$.

  As $\Gamma'' \geq \Gamma_3'$, it follows that $\Gamma'' = \Gamma_4, x : V''$ such that $\Gamma_4 \geq \Gamma_3$ and $V'' <: V$. Transitively, $\Gamma_4 \geq \Gamma_2$.

  By SUB_UP_EFF, as $U \ggg V \leq U' \ggg V'$ it follows that $V <: V'$. Transitively, $V'' <: V'$. Therefore, $\Gamma'' \geq \Gamma'$.

□

Lemma A.1.3. *Let* Υ *be a context update function and* $\Gamma' \geq \Gamma$. *If* $\Upsilon(\Gamma)$ *is defined, then* $\Upsilon(\Gamma') \geq \Upsilon(\Gamma)$.

Proof. By case analysis of Υ:

- $\Upsilon = Id$. It follows that $\Upsilon(\Gamma) = \Gamma$ and $\Upsilon(\Gamma') = \Gamma'$. Trivially, $\Upsilon(\Gamma') \geq \Upsilon(\Gamma)$.

- $\Upsilon = \mathit{Replace}(x, T)$. It follows that $\Upsilon(\Gamma) = \Gamma[x \mapsto T]$ and $\Upsilon(\Gamma') = \Gamma'[x \mapsto T]$. Trivially, $\Upsilon(\Gamma') \geq \Upsilon(\Gamma)$.

- $\Upsilon = \mathit{Call}(x, m)$. It follows that $\Gamma = \Gamma_1, x : T$ such that $m : V \Rightarrow U \in T$ and that $\Upsilon(\Gamma) = \Gamma_1, x : U$. As $\Gamma' \geq \Gamma$, it follows that $\Gamma' = \Gamma'_1, x : T'$ where $\Gamma'_1 \geq \Gamma_1$ and $T' <: T$.

  By Lemma A.5.4, there exists a $V' <: V$ and $U' <: U$ such that $m : V' \Rightarrow U' \in T'$. Consequently, $\Upsilon(\Gamma') = \Gamma'_1, x : U'$, meaning $\Upsilon(\Gamma') \geq \Upsilon(\Gamma)$.

□

Lemma A.1.4. *Let υ be a store update function and Υ be a context update function such that $\upsilon : \Upsilon$. If $\Gamma' \vdash \mu$, $\Gamma' \geq \Gamma$ and $\Upsilon(\Gamma) \vdash \upsilon(\mu)$, then $\Upsilon(\Gamma') \vdash \upsilon(\mu)$.*

Proof. By case analysis of the context update function υ:

- $\upsilon = id$. It follows that $\upsilon(\mu) = \mu$ and $\Upsilon(\Gamma') = \Gamma'$. Therefore, $\Gamma' \vdash \mu$ is equivalent to $\Upsilon(\Gamma') \vdash \upsilon(\mu)$.

- $\upsilon = \mathit{replace}(x, v)$. It follows that $\upsilon(\mu) = \mu[x \mapsto v]$. and that $\Upsilon(\Gamma) = \Gamma[x \mapsto T]$ where $\varnothing \rhd v : T \lhd \varnothing$. As $\Gamma' \vdash \mu$, it follows that $\Upsilon(\Gamma') \vdash \upsilon(\mu)$.

- $\upsilon = \mathit{call}(x, m)$. As $\upsilon(\mu)$ is defined, it follows that $\mu = \mu_1, x \mapsto o_1@S$ where $o_1 = [\ldots S\{\ldots m = (v, S') \ldots\} \ldots]$, and that $\mu' = \mu_1, x \mapsto o@S'$.

  As $\Upsilon(\Gamma) \vdash \upsilon(\mu)$, it follows that $\Gamma = \Gamma_1, x : T$ and $\Upsilon(\Gamma) = \Gamma_1, x : V$ such that $\Gamma_1 \vdash \mu_1$, $\varnothing \rhd o_1@S' : V \lhd \varnothing$ and $m : U \Rightarrow V \in T$. Therefore, $\varnothing \rhd o_1@S : T \lhd \varnothing$

  As $\Gamma' \geq \Gamma$, it follows that $\Gamma' = \Gamma'_1, x : T'$ such that and $T' <: T$. Therefore, there exists a $U' <: U$ and $V' <: V$ such that $m : U' \Rightarrow V' \in T'$. As $\Gamma' \vdash \mu$, it follows that $\Gamma'_1 \vdash \mu_1$ and that $\varnothing \rhd o_1@S : T' \lhd \varnothing$.

  By definition, $\Upsilon(\Gamma') = \Gamma'_1, x : V'$ and $\varnothing \rhd o_1@S' : V' \lhd \varnothing$. Therefore, $\Upsilon(\Gamma') \vdash \upsilon(\mu)$.

□

Lemma A.1.5 (Weakening). *If $\Gamma \rhd t : T \lhd \Gamma'$, then for all Γ'' such that $\mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma'') = \varnothing$, it follows that $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.*

Proof. By Lemma A.10.2, $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$, meaning $\mathit{dom}(\Gamma') \cap \mathit{dom}(\Gamma'') = \varnothing$.

We proceed induction on the typing derivation of t.

- derived by one of T_UNIT, T_TRUE, T_FALSE, T_OBJECT or T_FUN_FL_DEF, meaning t is a value and $\Gamma = \Gamma'$. By Lemma A.10.4 it follows that $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_LET. It follows that $t = \textbf{let } x = t' \textbf{ in } t''$ and there exists $\Gamma_1, \Gamma_2, T', T''$ such that $\Gamma \rhd t' : T' \lhd \Gamma_1$ and $\Gamma_1, x : T' \rhd t'' : T \lhd \Gamma', x : T''$.

  By induction, t' can be typed such that $\Gamma, \Gamma'' \rhd t' : T' \lhd \Gamma_1, \Gamma''$, and t'' can be typed such that $\Gamma_1, \Gamma'', x : T' \rhd t'' : T \lhd \Gamma', \Gamma'', x : T''$. We can assume by Barendregt's convention that x can be made distinct from any variable names in Γ'' by relabeling x. Therefore by rule T_LET, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_FUN_CALL. It follows that there exists a $\Gamma_1, T, \overline{T_i}$ and $\overline{T_i'}$ such that $\Gamma = \Gamma_1, x : T, \overline{x_i : T_i}$ and that $\Gamma' = \Gamma_1, x : T, \overline{x_i : T_i'}$. As Γ_1 can be arbitrary in the rule T_FUN_CALL, it follows that we can extend the input and output contexts such that $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_METH_CALL. It follows that there exists a Γ_1 such that $\Gamma = \Gamma_1, x : O@\overline{S}$ and that $\Gamma' = \Gamma_1, x : O@\overline{S}$. As Γ_1 can be arbitrary in rule T_METH_CALL, it follows that we can extend the input and out contexts such that $\Gamma, \Gamma'' \rhd x.m : T \lhd \Gamma', \Gamma''$.

- derived by T_SEQ. Therefore $t = t'; t''$ and there exists Γ_1 and T' such that $\Gamma \rhd t' : T' \lhd \Gamma_1$ and $\Gamma_1 \rhd t'' : T \lhd \Gamma'$. By Lemma A.10.2 it follows that $dom(\Gamma') = dom(\Gamma_1) = dom(\Gamma)$. By induction $\Gamma, \Gamma'' \rhd t' : T' \lhd \Gamma_1, \Gamma''$ and $\Gamma_1, \Gamma'' \rhd t'' : T \lhd \Gamma', \Gamma''$. Therefore by T_SEQ, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma_1, \Gamma''$.

- derived by T_IF. Therefore $t = \textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f$ and there exists Γ_1, Γ_2, Γ_3, T_t and T_f where $T = T_t \sqcup T_f$ and $\Gamma' = \Gamma_2 \sqcap \Gamma_3$ such that:

  $$\Gamma \rhd t_c : \textbf{Bool} \lhd \Gamma_1 \qquad \Gamma_1 \rhd t_t : T_t \lhd \Gamma_2 \qquad \Gamma_1 \rhd t_f : T_f \lhd \Gamma_3$$

  By Lemma A.10.2, the domains of all the contexts are the same. By induction:

  $$\Gamma, \Gamma'' \rhd t_c : \textbf{Bool} \lhd \Gamma_1, \Gamma'' \quad \Gamma_1, \Gamma'' \rhd t_t : T_t \lhd \Gamma_2, \Gamma'' \quad \Gamma_1, \Gamma'' \rhd t_f : T_f \lhd \Gamma_2, \Gamma''$$

  Therefore by T_IF, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_WHILE_A. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$. Contexts Γ_1 through Γ_4 and type T_b exist such that $\Gamma_1 \rhd t_c : \textbf{Bool} \lhd \Gamma_2$ and $\Gamma_3 \rhd t_b : T_b \lhd \Gamma_4$. By definition $\Gamma' = \{x : remap(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x))*) \mid x \in dom(\Gamma_1)\}$ and $T = \textbf{Unit}$, where $E_c(x) = extract(x, \Gamma_1, \Gamma_2)$ and $E_b(x) = extract(x, \Gamma_3, \Gamma_4)$.

  As $dom(\Gamma'') \cap dom(\Gamma) = \varnothing$, it follows that $E_c(x) \cdot (E_b(x) \cdot E_c(x))* = \top \gg \top$ for all $x \in dom(\Gamma'')$. By Lemma A.6.3, $remap(\Gamma''(x), \top \gg \top) = \Gamma''(x)$ for all $x \in dom(\Gamma'')$.

  Therefore, by T_WHILE_A, $\Gamma, \Gamma'' \rhd \textbf{while } t_c \textbf{ do } t_b : \textbf{Unit} \lhd \Gamma', \Gamma''$.

- derived by T_WHILE_B. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$ and $T = \textbf{Unit}$, with $\Gamma \rhd t_c : \textbf{Bool} \lhd \Gamma'$ and $\Gamma' \rhd t_b : T_b \lhd \Gamma$.

  By Lemma A.10.2, $dom(\Gamma') = dom(\Gamma)$, therefore $dom(\Gamma') \cap dom(\Gamma'') = \varnothing$. By induction, $\Gamma, \Gamma'' \rhd t_c : \textbf{Bool} \lhd \Gamma', \Gamma''$ and $\Gamma', \Gamma'' \rhd t_b : T_b \lhd \Gamma, \Gamma''$. Therefore by T_WHILE_B, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_SUB. It follows that $\Gamma \rhd t : T' \lhd \Gamma'$ for $T' <: T$. By induction, $\Gamma, \Gamma'' \rhd t : T' \lhd \Gamma', \Gamma''$. Therefore by rule T_SUB, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

- derived by T_WIDEN_FL_EFF or T_WIDEN_UP_EFF. It follows that there exists effects E and E' such that $E \leq E'$, $\Gamma_1, x : in(E) \rhd t : T \lhd \Gamma_2, x : out(E)$ and $\Gamma_1, x : in(E') \rhd t : T \lhd \Gamma_2, x : out(E')$ where $\Gamma = \Gamma_1, x : in(E')$ and $\Gamma' = \Gamma_2, x : in(E')$.

  By induction, $\Gamma_1, x : in(E), \Gamma'' \rhd t : T \lhd \Gamma_2, x : out(E), \Gamma''$. By T_WIDEN_FL_EFF or T_WIDEN_UP_EFF, $\Gamma, \Gamma'' \rhd t : T \lhd \Gamma', \Gamma''$.

□

A.2 Constraint typing correctness

Theorem A.2.1 (Constraint typing correctness with flow effects). *Let $\Delta \vdash \hat{t} \Rightarrow \dot{t} : \dot{T} \mid_{\chi} C$ and ρ be a substitution such that $\rho \vdash C$ and $\chi \subseteq dom(\rho)$. It follows that $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\rho(\Delta))$.*

Proof. By induction on the derivation of the constraint typing $\Delta \vdash \hat{t} : \dot{t} \mid_{\dot{T}} \chi C$:

- Derived by TC_UNIT, TC_TRUE or TC_FALSE. Trivially by rules T_UNIT, T_TRUE or T_FALSE respectively, $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\rho(\Delta))$.

- Derived by TC_FUN. It follows that:

$$\begin{array}{ccc} \hat{t} = \lambda(\overrightarrow{x_i}).\hat{t}' & \Delta' \vdash \hat{t}' \Rightarrow \dot{t}' : \dot{T}' \mid_{\chi} C & \dot{t} = \lambda(\overrightarrow{x_i : \Delta'(x_i)}).\dot{t}' \\ dom(\Delta') \subseteq \overline{x_i} & \Delta = \varnothing & \dot{T} = (\overrightarrow{\Delta'(x_i)}) \to \dot{T}' \end{array}$$

  Let $\Delta'' = \rho(\Delta')$. As $\rho \vdash C$ and $\chi \subseteq dom(\rho)$, it follows by induction that $in(\Delta'') \rhd \rho(\dot{t}') : \rho(\dot{T}') \lhd out(\Delta'')$. By the weakening lemma (A.1.5) , we can extend the typing of $\dot{t}'$ to $\overline{x_i : in(\Delta''(x_i))} \rhd \rho(t') : \rho(\dot{T}') \lhd \overline{x_i : out(\Delta''(x_i))}$. By definition, $valid(\Delta''(x_i))$ for each x_i. Therefore by rule T_FUN_FL_DEF, $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\rho(\Delta))$.

- Derived by TC_OBJ. It follows that:

$$\forall i,j.\ \varnothing \vdash \hat{v}_{ij} \Rightarrow \dot{v}_{ij} : \dot{T}_{ij} \mid_{\chi_{ij}} C_{ij}$$
$$\hat{t} = \left[\overline{S_i\{\overline{m_{ij} = (\hat{v}_{ij}, S_{ij})}\}}\right]@S \quad \dot{t} = \left[\overline{S_i\{\overline{m_{ij} = (\dot{v}_{ij}, S_{ij})}\}}\right]@S$$
$$\dot{T} = \left\{\overline{S_i\{\overline{m_{ij} : \dot{T}_{ij} \Rightarrow S_{ij}}\}}\right\}@S \quad \Delta = \varnothing \quad C = \bigwedge \overline{C_{ij}} \quad \chi = \bigcup \overline{\chi_{ij}}$$

As $\rho \vdash C$ and $\chi \subseteq dom(\rho)$, it follows that $\rho \vdash C_{ij}$ and $\chi_{ij} \subseteq dom(\rho)$ for each i and j. By induction, $\varnothing \rhd \rho(\dot{v}_{ij}) : \rho(\dot{T}_{ij}) \lhd \varnothing$ for each i and j.

Therefore by rule T_OBJECT, $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\rho(\Delta))$.

- Derived by TC_METH. It follows that:

$$\hat{t} = \dot{t} = x.m \quad \chi = \{\alpha\} \quad \Delta = \{x : O@S \gg O@S'\} \quad \dot{T} = \alpha \quad C = \texttt{true}$$
$$O = \{\ S\{\ m : \alpha \Rightarrow S'\ \}\ S'\{\}\ \}$$

Trivially by rule T_METH_CALL, $in(\rho(\Delta)) \rhd \dot{t} : \rho(\dot{T}) \lhd out(\rho(\Delta))$.

- Derived by TC_CALL_FL. It follows that:

$$\hat{t} = \dot{t} = x(\overrightarrow{x_i}) \quad \Delta = \overline{x_i : \dot{E}_i}, x : (\overrightarrow{\alpha_i \gg \alpha_i'}) \to \alpha \quad \dot{E}_i = \alpha_i'' \gg remap(\alpha_i'', \alpha_i \gg \alpha_i')$$
$$\chi = \overline{\alpha_i}, \overline{\alpha_i'}, \overline{\alpha_i''}, \alpha \quad C = \bigwedge \overline{valid(\alpha_i \gg \alpha_i') \wedge \alpha_i'' <: \alpha_i}$$

As $\rho \vdash C$, it follows that $\rho \vdash valid(\alpha_i \gg \alpha_i')$ and $\rho \vdash \alpha_i'' <: \alpha_i$ for all i. Therefore, $remap(\rho(\alpha_i''), \rho(\alpha_i) \gg \rho(\alpha_i'))$ is defined for each i.

Therefore by rule T_FUN_CALL, $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\alpha) \lhd out(\rho(\Delta))$.

- Derived by TC_SEQ. It follows that:

$$\Delta_a \vdash \hat{t}_a \Rightarrow \dot{t}_a : \dot{T}_a \mid_{\chi_a} C_a \quad \Delta_b \vdash \hat{t}_b \Rightarrow \dot{t}_b : \dot{T} \mid_{\chi_b} C_b$$
$$\hat{t} = \hat{t}_a\ ;\ \hat{t}_b \quad \dot{t} = \dot{t}_a\ ;\ \dot{t}_b \quad \chi = \chi_a \cup \chi_b \quad \Delta = \Delta_a \cdot \Delta_b$$
$$C = \bigwedge\{valid(\Delta(x)) \mid x \in dom(\Delta)\} \wedge C_a \wedge C_b$$

As $\rho \vdash C$, it follows that $\rho \vdash C_a$ and $\rho \vdash C_b$. As $\chi \subseteq dom(\rho)$, it follows that $\chi_a \subseteq dom(\rho)$ and $\chi_b \subseteq dom(\rho)$. Therefore by induction $in(\Delta_a') \rhd \rho(\dot{t}_a) : \rho(\dot{T}_a) \lhd out(\Delta_a')$ and $in(\Delta_b') \rhd \rho(\dot{t}_b) : \rho(\dot{T}) \lhd out(\Delta_b')$ where $\Delta_a' = \rho(\Delta_a)$ and $\Delta_b' = \rho(\Delta_b)$.

Let $dom(\Delta) = \overline{x_i}$. As $\rho \vdash valid(\Delta(x_i))$ for each x_i, it follows that $valid(\Delta_a'(x_i) \cdot \Delta_b'(x_i))$ for each $x_i \in dom(\Delta)$.

Let $\Delta' = \rho(\Delta)$. By Definition A.10.1, $\Delta' = \overline{x_i : T_i \gg V_i}$ such that $\Delta_a'(x_i) \leq T_i \gg U_i$ and $\Delta_b'(x_i) \leq U_i \gg V_i$ where $U_i = remap(T_i, \Delta_a'(x_i))$. By repeated application of

T_WIDEN_FL_EFF and the weakening lemma (A.1.5) , $\overline{x_i : T_i} \rhd \rho(\dot{t}_a) : \rho(\dot{T}_a) \lhd \overline{x_i : U_i}$ and $\overline{x_i : U_i} \rhd \rho(\dot{t}_b) : \rho(\dot{T}) \lhd \overline{x_i : V_i}$.

Therefore by T_SEQ, $in(\Delta') \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\Delta')$.

- Derived by TC_LET. It follows that:

$$\Delta_x \vdash \hat{t}_x \Rightarrow \dot{t}_x : \dot{T}_x \,|_{\chi_x}\, C_x \quad \Delta_b \vdash \hat{t}_b \Rightarrow \dot{t}_x : \dot{T} \,|_{\chi_b}\, C_b$$
$$\hat{t} = \textbf{let } x = \hat{t}_x \textbf{ in } \hat{t}_b \quad \dot{t} = \textbf{let } x = \dot{t}_x \textbf{ in } \dot{t}_b \quad \Delta = \Delta_x \cdot (\Delta_b / x) \quad \chi = \chi_x \cup \chi_b$$
$$C = \bigwedge\{valid(\Delta(y)) \mid y \in dom(\Delta)\} \;\wedge\; C_x \;\wedge\; C_b \;\wedge\; \dot{T}_x <: in(\Delta_b(x))$$

As $\rho \vdash C$, it follows that $\rho \vdash C_x$ and $\rho \vdash C_b$. As $\chi = \chi_x \cup \chi_b$, it follows that $\chi_x \subseteq dom(\rho)$ and $\chi_b \subseteq dom(\rho)$. Therefore by induction $in(\Delta'_x) \rhd \rho(\dot{t}_x) : \rho(\dot{T}_x) \lhd out(\Delta'_x)$ and $in(\Delta'_b) \rhd \rho(\dot{t}_b) : \rho(\dot{T}) \lhd out(\Delta'_b)$ where $\Delta'_x = \rho(\Delta_x)$ and $\Delta'_b = \rho(\Delta_b)$.

Let $dom(\Delta) = \overline{x_i}$. As $\rho \vdash valid(\Delta(x_i))$ for each x_i, it follows that $valid(\Delta'_x(x_i) \cdot \Delta'_b(x_i))$ for each $x_i \in dom(\Delta)$.

By Definition A.10.1, $\Delta = \overline{x_i : T_i \gg V_i}$ such that $\Delta'_x(x_i) \leq T_i \gg U_i$ and $\Delta'_b(x_i) \leq U_i \gg V_i$ where $U_i = remap(T_i, \Delta'_x(x_i))$. Let $T_x = \rho(\dot{T}_x)$. As $\rho \vdash \dot{T}_x <: \Delta_b(x)$, $U_x = remap(T_x, \Delta'_b(x))$ is defined. By Lemma A.7.6, $\Delta'_b(x) \leq T_x \gg U_x$.

By repeated application of T_WIDEN_FL_EFF and the weakening lemma (A.1.5) , $\overline{x_i : T_i} \rhd \rho(\dot{t}_x) : \rho(\dot{T}_x) \lhd \overline{x_i : U_i}$ and
$\overline{x_i : U_i}, x : T_x \rhd \rho(\dot{t}_b) : \rho(\dot{T}) \lhd \overline{x_i : V_i}, x : remap(T_x, \Delta'_b(x))$.

By T_LET, $\overline{x_i : T_i} \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd \overline{x_i : V_i}$, which is equivalent to $in(\rho(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}) \lhd out(\rho(\Delta))$

- Derived by TC_IF. It follows that:

$$\Delta_c \vdash \hat{t}_c \Rightarrow \dot{t}_c : \dot{T}_c \,|_{\chi_c}\, C_c \quad \Delta_t \vdash \hat{t}_t \Rightarrow \dot{t}_t : \dot{T}_t \,|_{\chi_t}\, C_t$$
$$\Delta_f \vdash \hat{t}_f \Rightarrow \dot{t}_f : \dot{T}_f \,|_{\chi_f}\, C_f$$
$$\hat{t} = \textbf{if } \hat{t}_c \textbf{ then } \hat{t}_t \textbf{ else } \hat{t}_f \quad \dot{t} = \textbf{if } \dot{t}_c \textbf{ then } \dot{t}_t \textbf{ else } \dot{t}_f$$
$$\Delta = \Delta_c \cdot (\Delta_t \mid \Delta_f) \quad \chi = \chi_c \cup \chi_t \cup \chi_f$$
$$C = \bigwedge\{valid(\Delta(x)) \mid x \in dom(\Delta)\} \wedge C_c \wedge C_t \wedge C_f \wedge \dot{T}_c <: \textbf{Bool}$$

As $\rho \vdash C$, it follows that $\rho \vdash C_c$, $\rho \vdash C_t$ and $\rho \vdash C_f$. As $\chi \subseteq dom(\rho)$, it follows that $\chi_c \subseteq dom(\rho)$, $\chi_t \subseteq dom(\rho)$ and $\chi_f \subseteq dom(\rho)$. Therefore by induction:

$$in(\Delta'_c) \rhd \rho(\dot{t}_c) : \rho(\dot{T}_c) \lhd out(\Delta'_c) \text{ where } \Delta'_c = \rho(\Delta_c)$$
$$in(\Delta'_t) \rhd \rho(\dot{t}_t) : \rho(\dot{T}_t) \lhd out(\Delta'_t) \text{ where } \Delta'_t = \rho(\Delta_t)$$
$$in(\Delta'_f) \rhd \rho(\dot{t}_f) : \rho(\dot{T}_f) \lhd out(\Delta'_f) \text{ where } \Delta'_f = \rho(\Delta_f)$$

As $\rho \vdash \dot{T}_c <: \mathbf{Bool}$ and the only subtype of $\mathbf{Bool}$ is itself by SUB_REFL, it follows that $\rho(\dot{T}_c) = \mathbf{Bool}$.

Let $dom(\Delta) = \overline{x_i}$. As $\rho \vdash valid(\Delta(x_i))$ for each x_i it follows that $valid(\Delta'_c(x_i) \cdot (\Delta'_t(x_i) \mid \Delta'_f(x_i)))$.

Let $\Delta' = \rho(\Delta)$. Let $x_i \in \Delta'$. By Definition A.10.1, $\Delta'(x_i) = T_i \gg V_i$ for some T_i and V_i such that $\Delta'_c(x_i) \leq T_i \gg U_i$ and $(\Delta'_t(x_i) \mid \Delta'_f(x_i)) \leq U_i \gg V_i$ where $U_i = remap(T_i, \Delta'_c(x_i) \gg)$.

Also by Definition A.10.1, $\Delta'_t(x_i) \leq U_i \gg V_i$ and $\Delta'_f(x_i) \leq U_i \gg V_i$.

By repeated application of T_WIDEN_FL_EFF and the weakening lemma (A.1.5) :

$$\begin{gathered}
\overline{x_i : T_i} \rhd \rho(\dot{t}_c) : \mathbf{Bool} \lhd \overline{x_i : U_i} \\
\overline{x_i : U_i} \rhd \rho(\dot{t}_t) : \rho(\dot{T}_t) \lhd \overline{x_i : V_i} \\
\overline{x_i : U_i} \rhd \rho(\dot{t}_f) : \rho(\dot{T}_f) \lhd \overline{x_i : V_i}
\end{gathered}$$

By T_IF, $\rho(in(\Delta)) \rhd \rho(\dot{t}) : \rho(\dot{T}_t \sqcup \dot{T}_f) \lhd \rho(out(\Delta))$.

- Derived by TC_WHILE. It follows that:

$$\begin{gathered}
\Delta_c \vdash \hat{t}_c \Rightarrow \dot{t}_c : T_c \mid_{\chi_c} C_c \quad \Delta_b \vdash \hat{t}_b \Rightarrow \dot{t}_b : T_b \mid_{\chi_b} C_b \\
\hat{t} = \mathbf{while}\ \hat{t}_c\ \mathbf{do}\ \hat{t}_b \quad \dot{t} = \mathbf{while}\ \dot{t}_c\ \mathbf{do}\ \dot{t}_b \\
\Delta = \Delta_c \cdot (\Delta_b \cdot \Delta_c)* \quad \chi = \chi_c \cup \chi_b \\
C = \bigwedge\{valid(\Delta(x)) \mid x \in dom(\Delta)\} \wedge C_c \wedge C_b \wedge T_c <: \mathbf{Bool}
\end{gathered}$$

As $\rho \vdash C$ it follows that $\rho \vdash C_c$ and $\rho \vdash C_b$. As $\chi \subseteq dom(\rho)$, it follows that $\chi_c \subseteq dom(\rho)$ and $\chi_b \subseteq dom(\rho)$. Therefore by induction:

$$\begin{gathered}
in(\Delta'_c) \rhd \rho(t_c) : \rho(T_c) \lhd out(\Delta'_c) \text{ where } \Delta'_c = \rho(\Delta_c) \\
in(\Delta'_b) \rhd \rho(t_b) : \rho(T_b) \lhd out(\Delta'_b) \text{ where } \Delta'_b = \rho(\Delta_b)
\end{gathered}$$

As $\rho \vdash \dot{T}_c <: \mathbf{Bool}$ and the only subtype of $\mathbf{Bool}$ is itself by SUB_REFL, it follows that $\rho(\dot{T}_c) = \mathbf{Bool}$.

Let $dom(\Delta) = \overline{x_i}$. As $\rho \vdash valid(\Delta(x_i))$ for each x_i, it follows that $valid(\Delta'_c(x_i) \cdot (\Delta'_b(x_i) \cdot \Delta'_c(x_i))*)$.

Let $\Delta' = \rho(\Delta)$, and $x \in dom(\Delta')$. By definition, $\Delta'(x_i) = \Delta'_c(x_i) \cdot (\Delta'_b(x_i) \cdot \Delta'_c(x_i))*$, therefore $out(\Delta'(x_i)) = remap(in(\Delta'(x_i)), \Delta'_c(x_i) \cdot (\Delta'_b(x_i) \cdot \Delta'_c(x_i))*)$ by Lemma A.7.5.

By rule T_WHILE_A, $\rho(in(\Delta)) \rhd \rho(t) : \rho(\dot{T}_t \sqcup \dot{T}_f) \lhd \rho(out(\Delta))$

□

A.3 Properties of contexts

Lemma A.3.1. *Let* $\Gamma_1 \leq \Gamma_2$. *It follows that* $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ *and* $\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$.

Proof. By CTX_LEQ, there are two possibilities:

- $\Gamma_1 \equiv \Gamma_2$, derived by CTX_EQUIV. It follows that $dom(\Gamma_1) = dom(\Gamma_2)$ and that $\forall x \in dom(\Gamma_1).\Gamma_2(x) \equiv \Gamma_1(x)$. Trivially, $dom(\Gamma_1) \subseteq dom(\Gamma_2)$.
  By TY_EQUIV, $\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$.
- $\Gamma_1 < \Gamma_2$, derived by CTX_LT_A or CTX_LT_B. Directly, $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and $\forall x \in dom(\Gamma_1).\,\Gamma_2(x) <: \Gamma_1(x)$.

□

Lemma A.3.2. *Let* Γ_1 *and* Γ_2 *be contexts such that* $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ *and* $\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$. *It follows that* $\Gamma_1 \leq \Gamma_2$.

Proof. There are three cases to consider:

- $dom(\Gamma_1) \subset dom(\Gamma_2)$. By CTX_LT_A, $\Gamma_1 < \Gamma_2$, therefore by CTX_LEQ, $\Gamma_1 \leq \Gamma_2$.
- $dom(\Gamma_1) = dom(\Gamma_2)$ and there exists an $x \in dom(\Gamma_1)$ such that $\Gamma_3(x) \ll: \Gamma_1(x)$, then $\Gamma_1 < \Gamma_3$ by CTX_LT_B, meaning that by CTX_LEQ, $\Gamma_1 \leq \Gamma_3$.
- $dom(\Gamma_1) = dom(\Gamma_2)$ and there does not exist an $x \in dom(\Gamma_1)$ such that $\Gamma_3(x) \ll: \Gamma_1(x)$. This is equivalent to $\forall x \in dom(\Gamma_1).\neg(\Gamma_3(x) \ll: \Gamma_1(x))$.

  Let $y \in dom(\Gamma_1)$. We know that $\Gamma_3(y) <: \Gamma_1(y)$, and that $\neg(\Gamma_3(y) \ll: \Gamma_1(y))$. By SUB_STRICT, the latter is equivalent to $\neg(\Gamma_3(y) <: \Gamma_1(y) \wedge \neg(\Gamma_1(y) <: \Gamma_3(y)))$. In turn this is equivalent to $\neg(\Gamma_3(y) <: \Gamma_1(y)) \vee \neg(\neg(\Gamma_1(y) \leq \Gamma_3(y)))$. It follows that $\Gamma_1(y) \leq \Gamma_3(y)$. Therefore by TY_EQUIV, $\Gamma_3(y) \equiv \Gamma_1(y)$.

  As y was arbitrary in $dom(\Gamma_1)$, it follows that $\forall x \in dom(\Gamma_1).\Gamma_3(x) \equiv \Gamma_1(x)$, therefore by CTX_EQUIV, $\Gamma_1 \equiv \Gamma_3$. Consequently by CTX_LEQ, $\Gamma_1 \leq \Gamma_3$.

□

Corollary A.3.3. $\Gamma_1 \leq \Gamma_2$ *if and only if* $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ *and* $\forall x \in dom(\Gamma_1).\,\Gamma_2(x) <: \Gamma_1(x)$.

Lemma A.3.4. *Let* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \leq \Gamma_1$. *It follows that* $\Gamma_1 \equiv \Gamma_2$.

Proof. By Lemma A.3.1 we have that $dom(\Gamma_1) \subseteq dom(\Gamma_2)$, and that $dom(\Gamma_2) \subseteq dom(\Gamma_1)$. Therefore, $dom(\Gamma_1) = dom(\Gamma_2)$. Also by Lemma A.3.1, we have that
$\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$ and that $\forall x \in dom(\Gamma_2).\Gamma_1(x) <: \Gamma_2(x)$. Therefore by TY_EQUIV, as $dom(\Gamma_1) = dom(\Gamma_2)$, $\forall x \in dom(\Gamma_1).\Gamma_1(x) \equiv dom(\Gamma_2)$. Consequently by CTX_EQUIV, $\Gamma_1 \equiv \Gamma_2$. □

Lemma A.3.5 (Context ordering is reflexive)**.** *Let* Γ *be a context. It follows that* $\Gamma \leq \Gamma$.

Proof. By definition, $dom(\Gamma) \subseteq dom(\Gamma)$. By SUB_REFL, $\forall x \in dom(\Gamma).\Gamma(x) <: \Gamma(x)$. Therefore by CTX_EQUIV, $\Gamma \equiv \Gamma$, meaning $\Gamma \leq \Gamma$ by CTX_LEQ. □

Lemma A.3.6 (Context ordering is transitive)**.** *Let* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \leq \Gamma_3$. *It follows that* $\Gamma_1 \leq \Gamma_3$.

Proof. By Lemma A.3.1 $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and $dom(\Gamma_2) \subseteq dom(\Gamma_3)$. Transitively, $dom(\Gamma_1) \subseteq dom(\Gamma_3)$.
Additionally, $\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$ and $\forall x \in dom(\Gamma_2).\Gamma_3(x) <: \Gamma_2(x)$. By SUB_TRANS, $\forall x \in dom(\Gamma_1).\Gamma_3(x) <: \Gamma_1(x)$. It follows by Lemma A.3.2 that $\Gamma_1 \leq \Gamma_3$. □

Lemma A.3.7. *Let* $\Gamma_1 \equiv \Gamma_1'$ *and* $\Gamma_2 \equiv \Gamma_2'$. *It follows that* $\Gamma_1 \leq \Gamma_2 \iff \Gamma_1' \leq \Gamma_2'$.

Proof. Assume $\Gamma_1 \leq \Gamma_2$. By CTX_EQUIV, $\Gamma_1' \leq \Gamma_1$ and $\Gamma_2 \leq \Gamma_2'$. Therefore by Lemma A.3.6, $\Gamma_1' \leq \Gamma_2'$.

Assume $\Gamma_1' <: \Gamma_2'$. By CTX_EQUIV, $\Gamma_1 <: \Gamma_1'$ and $\Gamma_2' <: \Gamma_2$. Therefore by Lemma A.3.6, $\Gamma_1 <: \Gamma_2$.

Consequently, $\Gamma_1 <: \Gamma_2 \iff \Gamma_1' <: \Gamma_2'$. □

Corollary A.3.8. *The context ordering* $\Gamma \leq \Gamma'$ *is a partial order, up to equivalence.*

Lemma A.3.9. *Context join is not defined for all pairs of contexts.*

Proof. Let $\Gamma_1 = x : \mathbf{Bool}$ and $\Gamma_2 = x : \mathbf{Unit}$. As $\mathbf{Bool} \sqcap \mathbf{Unit}$ is undefined, $\Gamma_1 \sqcup \Gamma_2$ is also not defined. Therefore, join on contexts is partial. □

Lemma A.3.10. *Let* $\Gamma_1 \sqcup \Gamma_2 = \Gamma'$ *be defined. It follows that* $\Gamma_1 \leq \Gamma'$ *and* $\Gamma_2 \leq \Gamma'$.

Proof. By CTX_JOIN, $dom(\Gamma') = dom(\Gamma_1) \cup dom(\Gamma_2)$ and $\forall x \in dom(\Gamma').\Gamma'(x) = \hat{\Gamma}_1(x) \sqcap \hat{\Gamma}_2(x)$.

Consequently, $dom(\Gamma_1) \subseteq dom(\Gamma')$ and $dom(\Gamma_2) \subseteq dom(\Gamma')$. By Theorem A.9.1, $\forall x \in dom(\Gamma_1).\Gamma'(x) <: \Gamma_1(x)$ and $\forall x \in dom(\Gamma_2).\Gamma'(x) <: \Gamma_2(x)$.

Therefore by Lemma A.3.2, $\Gamma_1 \leq \Gamma'$ and $\Gamma_2 \leq \Gamma'$. □

Lemma A.3.11. *Let* Γ_1*,* Γ_2*,* Γ_3 *be contexts such that* $\Gamma_1 \leq \Gamma_3$ *and* $\Gamma_2 \leq \Gamma_3$*. It follows that* $\Gamma_1 \sqcup \Gamma_2 \leq \Gamma_3$.

Proof. Let $\Gamma_1, \Gamma_2, \Gamma_3$ be contexts such that $\Gamma_1 \leq \Gamma_3$ and $\Gamma_2 \leq \Gamma_3$. Let $\Gamma' = \Gamma_1 \sqcup \Gamma_2$.

By CTX_JOIN, $dom(\Gamma') = dom(\Gamma_1) \sqcup dom(\Gamma_2)$ and $\forall x \in dom(\Gamma').\Gamma'(x) <: \hat{\Gamma_1}(x) \sqcap \hat{\Gamma_2}(x)$.

As $\Gamma_1 \leq \Gamma_3$ and $\Gamma_2 \leq \Gamma_3$, it follows by Lemma A.3.1 that $dom(\Gamma_1) \subseteq dom(\Gamma_3)$ and $dom(\Gamma_2) \subseteq dom(\Gamma_3)$. Consequently, $dom(\Gamma_1) \cup dom(\Gamma_2) \subseteq dom(\Gamma_3)$. Therefore $dom(\Gamma') \subseteq dom(\Gamma_3)$.

As $\Gamma_1 \leq \Gamma_3$ and $\Gamma_2 \leq \Gamma_3$, it follows by Lemma A.3.1 that $\forall x \in dom(\Gamma_1).\Gamma_3(x) <: \Gamma_1(x)$ and $\forall x \in dom(\Gamma_2).\Gamma_3(x) <: \Gamma_2(x)$. Let $x \in dom(\Gamma')$. It follows that $\Gamma_3(x) <: \hat{\Gamma_1}(x)$, as either $x \in dom(\Gamma_1)$ and $\hat{\Gamma_1}(x) = \Gamma_1(x)$ where $\Gamma_3(x) <: \Gamma_1(x)$ is already known, or $x \notin dom(\Gamma_1)$ and $\hat{\Gamma_1}(x) = \top$ with $\Gamma_3(x) <: \top$ by SUB_TOP. By similar reasoning, $\Gamma_3(x) <: \hat{\Gamma_2}(x)$. Therefore by Theorem A.9.1, it follows that $\Gamma_3(x) <: \hat{\Gamma_1}(x) \sqcup \hat{\Gamma_2}(x)$, meaning $\Gamma_3(x) <: \Gamma'(x)$. As x was arbitrary within $dom(\Gamma')$, it follows that $\forall x \in dom(\Gamma').\Gamma_3(x) <: \Gamma'(x)$.

By Lemma A.3.2, $\Gamma' \leq \Gamma_3$. □

Lemma A.3.12. *Let* Γ_1*,* Γ_2 *and* Γ_3 *be contexts such that* $\Gamma_1 \leq \Gamma_3$ *and* $\Gamma_2 \leq \Gamma_3$*.*
If $\Gamma_3 \leq \Gamma_1 \sqcup \Gamma_2$*, then* $\Gamma_3 \equiv \Gamma_1 \sqcup \Gamma_2$.

Proof. Let $\Gamma_3 \leq \Gamma_1 \sqcup \Gamma_2$. As $\Gamma_1 \leq \Gamma_3$ and $\Gamma_2 \leq \Gamma_3$, it follows by Lemma A.3.11 that $\Gamma_1 \sqcup \Gamma_2 \leq \Gamma_3$. By Lemma A.3.4, it follows that $\Gamma' \equiv \Gamma_3$. □

Corollary A.3.13. $\Gamma \sqcup \Gamma'$ *is a partial join operator, up to equivalence.*

Lemma A.3.14. *Context meet is defined for any arbitrary pair of contexts.*

Proof. Let Γ_1 and Γ_2 be arbitrary contexts. By definition, $dom(\Gamma_1) \cap dom(\Gamma_2)$ is always defined. Let $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$. By Lemma A.9.2, $\Gamma_1(x) \sqcup \Gamma_2(x)$ is always defined. □

Lemma A.3.15. *Let* $\Gamma_1 \sqcap \Gamma_2 = \Gamma'$*. It follows that* $\Gamma' \leq \Gamma_1$ *and* $\Gamma' \leq \Gamma_2$.

Proof. Let Γ_1 and Γ_2 be contexts such that $\Gamma' = \Gamma_1 \sqcap \Gamma_2$. By CTX_MEET, $dom(\Gamma') = dom(\Gamma_1) \cap dom(\Gamma_2)$ and $\forall x \in dom(\Gamma').\ \Gamma'(x) = \Gamma_1(x) \sqcup \Gamma_2(x)$. By definition, $dom(\Gamma') \subseteq dom(\Gamma_1)$ and $dom(\Gamma') \subseteq dom(\Gamma_2)$. By Theorem A.9.1,
$\forall x \in dom(\Gamma').\ \Gamma_1(x) <: \Gamma'(x)$. and $\forall x \in dom(\Gamma').\ \Gamma_2(x) <: \Gamma'(x)$. Therefore by Lemma A.3.2, $\Gamma' \leq \Gamma_1$ and $\Gamma' \leq \Gamma_2$. □

Lemma A.3.16. *Let* Γ_1*,* Γ_2*,* Γ_3 *be contexts such that* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_1 \leq \Gamma_3$*. It follows that* $\Gamma_1 \leq \Gamma_2 \sqcap \Gamma_3$.

Proof. Let $\Gamma' = \Gamma_2 \sqcap \Gamma_3$. By CTX_MEET, $dom(\Gamma') = dom(\Gamma_2) \cap dom(\Gamma_3)$ and $\forall x \in dom(\Gamma').\Gamma'(x) = \Gamma_2(x) \sqcup \Gamma_3(x)$.

As $\Gamma_1 \leq \Gamma_2$ and $\Gamma_1 \leq \Gamma_3$, it follows by Lemma A.3.1 that $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and $dom(\Gamma_1) \subseteq dom(\Gamma_3)$. Consequently, $dom(\Gamma_1) \subseteq dom(\Gamma_2) \cap dom(\Gamma_3)$.

Also by Lemma A.3.1, it follows that $\forall x \in dom(\Gamma_1).\Gamma_2(x) <: \Gamma_1(x)$ and $\forall x \in dom(\Gamma_1).\Gamma_3(x) <: \Gamma_1(x)$. By Theorem A.9.1, $\forall x \in dom(\Gamma_1).\Gamma_2(x) \sqcup \Gamma_3(x) <: \Gamma_1(x)$, meaning $\forall x \in dom(\Gamma_1).\Gamma'(x) <: \Gamma_1(x)$. Therefore by Lemma A.3.2, $\Gamma_1 \leq \Gamma'$. □

Lemma A.3.17. *Let* Γ_1, Γ_2 *and* Γ_3 *be contexts such that* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_1 \leq \Gamma_3$. *If* $\Gamma_2 \sqcap \Gamma_3 \leq \Gamma_1$, *then* $\Gamma_2 \sqcap \Gamma_3 \equiv \Gamma_1$.

Proof. Let $\Gamma' = \Gamma_2 \sqcap \Gamma_3$. Let $\Gamma' \leq \Gamma_1$. As $\Gamma_1 \leq \Gamma_2$ and $\Gamma_1 \leq \Gamma_3$, it follows by Lemma A.3.16 that $\Gamma_1 \leq \Gamma'$. By Lemma A.3.4, $\Gamma_1 \equiv \Gamma'$. □

Corollary A.3.18. $\Gamma_1 \sqcap \Gamma_2$ *is a total meet operator, up to equivalence.*

Lemma A.3.19. *Let* $\Gamma_1 \leq \Gamma'_1$ *and* $\Gamma_2 \leq \Gamma'_2$. *It follows that* $\Gamma_1 \sqcap \Gamma_2 \leq \Gamma'_1 \sqcap \Gamma'_2$.

Proof. Let $\Gamma = \Gamma_1 \sqcap \Gamma_2$ and $\Gamma' = \Gamma'_1 \sqcap \Gamma'_2$.

By Lemma A.3.1, $dom(\Gamma_1) \subseteq dom(\Gamma'_1)$ and $dom(\Gamma_2) \subseteq dom(\Gamma'_2)$.

By CTX_MEET, $dom(\Gamma) = dom(\Gamma_1) \sqcap dom(\Gamma_2)$ and $dom(\Gamma') = dom(\Gamma'_1) \sqcap dom(\Gamma'_2)$.

Let $x \in dom(\Gamma)$. It follows that $x \in dom(\Gamma_1)$ and $x \in dom(\Gamma_2)$. Therefore, $x \in dom(\Gamma'_1)$ and $x \in dom(\Gamma'_2)$, meaning $x \in dom(\Gamma')$. As x was arbitrary within $dom(\Gamma)$, $dom(\Gamma) \subseteq dom(\Gamma')$.

By Lemma A.3.1, $\forall x \in dom(\Gamma_1).\Gamma'_1(x) <: \Gamma_1(x)$ and $\forall x \in dom(\Gamma_2).\Gamma'_2(x) <: \Gamma_2(x)$.

By CTX_MEET, $\forall x \in dom(\Gamma).\Gamma(x) = \Gamma_1(x) \sqcup \Gamma_2(x)$ and $\forall x \in dom(\Gamma').\Gamma'(x) = \Gamma'_1(x) \sqcup \Gamma'_2(x)$.

Let $x \in dom(\Gamma)$. As $dom(\Gamma) \subseteq dom(\Gamma')$, it follows that $x \in dom(\Gamma')$. Therefore, $\Gamma(x) = \Gamma_1(x) \sqcup \Gamma_2(x)$ and $\Gamma'(x) = \Gamma'_1(x) \sqcup \Gamma'_2(x)$. By Theorem A.9.1, $\Gamma'(x) <: \Gamma(x)$. As x was arbitrary within $dom(\Gamma)$, $\forall x \in dom(\Gamma).\Gamma'(x) <: \Gamma(x)$. By Lemma A.3.2, $\Gamma \leq \Gamma'$. □

A.4 Properties of object and flow effect traces

Lemma A.4.1. *Let* $T <: U$. *It follows that* $Tr(U) \subseteq Tr(T)$.

Proof. by case analysis of the type U.

- U is not an object type. The only trace in $Tr(U)$ is ϵ, by TR_EMPTY. Also by TR_EMPTY, $\epsilon \in Tr(T)$.
- U is an object type, therefore T must be an object type. By SUB_OBJ, $Tr(U) \subseteq Tr(T)$.

□

Lemma A.4.2. *For all $\delta \in Tr(T)$ and $\delta' \in Tr(U)$, if $trmap(T,\delta) <: U$ then $\delta + \delta' \in Tr(T)$.*

Proof. We proceed by induction on the derivation of $\delta \in Tr(T)$:

- by TR_EMPTY. It follows that $\delta = \epsilon$. By TR_MAP_EMPTY, $trmap(T,\delta) = T$. By Lemma A.4.1, $Tr(T) \subseteq Tr(T)$. Therefore, $\delta' \in Tr(T)$. By TR_PLUS_EMPTY, $\delta + \delta' = \delta'$, meaning $\delta + \delta' \in Tr(T)$.
- by TR_PREFIX. It follows that $\delta = (m,V).\delta''$ such that $m : V' \Rightarrow W \in T$ where $V' <: V$ and $\delta'' \in Tr(W)$.

  By TR_MAP_PREFIX, $trmap(T,\delta) = trmap(W,\delta'')$. Consequently,
  $trmap(W,\delta'') <: U$. By induction, $\delta'' + \delta' \in Tr(W)$. By TR_PREFIX,
  $(m,V).\delta'' + \delta' \in Tr(T)$. By TR_PLUS_PREFIX, $\delta + \delta' = (m,V).\delta'' + \delta'$.
  Therefore, $\delta + \delta' \in Tr(T)$.

□

Lemma A.4.3. *If $valid(T \overset{\overline{\delta}}{\gg} U)$, $\delta \in \overline{\delta}$ and $\delta' \in Tr(U)$, then then $\delta + \delta' \in Tr(T)$.*

Proof. By VALID_FL_EFF, $U = \bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta}\}$. Therefore, $trmap(T,\delta) <: U$. By Lemma A.4.2, $\delta + \delta' \in Tr(T)$. □

Lemma A.4.4. *Let δ be a trace of length n. For all T and δ', if $\delta + \delta' \in Tr(T)$ then $\delta' \in Tr(trmap(T,\delta))$.*

Proof. by induction on n. Let $\delta + \delta' \in Tr(T)$.

- $n = 0$. It must be the case that $\delta = \epsilon$. By TR_PLUS_EMPTY, $\delta + \delta' = \delta'$. By TR_MAP_EMPTY, $trmap(T,\delta) = T$. Therefore, $\delta' \in Tr(trmap(T,\delta))$.
- $n = n' + 1$. It must be the case that $\delta = (m,V).\delta''$ where $len(\delta'') = n'$. By TR_PLUS_PREFIX, $\delta + \delta' = (m,V).(\delta'' + \delta')$. By TR_PREFIX, $T = O@\overline{S}$, $m : V' \Rightarrow \overline{S'} \in O@\overline{S}$ where $V' <: V$ and $\delta'' + \delta' \in Tr(O@\overline{S'})$.

  By induction, $\delta' \in Tr(trmap(O@\overline{S'},\delta''))$. By TR_MAP_PREFIX, $trmap(O@\overline{S},\delta) = trmap(O@\overline{S'},\delta'')$. Therefore, $\delta' \in Tr(trmap(T,\delta))$.

□

Lemma A.4.5. *If $\delta \in \mathit{Tr}(T)$, then $\delta \in \mathit{Tr}(T \gg \mathit{trmap}(T, \delta))$.*

Proof. by induction on the length n of the trace $\delta \in \mathit{Tr}(T)$.

- $n = 0$. It must be the case that $\delta = \epsilon$. By TR_MAP_EMPTY, $\mathit{trmap}(T, \delta) = T$. By TR_EFF_EMPTY, $\delta \in \mathit{Tr}(T \gg T)$.
- $n = n' + 1$. It must be the case that $\delta = (m, V).\delta'$ where $\mathit{len}(\delta') = n'$. By TR_PREFIX, $T = O@\overline{S}$, $m : V' \Rightarrow \overline{S'} \in T$ where $V' <: V$ and $\delta' \in \mathit{Tr}(O@\overline{S'})$.

  By induction, $\delta' \in \mathit{Tr}(O@\overline{S'} \gg \mathit{trmap}(O@\overline{S'}, \delta'))$. By TR_MAP_PREFIX, $\mathit{trmap}(O@\overline{S'}, \delta') = \mathit{trmap}(O@\overline{S}, \delta)$. Therefore by TR_EFF_PREFIX, $\delta \in \mathit{Tr}(O@\overline{S} \gg \mathit{trmap}(O@\overline{S}, t))$.

□

Lemma A.4.6. *Let δ be a trace of length n. For all T and U such that $T <: U$ and $\delta \in \mathit{Tr}(U)$, $\mathit{trmap}(T, \delta) <: \mathit{trmap}(U, \delta)$.*

Proof. by induction on n.

- $n = 0$. It must be the case that $\delta = \epsilon$. By TR_MAP_EMPTY, $\mathit{trmap}(T, \delta) = T$ and $\mathit{trmap}(U, \delta) = U$. Trivially, $\mathit{trmap}(T, \delta) <: \mathit{trmap}(U, \delta)$.
- $n = n' + 1$. It must be the case that $\delta = (m, V)\delta'$ where $\mathit{len}(\delta') = n'$. By TR_PREFIX, $U = O_1@\overline{S_1}$, $m : V' \Rightarrow \overline{S_2} \in O_1@\overline{S_1}$ where $V' <: V$ and $\delta' \in \mathit{Tr}(O_1@\overline{S_2})$.

  By Lemma A.4.1, $\delta \in \mathit{Tr}(T)$, therefore by TR_PREFIX $T = O_2@\overline{S_3}$, $m : V'' \Rightarrow \overline{S_4} \in O_2@\overline{S_3}$ where $V'' <: V$ and $\delta' \in \mathit{Tr}(O_2@\overline{S_4})$.

  By induction, $\mathit{trmap}(O_2@\overline{S_4}, \delta') <: \mathit{trmap}(O_1@\overline{S_2}, \delta')$. By TR_MAP_PREFIX, $\mathit{trmap}(T, \delta) = \mathit{trmap}(O_2@\overline{S_4}, \delta')$ and $\mathit{trmap}(U, \delta) = \mathit{trmap}(O_1@\overline{S_3}, \delta)$. Therefore, $\mathit{trmap}(T, \delta) <: \mathit{trmap}(U, \delta)$.

□

Lemma A.4.7. *Let $\mathit{valid}(T \overset{\overline{\delta}}{\gg} U)$ where T is not an object type. It follows that $\overline{\delta} = \{\epsilon\}$ and $T = U$.*

Proof. By VALID_FL_EFF, $\varnothing \subset \overline{\delta} \subseteq \mathit{Tr}(T)$. The only trace that exists in $\mathit{Tr}(T)$ is ϵ, by TR_EMPTY. Consequently, $\overline{\delta} = \{\epsilon\}$, which also means that $U = \mathit{trmap}(T, \epsilon)$. Therefore, $T = U$ by TR_MAP_EMPTY. □

Lemma A.4.8. *For all T, δ and δ' such that $\delta \in Tr(T)$ and $\delta' \in Tr(trmap(T, \delta))$, it follows that $trmap(T, \delta + \delta') = trmap(trmap(T, \delta), \delta')$.*

Proof. By induction on the derivation of $\delta \in Tr(T)$.

- by TR_EMPTY. It follows that $\delta = \epsilon$. By TR_PLUS_EMPTY, $\delta + \delta' = \delta'$. By TR_MAP_EMPTY, $trmap(T, \delta) = T$.

  Directly, $trmap(T, \delta') = trmap(trmap(T, \delta), \delta')$, which is equivalent to $trmap(T, \delta + \delta') = trmap(trmap(T, \delta), \delta')$.

- by TR_PREFIX. It follows that $\delta = (m, V).\delta''$ for some m, V and δ'' such that $len(\delta'') = n'$.

  By TR_PREFIX, there exists a U such that $m : V' \Rightarrow U \in T$ where $V' <: V$ and $\delta'' \in Tr(U)$. By TR_MAP_PREFIX, $trmap(T, \delta) = trmap(U, \delta'')$, therefore $\delta' \in trmap(U, \delta'')$.

  By induction, $trmap(U, \delta'' + \delta') = trmap(trmap(U, \delta''), \delta')$, which is equivalent to $trmap(U, \delta'' + \delta') = trmap(trmap(T, \delta), \delta')$.

  By TR_PLUS_PREFIX, $\delta + \delta' = (m, V).(\delta'' + \delta')$. By TR_PREFIX, $trmap(T, \delta + \delta') = trmap(U, (m, v).(\delta''+\delta'))$. Consequently, $trmap(T, \delta+\delta') = trmap(trmap(T, \delta), \delta')$.

□

A.5 Properties of the subtyping relation

Lemma A.5.1. *For all T and U such that $T <: U$,*

- *If $U = \mathrm{Bool}$, then $T = \mathrm{Bool}$.*
- *If $U = \mathrm{Unit}$, then $T = \mathrm{Unit}$.*
- *If $U = (\overrightarrow{E_i}) \to V$, then there exists a $\overrightarrow{F_i}$ and W such that $T = (\overrightarrow{F_i}) \to W$, $|F_i| = |E_i|$, and $\forall i.F_i \leq E_i$ and $W <: V$.*
- *If $U = O@\overline{S}$, then there exists an O' and $\overline{S'}$ such that $T = O'@\overline{S'}$ and $Tr(O@\overline{S}) \subseteq Tr(O'@\overline{S'})$.*

Proof. Straightforward induction on the derivation of $T <: U$. □

Lemma A.5.2. *For all T and U such that $T <: U$,*

- *If $T = \top$, then $U = \top$.*
- *If $T = \mathbf{Bool}$, then $U = \top$ or $U = \mathbf{Bool}$.*
- *If $T = \mathbf{Unit}$, then $U = \top$ or $U = \mathbf{Unit}$.*
- *If $T = (\overrightarrow{E_i}) \to V$, then $U = \top$ or $U = (\overrightarrow{F_i}) \to W$ such that $|E_i| = |F_i|$, $\forall i.E_i \le F_i$ and $V <: W$.*
- *If $T = O@\overline{S}$, then $U = \top$ or $U = O'@\overline{S'}$ such that $Tr(O'@\overline{S'}) \subseteq Tr(O@\overline{S})$.*

Proof. Straightforward induction on the derivation of $T <: U$. □

Lemma A.5.3. *If $T <: U$ and $kind(T) \ne kind(U)$, then $U = \top$.*

Proof. Straightforward case analysis of the derivation of $T <: U$. □

Lemma A.5.4. *Let $O_1@\overline{S_1} <: O_2@\overline{S_2}$ and $m : T \Rightarrow O_2@\overline{S_4} \in O_2@\overline{S_2}$. It follows that there exists an $\overline{S_3}$ and U such that $m : U \Rightarrow O_1@\overline{S_3}$ where $U <: T$ and $O_1@\overline{S_3} <: O_2@\overline{S_4}$.*

Proof. Let δ be a trace such that $\delta \in Tr(O_2@\overline{S_4})$. It follows by TR_PREFIX that $(m, T).\delta \in Tr(O_2@\overline{S_2})$. By SUB_OBJ, $Tr(O_2@\overline{S_2}) \subseteq Tr(O_1@\overline{S_1})$, meaning $(m, T).\delta \in Tr(O_1@\overline{S_1})$.

By TR_PREFIX, there exists a $\overline{S_3}$ and type $U <: T$ such that $m : U \Rightarrow O_1@\overline{S_3} \in O_1@\overline{S_1}$ and $\delta \in Tr(O_1@\overline{S_3})$. As δ was arbitrary, it follows that $Tr(O_2@\overline{S_4}) \subseteq Tr(O_1@\overline{S_3})$. By SUB_OBJ, $O_1@\overline{S_3} <: O_2@\overline{S_4}$. □

Lemma A.5.5. *Let $T \equiv T'$ and $U \equiv U'$. It follows that $T <: U \iff T' <: U'$.*

Proof. Assume $T <: U$. By TY_EQUIV, $T' <: T$ and $U <: U'$. Therefore by a double application of SUB_TRANS, $T' <: U'$.

Assume $T' <: U'$. By TY_EQUIV, $T <: T'$ and $U' <: U$. Therefore by a double application SUB_TRANS, $T <: U$. Consequently, $T <: U \iff T' <: U'$. □

Corollary A.5.6. *Subtyping is a partial order, up to equivalence.*

A.6 Properties of $remap(T, U \gg V)$

Lemma A.6.1. *If $remap(T, E) = W$, then $W <: out(E)$.*

Proof. There are two cases to consider:

- $E = U \ggg V$. By REMAP_UP_DEF, $remap(T, E) = V$. By SUB_REFL, $remap(T, E) <: V$.

- $E = U \overset{\overline{\delta}}{\gg} V$. By REMAP_FL_DEF, $T <: U$ and $W = \bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta}\}$.

  Let $\delta \in \overline{\delta}$ and $\delta' \in Tr(V)$. By Lemma A.4.3, $\delta + \delta' \in Tr(U)$. By SUB_OBJ, $\delta + \delta' \in Tr(T)$. By Lemma A.4.4, $\delta' \in Tr(trmap(T,t))$. As δ was arbitrary, $\delta' \in Tr(W)$. As δ' was arbitrary, $Tr(V) \subseteq Tr(W)$. Therefore, $W <: V$ by SUB_OBJ.

$\square$

Lemma A.6.2. *Let $valid(E)$ and $T' <: T <: in(E)$. It follows that $remap(T',E) <: remap(T,E)$.*

Proof. There are two cases to consider:

- $E = U \ggg V$. By REMAP_UP_DEF, $remap(T',E) = V$ and $remap(T,E) = V$. By SUB_FL_EFF, $remap(T',E) <: remap(T,E)$.

- $E = U \overset{\overline{\delta}}{\gg} V$ such that $\varnothing \subset \overline{\delta} \subseteq Tr(U)$. Let $\overline{W_1} = \{trmap(T,\delta) \mid \delta \in \overline{\delta}\}$ and $\overline{W_2} = \{trmap(T',\delta) \mid \delta \in \overline{\delta}\}$. By REMAP_FL_DEF, $remap(T,E) = \bigsqcup\overline{W_1}$ and $remap(T,E' \gg=) \bigsqcup\overline{W_2}$.

  Let $\delta \in \overline{\delta}$. By VALID_FL_EFF, $\delta \in Tr(U)$. By SUB_OBJ, $Tr(U) \subseteq Tr(T) \subseteq Tr(T')$, meaning $\delta \in Tr(T)$ and $\delta \in Tr(T')$. By Lemma A.4.6, $trmap(T',\delta) <: trmap(T,\delta)$. Consequently, $\bigsqcup\overline{W_1} <: \bigsqcup\overline{W_2}$, meaning $remap(T', U \overset{\overline{\delta}}{\gg} V) <: remap(T, U \overset{\overline{\delta}}{\gg} V)$.

$\square$

Lemma A.6.3. *Let $T <: U$, where U is not an object type. It follows that*
$T = remap(T, U \gg U)$.

Proof. By definition, $remap(T, U \gg U) = \bigsqcup\{trmap(T,\delta) \mid \delta \in Tr(U \gg U)\}$. As U is not an object type, the only trace in $Tr(U \gg U)$ is ϵ. By TR_MAP_EMPTY, $trmap(T,\epsilon) = T$. Therefore, $remap(T, U \gg U) = T$. $\square$

A.7 Properties of extracted effects

Lemma A.7.1. *For all T and U, $Tr(T \gg U) \subseteq Tr(T)$.*

Proof. Let $\delta \in Tr(T \gg U)$. It must be shown that $\delta \in Tr(T)$. We proceed by induction on the derivation of $\delta \in Tr(T \gg U)$.

- derived by TR_EFF_EMPTY. It follows that $\delta = \epsilon$. By TR_EMPTY, $\delta \in Tr(T)$.

- derived by TR_EFF_PREFIX. It follows that $\delta = (m, V).\delta'$, $T = O@\overline{S}$, $m : V' \Rightarrow O@\overline{S'} \in O@\overline{S}$ where $V' <: V$ and $\delta' \in \mathit{Tr}(O@\overline{S'} \gg U)$.

  By induction, $\delta' \in \mathit{Tr}(O@\overline{S'})$. By TR_PREFIX, $\delta \in \mathit{Tr}(T)$.

- derived by TR_EFF_BRANCH. It follows that there exists a U_1, U_2 and δ' such that $\delta \in \mathit{Tr}(T \gg U_1)$ and $\delta' \in \mathit{Tr}(T \gg U_2)$ such that $U_1 \sqcup U_2 = U$. By induction, $\delta \in \mathit{Tr}(T)$.

$\square$

Lemma A.7.2. *Let $T <: U$. It follows that $\mathit{Tr}(U \gg V) \subseteq \mathit{Tr}(T)$.*

Proof. By Lemma A.7.1, $\mathit{Tr}(U \gg V) \subseteq \mathit{Tr}(U)$. By Lemma A.4.1 $\mathit{Tr}(U) \subseteq \mathit{Tr}(T)$. Transitively, $\mathit{Tr}(U \gg V) \subseteq \mathit{Tr}(T)$. $\square$

Lemma A.7.3. *Let $\mathit{valid}(T \overset{\overline{\delta}}{\gg} U)$. It follows that $\overline{\delta} \subseteq \mathit{Tr}(T \gg U)$.*

Proof. By VALID_FL_EFF, $U = \bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\}$. It follows that $\mathit{trmap}(T, \delta) <: U$ for each δ.

Let $\delta \in \overline{\delta}$. By repeated application of TR_EFF_BRANCH using all other $\delta' \in \overline{\delta}$,
$\delta \in \mathit{Tr}(T \gg U)$. $\square$

Lemma A.7.4. *For all types T and U, if $\delta \in \mathit{Tr}(T \gg U)$, then there exists a set of traces $\overline{\delta} \subseteq \mathit{Tr}(T \gg U)$ where $\delta \in \overline{\delta}$ such that $\bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\} = U$.*

Proof. We proceed by induction on the derivation of $\delta \in \mathit{Tr}(T \gg U)$:

- derived by TR_EFF_EMPTY. By TR_MAP_EMPTY, $\mathit{trmap}(T, \delta) = U$. Let $\overline{\delta} = \{\delta\}$. By definition, $\bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\} = U$.

- derived by TR_EFF_PREFIX. It follows that $\delta = (m, V).\delta_2$ such that $m : V' \Rightarrow W \in T$ where $V' <: V$ and $\delta_2 \in \mathit{Tr}(W \gg U)$.

  By induction, there exists a set $\overline{\delta_2} \subseteq \mathit{Tr}(W \gg U)$ where $\delta_2 \in \overline{\delta_2}$ such that $\bigsqcup\{\mathit{trmap}(W, \delta') \mid \delta' \in \overline{\delta_3}\} = U$. Let $\overline{\delta} = \{(m, V).\delta' \mid \delta' \in \overline{\delta_3}\}$. It follows that $\delta \in \overline{\delta}$. By TR_MAP_PREFIX, $\mathit{trmap}(T, \delta) = \mathit{trmap}(W, \delta')$ for all $\delta \in \overline{\delta}$. Therefore, $\bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\} = U$.

- derived by TR_EFF_BRANCH. It follows that $\delta \in \mathit{Tr}(T \gg U_1)$ and that there exists a $\delta_2 \in \mathit{Tr}(T \gg U_2)$ such that $U_1 \sqcup U_2 = U$.

  By induction, there exists a set $\overline{\delta'} \subseteq \mathit{Tr}(T \gg U_1)$ where $\delta \in \overline{\delta'}$ such that
  $\bigsqcup\{\mathit{trmap}(T, \delta') \mid \delta' \in \overline{\delta'}\} = U_1$. Also, there exists a set $\overline{\delta''} \in \mathit{Tr}(T \gg U_2)$ where $\delta_2 \in \overline{\delta''}$ such that $\bigsqcup\{\mathit{trmap}(T, \delta'') \mid \delta'' \in \overline{\delta''}\} = U_2$.

Let $\overline{\delta} = \overline{\delta'} \cup \overline{\delta''}$. Trivially, $\delta \in \overline{\delta}$. By definition, $\bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta}\} = (\bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta'}\}) \sqcup (\bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta''}\})$. Therefore $\bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta}\} = U$. By TR_EFF_BRANCH, $\overline{\delta} \subseteq Tr(T \gg U)$.

□

Lemma A.7.5. *Let $valid(T \gg U)$. It follows that $remap(T, T \gg U) = U$.*

Proof. By definition, $remap(T,T \gg U) = \bigsqcup\{trmap(T,\delta) \mid \delta \in Tr(T \gg U)\}$. By Lemma A.7.4, for each $\delta_i \in Tr(T \gg U)$ there exists a $\overline{\delta_i} \subseteq Tr(T \gg U)$ where $\delta_i \in \overline{\delta_i}$ such that $\bigsqcup\{trmap(T,\delta_i) \mid \delta_i \in \overline{\delta_i}\} = U$. Let $f(\delta_i) = \overline{\delta_i}$. By definition, $\bigcup \overline{f(\delta_i)} = Tr(T \gg U)$. Finally, it may be observed that
$remap(T,T \gg U) = \bigsqcup\{trmap(T,\delta) \mid \delta \in Tr(T \gg U)\} = \bigsqcup\left\{\bigsqcup \overline{f(\delta_i)}\right\} = U$. □

Lemma A.7.6. *Let $valid(T \gg U)$ and $T' <: T$. It follows that $T \gg U \leq T' \gg U'$, where $U' = remap(T', T \gg U)$.*

Proof. By REMAP_FL_DEF, $U' = \bigsqcup\{trmap(T',t) \mid \delta \in Tr(T \gg U)\}$. By VALID_FL_EFF, $Tr(T \gg U) \neq \varnothing$.

Let $\delta \in Tr(T \gg U)$. By Lemma A.7.1 and Lemma A.4.5, $\delta \in Tr(T' \gg trmap(T',\delta))$. By repeated application of TR_EFF_BRANCH, $\delta \in Tr(T' \gg U')$. As there is at least one such δ, it follows that $valid(T' \gg U')$. As δ was arbitrary, $Tr(T \gg U) \subseteq Tr(T' \gg U')$.

By SUB_FL_EFF, $T \gg U \leq T' \gg U'$.

□

Corollary A.7.7. *Let T and U by types such that $valid(T \gg U)$. Let $T' <: T$ and $U' = remap(T', T \gg U)$. It follows that $valid(T' \gg U')$.*

Lemma A.7.8. *Let T be a type. It follows that $valid(T \gg T)$.*

Proof. By TR_EFF_EMPTY, $\epsilon \in Tr(T \gg T)$, therefore $Tr(T \gg T) \neq \varnothing$.
By VALID_FL_EFF, $valid(T \gg T)$. □

Lemma A.7.9. *Let T, U and V be types such that $valid(T \gg U)$ and $valid(U \gg V)$. It follows that $valid(T \gg V)$.*

Proof. By VALID_FL_EFF, $Tr(T \gg U) \neq \varnothing$ and $Tr(U \gg V) \neq \varnothing$. Let $\delta \in Tr(T \gg U)$ and $\delta' \in Tr(U \gg V)$. By Lemma A.7.10, $\delta + \delta' \in Tr(T \gg V)$, meaning $Tr(T \gg V) \neq \varnothing$. By VALID_FL_EFF, $valid(T \gg V)$. □

Lemma A.7.10. *If $\delta_1 \in Tr(T \gg U)$ and $\delta_2 \in Tr(U \gg V)$ then $\delta_1 + \delta_2 \in Tr(T \gg V)$.*

Proof. Let $\delta_2 \in Tr(U \gg V)$. We proceed by induction on the derivation of $\delta_1 \in Tr(T \gg U)$.

- derived by TR_EFF_EMPTY, meaning $\delta_1 = \epsilon$ and $T = U$. Therefore, $\delta_2 \in Tr(T \gg V)$. By TR_PLUS_EMPTY, $\delta_1 + \delta_2 = \delta_2$, therefore $\delta_1 + \delta_2 \in Tr(T \gg V)$.

- derived by TR_EFF_PREFIX, meaning $\delta_1 = (m, V).\delta_3, T = O@\overline{S}, m : V' \Rightarrow \overline{S'} \in O@\overline{S}$ where $V' <: V$ and $\delta_3 \in Tr(O@\overline{S'} \gg U)$.

  Let $\delta_4 = \delta_3 + \delta_2$. By induction, $\delta_4 \in Tr(O@\overline{S'} \gg V)$. By TR_PLUS_PREFIX, $\delta_1 + \delta_2 = (m, V).\delta_4$. By TR_EFF_PREFIX, $\delta_1 + \delta_2 \in Tr(T \gg V)$.

- derived by TR_EFF_BRANCH, meaning that there exists a U_1, U_2 and δ_1' such that $\delta_1 \in Tr(T \gg U_1)$ and $\delta_3 \in Tr(T \gg U_2)$ where $U_1 \sqcup U_2 = U$.

  If T is not an object type, then by Lemma A.4.7, $T = U_1 = U_2 = V$ and $\delta_1 = \delta_2 = \epsilon$. Trivially, $U_1 \sqcup U_2 = U$ $\delta_1 + \delta_2 \in Tr(T \gg V)$.

  If T is an object type, then by definition U_1 and U_2 are also an object types with the same object protocol. Let $T = O@\overline{S_1}$, $U_1 = O@\overline{S_2'}$ and $U_2 = O@\overline{S_2''}$. By definition, $U = O@\overline{S_2}$ where $\overline{S_2} = \overline{S_2'} \cup \overline{S_2''}$. By definition, V has the same object protocol such that $V = O@\overline{S_3}$ for some $\overline{S_3}$.

  By Lemma A.7.11, there exists a $\overline{S_3'}$, $\overline{S_3''}$ where $\overline{S_3} = \overline{S_3'} \cup \overline{S_3''}$ and δ_2' such that $\delta_2 \in Tr(O@\overline{S_2'} \gg O@\overline{S_3'})$ and $\delta_2' \in Tr(O@\overline{S_2''} \gg O@\overline{S_3''})$.

  By induction, $\delta_1 + \delta_2 \in Tr(O@\overline{S_1} \gg O@\overline{S_3'})$. Also by induction, $\delta_1' + \delta_2' \in Tr(O@\overline{S_1} \gg O@\overline{S_3''})$.

  By TR_EFF_BRANCH, $\delta_1 + \delta_2 \in Tr(O@\overline{S_1} \gg O@\overline{S_3})$, which is equivalent to $\delta_1 + \delta_2 \in Tr(T \gg V)$.

□

Lemma A.7.11. *Let $\delta \in Tr(O@\overline{S_1} \gg O@\overline{S_2})$. For all $\overline{S_1'}$ and $\overline{S_1''}$ such that $\overline{S_1} = \overline{S_1'} \cup \overline{S_1''}$, there exists a $\overline{S_2'}$, $\overline{S_2''}$ where $\overline{S_2} = \overline{S_2'} \cup \overline{S_2''}$ and δ' such that $\delta \in Tr(O@\overline{S_1'} \gg O@\overline{S_2'})$ and $\delta' \in Tr(O@\overline{S_1''} \gg O@\overline{S_2''})$.*

Proof. By induction on the derivation of $\delta \in Tr(O@\overline{S_1} \gg O@\overline{S_2})$.

- derived by TR_EFF_EMPTY. It follows that $\delta = \epsilon$ and that $O@\overline{S_1} = O@\overline{S_2}$.

  Let $\overline{S_2'} = \overline{S_1'}$ and $\overline{S_2''} = \overline{S_1''}$. Trivially, $\overline{S_2} = \overline{S_2'} \cup \overline{S_2''}$.

  Let $\delta' = \delta$. Trivially, $\delta \in Tr(O@\overline{S_1'} \gg O@\overline{S_2'})$ and $\delta' \in Tr(O@\overline{S_1''} \gg O@\overline{S_2''})$.

- derived by TR_EFF_PREFIX. It follows that $\delta = (m, T).\delta'$, $m : V \Rightarrow \overline{S} \in O@\overline{S_1}$ where $V <: T$ and $\delta' \in Tr(OS \gg O@\overline{S_2})$.

  The method existence predicate is defined such that $m : V \Rightarrow \overline{S} \in O@\overline{S_1}$ holds when, for each $S_i \in \overline{S_1}$, $m : V_i \Rightarrow S'_i \in O@S_i$. V and $\overline{S}$ are defined such that $V = \bigsqcup \overline{V_i}$ and $\overline{S} = \bigcup \overline{S'_i}$.

  As $\overline{S_1} = \overline{S'_1} \cup \overline{S''_1}$, it follows that two related method existence predicates hold: $m : V' \Rightarrow \overline{S'} \in O@\overline{S'_1}$ and $m : V'' \Rightarrow \overline{S''} \in O@\overline{S''_1}$ such that $V = V' \sqcup V''$ and $\overline{S} = \overline{S'} \cup \overline{S''}$.

  By induction, there exists a $\overline{S'_2}$, $\overline{S''_2}$ and δ'' such that $\delta' \in Tr(O@\overline{S'} \gg \overline{S'_2})$ and $\delta'' \in Tr(O@\overline{S''} \gg \overline{S''_2})$ where $\overline{S_2} = \overline{S'_2} \cup \overline{S''_2}$.

  By TR_EFF_PREFIX, $\delta \in Tr(O@\overline{S'_1} \gg O@\overline{S'_2})$ and $(m, V_4).\delta'' \in Tr(O@\overline{S''_1} \gg O@\overline{S''_2})$.

- derived by TR_EFF_BRANCH. It follows that there exists a $\overline{S_{2a}}$, $\overline{S_{2b}}$ and δ_2 such that $\delta \in Tr(O@\overline{S_1} \gg \overline{S_{2a}})$ and $\delta_2 \in Tr(O@\overline{S_1} \gg \overline{S_{2b}})$ where $\overline{S_2} = \overline{S_{2a}} \cup \overline{S_{2b}}$.

  By induction, there exists a $\overline{S'_{2a}}$, $\overline{S''_{2a}}$ and δ_3 such that $\delta \in Tr(O@\overline{S'_1} \gg O@\overline{S'_{2a}})$ and $\delta_3 \in Tr(O@\overline{S''_1} \gg O@\overline{S''_{2a}})$ where $\overline{S_{2a}} = \overline{S'_{2a}} \cup \overline{S''_{2a}}$.

  Also by induction, there exists a $\overline{S'_{2b}}$, $\overline{S''_{2b}}$ and δ_4 such that $\delta_2 \in Tr(O@\overline{S'_1} \gg O@\overline{S'_{2b}})$ and $\delta_4 \in Tr(O@\overline{S''_1} \gg O@\overline{S''_{2b}})$ where $\overline{S_{2b}} = \overline{S'_{2b}} \cup \overline{S''_{2b}}$.

  Let $\overline{S'_2} = \overline{S'_{2a}} \cup \overline{S'_{2b}}$ and $\overline{S''_2} = \overline{S''_{2a}} \cup \overline{S''_{2b}}$.

  By TR_EFF_BRANCH, $\delta \in Tr(O@\overline{S'_1} \gg O@\overline{S'_2})$. Let $\delta' = \delta_3$. By TR_EFF_BRANCH, $\delta' \in Tr(O@\overline{S''_1} \gg O@\overline{S''_2})$.

□

A.8 Properties of the sub-effect relation

Lemma A.8.1. *For all E and F such that $E \leq F$,*

- $F = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ *if and only if* $E = T_1 \overset{\overline{\delta_1}}{\gg} U_1$ *such that* $T_2 <: T_1$ *and* $\overline{\delta_1} \subseteq \overline{\delta_2}$.

- $F = T_1 \ggg U_1$ *if and only if* $E = T_2 \ggg U_2$ *such that* $T_2 <: T_1$ *and* $U_2 <: U_1$.

Proof. Straightforward induction on the derivation of $E \leq F$. □

Lemma A.8.2. *Let E be an effect type such that $valid(E)$. It follows that $E \leq E$.*

Proof. By case analysis of the effect type E:

- $E = T \ggg U$ for some types T and U. By SUB_REFL, $T <: T$ and $U <: U$. Therefore, by SUB_UP_EFF, $E \le E$.

- $E = T \gg U$ for some types T and U. By SUB_REFL, $T <: T$. Trivially, $Tr(T \gg U) \subseteq Tr(T \gg U)$. Therefore by SUB_FL_EFF, $T \gg U \le T \gg U$.

□

Corollary A.8.3. *The sub-effect relation is reflexive.*

Lemma A.8.4. *Let $E \equiv E'$ and $F \equiv F'$. It follows that $E \le F \iff E' \le F'$.*

Proof. Assume $E \le F$. By EFF_EQUIV, $E' \le E$ and $F \le F'$. By double application of SUB_EFF_TRANS, $E' \le F'$.

Assume $E' \le F'$. By EFF_EQUIV, $E \le E'$ and $F' \le F$. By double application of SUB_EFF_TRANS, $E \le F$. □

Corollary A.8.5. *The sub-effect relation is a partial order, up to equivalence.*

Lemma A.8.6. *Let $valid(T \overset{\overline{\delta_1}}{\gg} U)$ and $valid(T \overset{\overline{\delta_2}}{\gg} V)$. It follows that $valid(T \overset{\overline{\delta}}{\gg} U \sqcup V)$ where $\overline{\delta} = \overline{\delta_1} \cup \overline{\delta_2}$.*

Proof. By VALID_FL_EFF:

$$\begin{array}{ll} \varnothing \subset \overline{\delta_1} \subseteq Tr(T) & U = \bigsqcup\{trmap(T,t) \mid \delta \in \overline{\delta_1}\} \\ \varnothing \subset \overline{\delta_2} \subseteq Tr(T) & V = \bigsqcup\{trmap(T,t) \mid \delta \in \overline{\delta_2}\} \end{array}$$

It follows that $\overline{\delta_1} \cup \overline{\delta_2} \subseteq Tr(T)$. Let $W = \bigsqcup\{trmap(T,t) \mid \delta \in \overline{\delta}\}$. It follows that $W = U \sqcup V$, and that $valid(T \overset{\overline{\delta}}{\gg} W)$ by VALID_FL_EFF. □

Lemma A.8.7. *Let $E \le E'$ and $remap(T, E')$ be defined. It follows that $remap(T, E) <: remap(T, E')$.*

Proof. There are two cases to consider:

- $E' = T_2 \ggg U_2$ for some T_2 and U_2. By Lemma A.8.1, $E = T_1 \ggg U_1$ for some T_1 and U_1 such that $T_2 <: T_1$ and $U_1 <: U_2$.

  By REMAP_UP_DEF, $T <: T_2$ and $remap(T, E') = U_2$. By SUB_TRANS, $T <: T_1$ and therefore $remap(T, E) = U_1$ by REMAP_UP_DEF. Consequently, $remap(T, E) <: remap(T, E')$.

- $E' = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ for some T_2, U_2 and $\overline{\delta_2}$. By Lemma A.8.1, $E = T_1 \overset{\overline{\delta_1}}{\gg} U_1$ such that $T_2 <: T_1$ and $\overline{\delta_1} \subseteq \overline{\delta_2}$, and that both effects are valid. By REMAP_FL_DEF, $T <: T_2$.

  Let $\overline{\delta} = \overline{\delta_2} - \overline{\delta_1}$. Let $W = \bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta_1}\}$ and $W' = \bigsqcup\{trmap(T,\delta) \mid \delta \in \overline{\delta}\}$. By REMAP_FL_DEF, $remap(T,E) = W$ and $remap(T,E') = W \sqcup W'$. It follows that $remap(T,E) <: remap(T,E')$.

□

Lemma A.8.8. *Let $U \overset{\overline{\delta}}{\gg} V \leq U' \overset{\overline{\delta'}}{\gg} V'$, $T <: U'$, $W = remap(T, U \overset{\overline{\delta}}{\gg} V)$ and $W' = remap(T, U' \overset{\overline{\delta'}}{\gg} V')$. It follows that $T \overset{\overline{\delta}}{\gg} W \leq T \overset{\overline{\delta'}}{\gg} W'$ and $W <: W'$.*

Proof. By SUB_OBJ, $Tr(U') \subseteq Tr(T)$. By SUB_FL_EFF, $valid(U \overset{\overline{\delta}}{\gg} V)$, $valid(U' \overset{\overline{\delta'}}{\gg} V')$ and $\overline{\delta} \subseteq \overline{\delta'}$, By VALID_FL_EFF, $\varnothing \subset \overline{\delta} \subseteq Tr(U)$ and $\varnothing \subset \overline{\delta'} \subseteq Tr(U')$. Transitively, $\overline{\delta} \subseteq Tr(T)$ and $\overline{\delta'} \subseteq Tr(T)$.

Consequently by VALID_FL_EFF, $valid(T \overset{\overline{\delta}}{\gg} W)$ and $valid(T \overset{\overline{\delta'}}{\gg} W')$. As $\overline{\delta} \subseteq \overline{\delta'}$, $T \overset{\overline{\delta}}{\gg} W \leq T \overset{\overline{\delta'}}{\gg} W'$ by SUB_FL_EFF. □

A.9 Properties of join and meet

Theorem A.9.1. *For all type and effect expressions of form $P \diamond Q = R$:*

1. *If P is a type and $\diamond = \sqcup$, it follows that Q and R are also types such that:*
   (a) *R is an upper bound of P and Q: $P <: R$ and $Q <: R$.*
   (b) *R is a lower bound of any other upper bound of P and Q: $P <: R' \wedge Q <: R' \implies R <: R'$.*
2. *If P is a type and $\diamond = \sqcap$, it follows that Q and R are also types such that:*
   (a) *R is a lower bound of P and Q: $R <: P$ and $R <: Q$.*
   (b) *R is an upper bound of any other lower bound of P and Q: $R' <: P \wedge R' <: Q \implies R' <: R$.*
3. *If P is an effect and $\diamond = \sqcup$, it follows that Q and R are also effects such that:*
   (a) *R is an upper bound of P and Q: $P \leq R$ and $Q \leq R$.*
   (b) *R is a lower bound of any other upper bound of P and Q: $P \leq R' \wedge Q \leq R' \implies R \leq R'$.*

4. *If P is an effect and $\diamond = \sqcap$, it follow that Q and R are also effects such that:*

   (a) *R is a lower bound of P and Q: $R \leq P$ and $R \leq Q$.*

   (b) *R is an upper bound of any other lower bound of P and Q: $R' \leq P \wedge R' \leq Q \implies R' \leq R$.*

Proof. By induction on the derivation of the expression $P \diamond Q = R$:

- derived by TY_JOIN_REFL. It follows that P is a type and $\diamond = \sqcup$ such that $P = Q = R$. By SUB_REFL, $P <: R$ and $Q <: R$, satisfying (1a). Let R' be some other type such that $P <: R'$ and $Q <: R'$. As $P = R$, trivially $R <: R'$, satisfying (1b).

- derived by TY_JOIN_FUN. It follows that P is a type and $\diamond = \sqcup$ such that $P = (\overrightarrow{E_i}) \to X$, $Q = (\overrightarrow{F_i}) \to Y$ where $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right|$ and $R = (\overrightarrow{G_i}) \to Z$ such that $\forall i.\ E_i \sqcup F_i = G_i$ and $X \sqcup Y = Z$.

  By induction, $\forall i.\ E_i \leq G_i \wedge F_i \leq G_i$ and $\forall i, G_i'.\ E_i \leq G_i' \wedge F_i \leq G_i' \implies G_i \leq G_i'$. Also by induction, $X <: Z, Y <: Z$ and $\forall Z'.\ X <: Z' \wedge Y <: Z' \implies Z <: Z'$.

  It follows by SUB_FN that $P <: R$ and $Q <: R$, satisfying (1a).

  Let R' be a type such that $P <: R'$ and $Q <: R'$. By Lemma A.5.1, it follows that $R' = \top$ or $R' = (\overrightarrow{H_i}) \to A$ such that $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right| = \left|\overrightarrow{H_i}\right|$, $\forall i.\ E_i \leq H_i \wedge F_i \leq H_i$, $X <: A$ and $Y <: A$.

  - If $R' = \top$, trivially $R <: R'$ by SUB_TOP, satisfying (1b).
  - If $R' = (\overrightarrow{H_i}) \to A$, as $\forall i.\ E_i \leq H_i \wedge F_i \leq H_i$ and $\forall i, G_i'.\ E_i \leq G_i' \wedge F_i \leq G_i' \implies G_i \leq G_i'$, it follows that $\forall i.\ G_i \leq H_i$. As $X <: A$, $Y <: A$ and $\forall Z'.\ X <: Z' \wedge Y <: Z' \implies Z <: Z'$, it follows that $Z <: A$. Therefore by SUB_FN, $R <: R'$, satisfying (1b).

- derived by TY_JOIN_OBJ. It follows that P is a type and $\diamond = \sqcup$, such that $P = O_1@S_1$ and $Q = O_2@S_2$ where $O_1 = (M_1, \Sigma_1, \Delta_1)$ and $O_2 = (M_2, \Sigma_2, \Delta_2)$. R is defined to be $O_3@S_3$ such that $O_3 = (M_3, \Sigma_3, \Delta_3)$ where $M_3 = M_1 \cup M_2$,
  $\Sigma_3 = \{L(S_1, S_2) \mid S_1 \in \Sigma_2,\ S_2 \in \Sigma_2\}$, $\Delta_3 = \Delta_1 \sqcup \Delta_2$ and $S_3 = L(S_1, S_2)$.

  It must be shown that $O_1@S_1 <: O_3@S_3$ and $O_2@S_2 <: O_3@S_3$, which requires that $Tr(O_3@S_3) \subseteq Tr(O_1@S_1)$ and $Tr(O_3@S_3) \subseteq Tr(O_2@S_2)$. In order to do this, we prove the following sub-lemma:

  For all $S_a \in \Sigma_1$, $S_b \in \Sigma_2$ such that $S = L(S_a, S_b)$, if $\delta \in Tr(O_3@S)$ then $\delta \in Tr(O_1@S_a)$ and $\delta \in Tr(O_2@S_b)$.

  We prove this by induction on the length of the trace δ:

- $len(\delta) = 0$, meaning $\delta = \epsilon$. By TR_EMPTY, $\delta \in Tr(O_1@S_1)$ and $\delta \in Tr(O_2@S_2)$.
- $len(\delta) = n + 1$, meaning $\delta = (m, T).\delta'$ such that $len(\delta') = n$. By TR_PREFIX, $(S_3, m, T', S_3') \in \Sigma_3$ with $T' <: T$ and $\delta' \in Tr(O_3@S_3')$. Therefore, by TRANS_JOIN, $(S_1, m, T_1, S_1') \in \Delta_1$ and $(S_2, m, T_2, S_2') \in \Delta_2$ where $T_1 \sqcup T_2 = T'$ and $S_3' = L(S_1', S_2')$.

  By induction, $\delta' \in Tr(O_1@S_1')$ and $\delta' \in Tr(O_2@S_2')$. By induction on the derivation of $T_1 \sqcup T_2 = T'$, $T_1 <: T'$ and $T_2 <: T'$. Therefore by TR_PREFIX, $(m, T).\delta' \in Tr(O_1@S_1)$ and $(m, T).\delta' \in Tr(O_2@S_2)$.

As δ is arbitrary, $Tr(O_3@S) \subseteq Tr(O_1@S_a)$ and $Tr(O_3@S) \subseteq Tr(O_1@S_b)$. By SUB_OBJ, $O_1@S_a <: O_3@S$ and $O_2@S_b <: O_3@S$.

Therefore, $P <: R$ and $Q <: R$, satisfying (1a).

Let R' be a type such that $P <: R'$ and $Q <: R'$. By Lemma A.5.1, $R' = \top$ or $R' = O_4@\overline{S_4}$ such that $Tr(R') \subseteq Tr(P)$ and $Tr(R') \subseteq Tr(Q)$.

If $R' = \top$, by SUB_TOP$R <: R'$, trivially satisfying (1b).

If $R' = O_4@\overline{S_4}$ for some O_4 and $\overline{S_4}$, it must be shown that $Tr(R') \subseteq Tr(R)$. By definition, R' is equivalent to $O_5@S_5 = \bigsqcup\{O_4@S \mid S \in \overline{S_4}\}$ such that $Tr(O_5@S_5) = Tr(R')$. Let $O_5 = (M_5, \Sigma_5, \Delta_5)$.

By showing $Tr(O_5@S_5) \subseteq Tr(R)$, we can establish that $Tr(R') \subseteq Tr(R)$. In order to do this, we prove the following sub-lemma:

For all states $S_a \in \Sigma_1$, $S_b \in \Sigma_2$, $S_c = L(S_a, S_b)$ and $S_d \in \Sigma_5$ such that $O_1@S_a <: O_5@S_d$ and $O_2@S_b <: O_5@S_d$, it follows that if $\delta \in Tr(O_5@S_d)$ then $\delta \in Tr(O_3@S_c)$.

We proceed by induction on the length of t:

- $len(\delta) = 0$, meaning $\delta = \epsilon$. It follows by TR_EMPTY that $\delta \in Tr(O_3@S_c)$.
- $len(\delta) = n + 1$, meaning $\delta = (m, T_d).\delta'$ such that $len(\delta') = n$. By TR_PREFIX, $(S_d, m, T_d', S_d') \in \Delta_5$ for some T_d' and S_d' such that $\delta' \in Tr(O_5@S_d')$ and $T_d' <: T_d$. As $O_1@S_a <: O_5@S_d$ and $O_2@S_b <: O_5@S_d$, it follows by SUB_OBJ that $\delta \in Tr(O_1@S_a)$ and $\delta \in Tr(O_2@S_b)$. By TR_PREFIX, $(S_a, m, T_a', S_a') \in \Delta_1$ and $(S_b, m, T_b', S_b') \in \Delta_2$ for some T_a', S_a', T_b' and S_b' such that $T_a' <: T_d'$, $T_b' <: T_d'$, $\delta' \in Tr(O_1@S_a')$, $\delta' \in Tr(O_2@S_b')$, $O_1@S_a' <: O_5@S_d'$ and $O_2@S_b' <: O_5@S_d'$. By TRANS_JOIN, $(S_c, m, T', S_c') \in \Delta_3$ such that $T_a' \sqcup T_b' = T'$ and $S_c' = L(S_a', S_b')$.

  By induction, $\delta' \in Tr(O_3@S_c')$ where $S_c' = L(S_a', S_b')$. By induction on the derivation of $T_a' \sqcup T_b' = T'$, as $T_a' <: T_d'$ and $T_b' <: T_d'$ it follows that $T' <: T_d'$. By TR_PREFIX, $\delta \in Tr(O_3@S_c)$.

As δ is arbitrary, it follows that $Tr(O_5@S_d) <: Tr(O_3@S_c)$, meaning $O_3@S_c <: O_5@S_d$ by SUB_OBJ. Consequently $R <: R'$, satisfying (1b).

- derived by TY_JOIN_KIND_DIFF. It follows that P is a type and $\diamond = \sqcup$ such that $\mathit{kind}(T) \neq \mathit{kind}(U)$ and $R = \top$.

  By SUB_TOP, $P <: R$ and $Q <: R$, satisfying (1a). Let R' be some other type such that $P <: R'$ and $Q <: R'$.

  It is not possible for $\mathit{kind}(R')$ to be equal to both $\mathit{kind}(P)$ and $\mathit{kind}(Q)$, as that would imply that $\mathit{kind}(P) = \mathit{kind}(Q)$, which is a contradiction. Therefore one (or both) of $\mathit{kind}(P) \neq \mathit{kind}(R')$ and $\mathit{kind}(Q) \neq \mathit{kind}(R')$ hold. By Lemma A.5.3, $R' = \top$, meaning that by SUB_REFL, $R <: R'$, satisfying (1b).

- derived by TY_JOIN_FUN_DIFF_ARITY. It follows that P is a type and $\diamond = \sqcup$ such that $P = (\overrightarrow{E_i}) \rightarrow T$, $Q = (\overrightarrow{F_j}) \rightarrow U$ and $R = \top$ where $\left|\overrightarrow{E_i}\right| \neq \left|\overrightarrow{F_j}\right|$.

  By SUB_TOP, $P <: R$ and $Q <: R$, satisfying (1a).

  Let R' be some other type such that $P <: R'$ and $Q <: R'$. As $P <: R'$, by Lemma A.5.1 either $R' = \top$ or $R' = (\overrightarrow{G_k}) \rightarrow V$ such that $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{G_k}\right|$, $\forall i.\ E_i \leq G_i$ and $T <: V$. If $R' = (\overrightarrow{G_k}) \rightarrow V$, then $\left|\overrightarrow{F_j}\right| \neq \left|\overrightarrow{G_k}\right|$ meaning that $Q <: R'$ cannot hold, which is a contradiction. Therefore, $R' = \top$. By SUB_REFL, $R <: R'$, satisfying (1b).

- derived by TY_JOIN_FUN_DIFF_EFF. It follows that P is a type and $\diamond = \sqcup$, such that $P = (\overrightarrow{E_i}) \rightarrow T$, $Q = (\overrightarrow{F_i}) \rightarrow U$ and $R = \top$ where $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right|$ and there exists an n such that $E_n \sqcup F_n$ is undefined.

  By SUB_TOP, $P <: R$ and $Q <: R$, satisfying (1a). Let R' be some other type such that $P <: R'$ and $Q <: R'$. As $P <: R'$, by Lemma A.5.1 either $R' = \top$ or $R' = (\overrightarrow{G_i}) \rightarrow V$ such that $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{G_i}\right|$, $\forall i.\ E_i \leq G_i$ and $T <: V$. Similarly, as $Q <: R'$, by Lemma A.5.1 if $R' = (\overrightarrow{G_i}) \rightarrow V$ then $\left|\overrightarrow{F_i}\right| = \left|\overrightarrow{G_i}\right|$, $\forall i.\ F_i \leq G_o$ and $U <: V$.

  Assume $R' = (\overrightarrow{G_i}) \rightarrow V$. Consequently, $E_n \leq G_n$ and $F_n \leq G_n$. As $E_n \sqcup F_n$ is undefined, by Lemma A.9.4 there exists no upper bound for E_n and F_n. This contradicts the existence of G_n. Consequently, $R' = \top$. By SUB_REFL, $R <: R'$, satisfying (1b).

- derived by TY_MEET_REFL, meaning P is a type and $\diamond = \sqcap$ such that $P = Q = R$. By SUB_REFL, $R <: P$ and $R <: Q$, satisfying (2a).

  Let $R' <: P$ and $R' <: Q$. By SUB_REFL, $P <: R$, meaning that by SUB_TRANS, $R' <: R$, satisfying (2b).

- derived by TY_MEET_TOP, meaning P is a type and $\diamond = \sqcap$ such that $P = R$ and $Q = \top$ (or vice versa for P and Q). By SUB_REFL, $R <: P$, and by SUB_TOP, $R <: Q$, satisfying (2a).

Let $R' <: P$ and $R' <: Q$. By SUB_REFL, $P <: R$, therefore by SUB_TRANS $R' <: R$, satisfying (2b).

- derived by TY_MEET_FUN, meaning P is a type and $\diamond = \sqcap$, such that $P = (\overrightarrow{E_i}) \to X$, $Q = (\overrightarrow{F_i}) \to Y$ where $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right|$ and $R = (\overrightarrow{G_i}) \to Z$ such that $\forall i.\ E_i \sqcap F_i = G_i$ and $X \sqcap Y = Z$.

  By induction on the derivation of $X \sqcap Y = Z$, it follows that $Z <: X$, $Z <: Y$ and $\forall Z'.\ Z' <: X \wedge Z' <: Y \implies Z' <: Z$.

  For each i, by induction on the derivation of $E_i \sqcap F_i = G_i$, it follows that $G_i \le E_i$, $G_i \le F_i$ and $\forall G_i'.\ G_i' \le E_i \wedge G_i' \le F_i \implies G_i' \le G_i$.

  Consequently by SUB_FN, $R <: P$ and $R <: Q$, satisfying (2a).

  Let R' by a type such that $R' <: P$ and $R' <: Q$. By Lemma A.5.1 it follows that $R' = (\overrightarrow{H_i}) \to A$ for some set of effects $\overrightarrow{H_i}$ and type A such that $A <: X$, $A <: Y$, $\left|\overrightarrow{H_i}\right| = \left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right|$, and for each i, $H_i \le E_i$ and $H_i \le F_i$.

  As $\forall Z'.\ Z' <: X \wedge Z' <: Y \implies Z' <: Z$, it follows that $A <: Z$.

  As $\forall i, G_i'.\ G_i' \le E_i \wedge G_i' \le F_i \implies G_i' \le G_i$, it follows that $\forall i.\ H_i \le G_i$.

  Consequently by SUB_FN, $R' <: R$, satisfying (2b).

- derived by TY_MEET_OBJ, meaning P is a type and $\diamond = \sqcap$ such that $P = O_1@S_1$ where $O_1 = (M_1, \Sigma_1, \Delta_1)$ and $S_1 \in \Sigma_1$, $Q = O_2@S_2$ where $O_2 = (M_2, \Sigma_2, \Delta_2)$ and $S_2 \in \Sigma_2$. R is defined to be $O@S$ such that $O = (M, \Sigma, \Delta)$ where $M = M_1 \cup M_2$, $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{L(S_1, L_t(S_2)) \mid S_1 \in \Sigma_1,\ S_2 \in \Sigma_2\}$, $\Delta = \mathit{left}(\Delta_1) \cup \mathit{right}(\Delta_2) \cup \mathit{mid}(\Delta_1, \Delta_2, S_1, S_2$ and $S = L(S_1, L_t(S_2))$.

  It must be shown that $O@S <: O_1@S_1$ and $O@S <: O_1@S_2$, which is equivalent to $\mathit{Tr}(O_1@S_1) \subseteq \mathit{Tr}(O@S)$ and $\mathit{Tr}(O_2@S_2) \subseteq \mathit{Tr}(O@S)$. In order to do this, we prove the following sub-lemma:

  For all $S_a \in \Sigma_1$, $S_b \in \Sigma_2$ and $S_c = L(S_a, L_t(S_b))$ such that $S_c \in R(\Delta_1, \Delta_2, S_1, S_2)$, if $\delta \in \mathit{Tr}(O_1@S_a)$ then $\delta \in \mathit{Tr}(O@S_c)$.

  We shall prove this by induction on $\mathit{len}(\delta)$:

  - $\mathit{len}(\delta) = 0$. By TR_EMPTY, $\delta \in \mathit{Tr}(O@S_c)$.
  - $\mathit{len}(\delta) = n + 1$, meaning $\delta = (m, T_a).t'$ where $\mathit{len}(\delta') = n$, $(S_a, m, T_a', S_a') \in \Delta_1$ for some $T_a' <: T_a$ and $S_a' \in \Sigma_1$ with $\delta' \in \mathit{Tr}(O@S_a')$.

    There are two cases to consider:

    * There exists a T_b' and S_b' such that $(S_b, m, T_b', S_b') \in \Sigma_2$.
      By TRANS_MEET_MID, $(S_c, m, T_c', S_c') \in \Delta$ where $T_c' = T_a' \sqcap T_b'$ and $S_c' = L(S_a', L_t(S_b'))$.

By REACH_REFL, $S'_c \in R(\Delta_1, \Delta_2, S'_a, S'_b)$. By REACH_ADJ, $S'_c \in R(\Delta_1, \Delta_2, S_1, S_2)$. By induction, $\delta' \in Tr(O@S'_c)$. By induction on the derivation of $T'_a \sqcap T'_b = T'_c$, it follows that $T'_c <: T'_a$. Therefore by TR_PREFIX, $\delta \in Tr(O@S_c)$.

* There does not exist any T'_b or S'_b such that $(S_b, m, T'_b, S'_b) \in \Sigma_2$.
  By TRANS_MEET_LEFT, $(S_c, m, T'_a, S'_a) \in \Delta$. By definition, $O@S'_a \equiv O_1@S'_a$ as all transitions are copied verbatim by TRANS_MEET_COPY_L, therefore $\delta' \in Tr(O@S'_a)$. By TR_PREFIX, $\delta \in Tr(O@S_c)$.

As δ is arbitrary within $O_1@S_a$, it follows that $Tr(O_1@S_a) \subseteq Tr(O@S_c)$. By SUB_OBJ, $O@S_c <: O_1@S_a$. Therefore, $O@S <: O_1@S_1$.

By very similar reasoning (involving TRANS_MEET_RIGHT where a transition for m does not exist in the relevant state in O_1), $O@S <: O_2@S_2$, satisfying (2a).

Let $R' <: O_1@S_1$ and $R' <: O_2@S_2$. It follows that $R' = O_4@S_4$ for some $O_4 = (M_4, \Sigma_4, \Delta_4)$ and $S_4 \in \Sigma_4$. It must be shown that $R' <: O@S$. In order to do this, we shall prove the following sub-lemma:

For all $S_a \in \Sigma_1$, $S_b \in \Sigma_2$ and $S_c = L(S_1, L_t(S_2))$ and $S_d \in \Sigma_4$ such that $O_4@S_d <: O_1@S_a$, $O_4@S_d <: O_4@S_b$ and $S_c \in R(\Delta_1, \Delta_2, S_1, S_2)$, if $\delta \in Tr(O@S_c)$ then $\delta \in Tr(O_4@S_d)$.

We shall prove this by induction on the length of the trace δ:

- $len(\delta) = 0$, meaning $\delta = \epsilon$. By TR_EMPTY, $\delta \in Tr(O_4@S_d)$.
- $len(\delta) = n + 1$, meaning $\delta = (m, T_c).\delta'$ such that $len(\delta') = n$, $(S_c, m, T'_c, S'_c) \in \Delta$ where $T'_c <: T_c$ and $\delta' \in Tr(O@S'_c)$.

  There are three possible derivations of $(S_c, m, T'_c, S'_c) \in \Delta$:

  * by TRANS_MEET_LEFT, such that $(S_a, m, T'_c, S'_c) \in \Delta_1$ and there does not exist a T_b or S'_b such that $(S_b, m, T_b, S'_b) \in \Delta_2$. By definition, $O@S'_c \equiv O_1@S'_c$ as all transitions are copied verbatim by TRANS_MEET_COPY_L. Therefore $\delta' \in Tr(O_1@S'_c)$. By TR_PREFIX, $\delta \in Tr(O_1@S_a)$. As $O_4@S_d <: O_1@S_a$, by SUB_OBJ, $\delta \in Tr(O_4@S_d)$.
  * by TRANS_MEET_RIGHT. By very similar reasoning to the TRANS_MEET_LEFT case, $\delta \in Tr(O_4@S_d)$.
  * by TRANS_MEET_MID, such that there exists types T_a and T_b and states S'_a and S'_b such that $(S_a, m, T_a, S'_a) \in \Delta_1$ and $(S_b, m, T_b, S'_b) \in \Delta_2$ where $T_a \sqcap T_b = T'_c$ and $S'_c = L(S'_a, L_t(S'_b))$.
    By REACH_REFL, $S'_c \in R(\Delta_1, \Delta_2, S'_a, S'_b)$.
    By REACH_ADJ, $S'_c \in R(\Delta_1, \Delta_2, S_1, S_2)$.

As $O_4@S_d <: O_1@S_a$, it follows by Lemma A.5.4 that there exists a T_d and S'_d such that $(S_d, m, T_d, S'_d) \in \Delta_4$, $T_d <: T_a$ and $O_4@S'_d <: O_1@S'_a$.
As $O_4@S_d <: O_2@S_b$, it follows by Lemma A.5.4 that $T_d <: T_b$ and $O_4@S'_d <: O_2@S'_b$.
By induction, $\delta' \in \mathit{Tr}(O_4@S'_d)$.
By induction on the derivation of $T_a \sqcap T_b = T'_c$, it follows that $\forall T'.\ T' <: T_a \wedge T'T_b \implies T' <: T'_c$. Therefore, $T_d <: T'_c$. By SUB_TRANS, $T_d <: T_c$. Therefore by TR_PREFIX, $\delta \in \mathit{Tr}(O_4@S_d)$.

Therefore, $\mathit{Tr}(O@S_c) \subseteq \mathit{Tr}(O_4@S_d)$, meaning $O_4@S_d <: O@S_c$ by SUB_OBJ.

Consequently, $O_4@S_4 <: O@S$, satisfying (2b).

- derived by EFF_JOIN_UP, meaning P is an effect and $\diamond = \sqcup$, such that $P = T_1 \ggg U_1$, $Q = T_2 \ggg U_2$ and $R = T \ggg U$ where $T_1 \sqcap T_2 = T$ and $U_1 \sqcup U_2 = U$.

  By induction on the derivation of $T_1 \sqcap T_2 = T$, it follows that $T <: T_1$, $T <: T_2$ and $\forall T'.\ T' <: T_1 \wedge T' <: T_2 \implies T' <: T$. By induction on the derivation of $U_1 \sqcup U_2 = U$, it follows that $U_1 <: U$, $U_2 <: U$ and $\forall U'.\ U_1 <: U' \wedge U_2 <: U' \implies U <: U'$.

  Consequently by SUB_UP_EFF, $P \leq R$ and $Q \leq R$, satisfying property (3a).

  Let R' be an effect such that $P \leq R'$ and $Q \leq R'$. By Lemma A.8.1, $R' = T_3 \ggg U_3$ for some T_3 and U_3 such that $T_3 <: T_1$, $T_3 <: T_2$, $U_1 <: U_3$ and $U_2 <: U_3$.

  As $\forall T'.\ T' <: T_1 \wedge T' <: T_2 \implies T' <: T$, it follows that $T_3 <: T$.
  As $\forall U'.\ U_1 <: U' \wedge U_2 <: U' \implies U <: U'$, it follows that $U <: U_3$.

  Therefore by SUB_UP_EFF, $R <: R'$, satisfying (3b).

- derived by EFF_JOIN_FL, meaning P is an effect and $\diamond = \sqcup$, such that $P = T_1 \overset{\overline{\delta_1}}{\gg} U_1$, $Q = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ and $R = T \overset{\overline{\delta}}{\gg} U$ such that $T = T_1 \sqcap T_2$, $\overline{\delta} = \overline{\delta_1} \cup \overline{\delta_2}$ and $U = \bigsqcup\{\mathit{trmap}(T, \delta) \mid \delta \in \overline{\delta}\}$.

  By induction on the derivation of $T_1 \sqcap T_2 = T$, it follows that $T <: T_1$, $T <: T_2$ and $\forall T'.\ T' <: T_1 \wedge T' <: T_2 \implies T' <: T$. Trivially, $\overline{\delta_1} \subseteq \overline{\delta}$ and $\overline{\delta_2} \subseteq \overline{\delta}$. Therefore, $P \leq R$ and $Q \leq R$, satisfying (3a).

  Let R' be an effect such that $P \leq R'$ and $Q \leq R'$. By Lemma A.8.1 it follows that $R' = T_3 \overset{\overline{\delta_3}}{\gg} U_3$ such that $T_3 <: T_1$, $T_3 <: T_2$, $\overline{\delta_1} \subseteq \overline{\delta_3}$ and $\overline{\delta_2} \subseteq \overline{\delta_3}$. Consequently, $T_3 <: T$ and $\overline{\delta_1} \cup \overline{\delta_2} \subseteq \overline{\delta_3}$. Therefore by SUB_FL_EFF, $R' \leq R$, satisfying (3b).

- derived by EFF_MEET_UP, meaning P is an effect and $\diamond = \sqcap$, such that $P = T_1 \ggg U_1$, $Q = T_2 \ggg U_2$ and $R = T \ggg U$ where $T_1 \sqcup T_2 = T$ and $U_1 \sqcap U_2 = U$.

  By induction on the derivation of $T_1 \sqcup T_2 = T$, it follows that $T_1 <: T$, $T_2 <: T$ and $\forall T'.\ T_1 <: T' \wedge T_2 <: T' \implies T <: T'$.

By induction on the derivation of $U_1 \sqcap U_2 = U$, it follows that $U <: U_1$, $U <: U_2$ and $\forall U'.\, U' <: U_1 \wedge U' <: U_2 \implies U' <: U$.

By SUB_UP_EFF, $R \leq P$ and $R \leq Q$, satisfying (4a).

Let R' be an effect such that $R' \leq P$ and $R' \leq Q$. By Lemma A.8.1, $R' = T_3 \ggg U_3$ such that $T_1 <: T_3$, $T_2 <: T_3$, $U_3 <: U_1$ and $U_3 <: U_2$.

As $\forall T'.\, T_1 <: T' \wedge T_2 <: T' \implies T <: T'$, it follows that $T <: T_3$. As $\forall U'.\, U' <: U_1 \wedge U' <: U_2 \implies U' <: U$, it follows that $U_3 <: U$. Consequently by SUB_UP_EFF, $R' \leq R$, satisfying (4b).

- derived by EFF_MEET_FL, meaning P is an effect and $\diamond = \sqcup$, such that $P = T_1 \overset{\overline{\delta_1}}{\gg} U_1$ and $Q = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ and $R = T \overset{\overline{\delta}}{\gg} U$ such that $T_1 \sqcup T_2 = T$, $\overline{\delta} = \overline{\delta_1} \cap \overline{\delta_2} \supset \varnothing$ and $U = \bigsqcup\{trmap(T, \delta) \mid \delta \in \overline{\delta}\}$.

  By induction on the derivation of $T_1 \sqcup T_2 = T$, it follows that $T_1 <: T$, $T_2 <: T$ and $\forall T'.\, T_1 <: T' \wedge T_2 <: T' \implies T <: T'$.

  By definition, $\overline{\delta} \subset \overline{\delta_1}$ and $\overline{\delta} \subset \overline{\delta_2}$. Consequently by SUB_FL_EFF, $R \leq P$ and $R \leq Q$, satisfying (4a).

  Let $R' \leq P$ and $R' \leq Q$. By Lemma A.8.1, $R = T_3 \overset{\overline{\delta_3}}{\gg} U_3$ such that $T_1 <: T_3$, $T_2 <: T_3$, $\overline{\delta_3} \subseteq \overline{\delta_1}$ and $\overline{\delta_3} \subseteq \overline{\delta_2}$, meaning $\overline{\delta_3} \subseteq \overline{\delta_1} \cap \overline{\delta_2}$. Consequently, $T <: T_3$ and $\overline{\delta_3} \subseteq \overline{\delta}$, meaning $R' \leq R$ by SUB_FL_EFF, satisfying (4b).

□

Lemma A.9.2. *For all types T and U, $T \sqcup U$ is defined.*

Proof. If $T = U$, then $T \sqcup U = T$ by TY_JOIN_REFL. If $T \neq U$, we proceed induction on the structure of T:

- $T = \top$. As $T \neq U$, it follows that $kind(T) \neq kind(U)$ and $T \sqcup U = \top$ by TY_JOIN_KIND_DIFF.

- $T = \mathbf{Unit}$. As $T \neq U$, it follows that $kind(T) \neq kind(U)$ and $T \sqcup U = \top$ by TY_JOIN_KIND_DIFF.

- $T = \mathbf{Bool}$. As $T \neq U$, it follows that $kind(T) \neq kind(U)$ and $T \sqcup U = \top$ by TY_JOIN_KIND_DIFF.

- $T = (\overrightarrow{E_i}) \to V$ for some $\overrightarrow{E_i}$ and V. If U is not a function type, then $kind(T) \neq kind(U)$ and $T \sqcup U = \top$ by TY_JOIN_KIND_DIFF. If $U = (\overrightarrow{F_j}) \to W$ for some $\overrightarrow{F_j}$ and W, there are three possibilities:

- $\left|\overrightarrow{E_i}\right| \neq \left|\overrightarrow{F_j}\right|$. It follows that $T \sqcup U = \top$ by TY_JOIN_FUN_DIFF_ARITY.
- $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_j}\right|$, but there exists an n such that $E_i \sqcup F_i$ is undefined. It follows that $T \sqcup U = \top$ by TY_JOIN_FUN_DIFF_EFF.
- $\left|\overrightarrow{E_i}\right| = \left|\overrightarrow{F_i}\right|$ and $E_i \sqcup F_i = G_i$ is defined for all i. By induction, $V \sqcup W = X$ is defined. It follows by TY_JOIN_FUN that $T \sqcup U = (\overrightarrow{G_i}) \to X$.

• $T = O_1@S_1$, where $O_1 = (M_1, \Sigma_1, \Delta_1)$ and $S_1 \in \Sigma_1$. If U is not an object type, then $kind(T) \neq kind(U)$ and $T \sqcup U = \top$ by TY_JOIN_KIND_DIFF. Let $U = O_2@S_2$ where $O_2 = (M_2, \Sigma_2, \Delta_2)$ and $S_2 \in \Sigma_2$.

  Let $(S_a, m, T_a, S'_a) \in \Delta_1$ and $(S_b, m, T_b, S'_b) \in \Delta_2$. By induction, $T_a \sqcup T_b$ is defined. As the transitions were arbitrary, $\Delta_3 = \Delta_1 \sqcup \Delta_2$ is defined. It follows that $O_1@S_1 \sqcup O_2@S_2$ is defined.

  In the case where $T = O@\overline{S}$, by definition, $O@\overline{S} = \bigsqcup\{O@S \mid S \in \overline{S}\}$, which by repeated induction is defined such that $O@\overline{S} = O'@S'$. Therefore, $T \sqcup U$ is defined for object types with state sets.

□

Lemma A.9.3. *For all P and Q such that $P \sqcap Q$ is undefined, there exists no lower bound of P and Q.*

Proof. Let R be a lower bound of P and Q. We proceed by induction on the structure of P.

• $P = \textbf{Unit}$ or $P = \textbf{Bool}$. By Lemma A.5.1, it follows that $R = P$. By Lemma A.5.2, either $Q = P$ or $Q = \top$ by SUB_REFL or SUB_TOP. If $Q = P$, then $P \sqcap Q$ is defined by TY_MEET_REFL, which is a contradiction. If $Q = \top$, then $P \sqcap Q$ is defined by TY_MEET_TOP, which is a contradiction.

• $P = (\overrightarrow{E_i}) \to R$. By Lemma A.5.1, $R = (\overrightarrow{F_i}) \to W$ such that $R <: W$ and $\forall i.F_i \leq E_i$. By Lemma A.5.2, $Q = \top$ or $Q = (\overrightarrow{G_i}) \to X$ such that $R <: X$ and $\forall i.F_i \leq G_i$. If $Q = \top$, then $P \sqcap Q$ is defined by TY_MEET_TOP, which is a contradiction. Therefore $Q = (\overrightarrow{G_i}) \to X$.

  As $P \sqcap Q$ is undefined, it follows that either $W \sqcap X$ is undefined or $E_i \sqcap G_i$ is undefined. By induction, if $W \sqcap X$ is undefined then there exists no lower bound of W and X, which contradicts the existence of R. By Lemma A.9.3 if $E_i \sqcap G_i$ is undefined then there exists no lower bound of E_i and G_i, which contradicts the existence of F_i and R.

- $P = O_1@S_1$. By Lemma A.5.1, $R = O@S$ such that $Tr(P) \subseteq Tr(R)$. By Lemma A.5.2, $Q = \top$ or $Q = O_2@S_2$ such that $Tr(Q) \subseteq Tr(R)$. If $Q = \top$ then $P \sqcap Q$ is defined by TY_MEET_TOP, which is a contradiction. Therefore, $Q = O_2@S_2$.

  Let $O_1 = (M_1, \Sigma_1, \Delta_1)$ $O_2 = (M_2, \Sigma_2, \Delta_2)$, and $O = (M, \Sigma, \Delta)$.

  As $P \sqcap Q$ is undefined, there exists some reachable state $S' \in R(\Delta_1, \Delta_2, S_1, S_2)$ where $S' = L(S_a, L_t(S_b))$ for $S_a \in \Sigma_1$ and $S_b \in \Sigma_2$ such that there exists two transitions $(S_a, m, W_1, S'_a) \in \Delta_1$ and $(S_b, m, W_2, S'_b) \in \Delta_2$ where $W_1 \sqcap W_2$ is undefined. By induction, there is no lower bound for W_1 and W_2.

  By Lemma A.9.5, there exists a trace δ such that $trmap(P, \delta) = O_1@S_a$ and $trmap(Q, \delta) = O_2@S_b$. As $R <: P$, $\delta \in Tr(R)$ and therefore $trmap(P, \delta) = O@S'$ for some S'. By Lemma A.4.6, $O@S' <: O_1@S_a$ and $O@S' <: O_2@S_b$.

  Let $\delta_1 \in Tr(O_1@S'_a)$ and $\delta_2 \in Tr(O_2@S'_b)$. By TR_PREFIX, $(m, W_1).\delta_1 \in Tr(O_1@S_a)$ and $(m, W_2).\delta_2 \in Tr(O_2@S_b)$. By SUB_OBJ, both traces must also be contained in $Tr(O@S')$, meaning there exists some W and S'' such that $(S', m, W, S'') \in \Delta$ where $W <: W_1$ and $W <: W_2$. This is a contradiction, as there is no lower bound of W_1 and W_2.

- $P = T_1 \ggg U_1$. Consequently, $Q = T_2 \ggg U_2$ and $R = T \ggg U$ such that $T_1 <: T$, $T_2 <: T$, $U <: U_1$ and $U <: U_2$.

  In order for $P \sqcap Q$ to be undefined, it must be the case that $T_1 \sqcup T_2$ is undefined or $U_1 \sqcap U_2$ is undefined. By Lemma A.9.2, $T_1 \sqcup T_2$ is always defined, therefore $U_1 \sqcap U_2$ is undefined.

  By induction, there is no lower bound of U_1 and U_2, which contradicts the existence of U.

- $P = T_1 \overset{\overline{\delta_1}}{\gg} U_1$. Consequently, $Q = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ and $R = T \overset{\overline{\delta}}{\gg} U$ such that $T_1 <: T$, $T_2 <: T$, $\varnothing \subset \overline{\delta} \subseteq \overline{\delta_2}$ and $\varnothing \subset \overline{\delta} \subseteq \overline{\delta_1}$. Consequently, $\varnothing \subset \overline{\delta} \subseteq \overline{\delta_1} \cap \overline{\delta_2}$.

  In order for $P \sqcap Q$ to be undefined, $T_1 \sqcup T_2$ is undefined or $\overline{\delta_1} \cap \overline{\delta_2}$ is empty. By Lemma A.9.2, $T_1 \sqcup T_2$ is always defined, therefore $\overline{\delta_1} \cap \overline{\delta_2}$ is empty, which is a contradiction.

□

Lemma A.9.4. *Let E and F be effects such that $E \sqcup F$ is not defined. It follows that there exists no upper bound for E and F.*

Proof. by contradiction. Let G be an effect such that $E \leq G$ and $F \leq G$. By Lemma A.8.1, there are two cases to consider:

- $E = T_1 \overset{\overline{\delta_1}}{\gg} U_1$, $F = T_2 \overset{\overline{\delta_2}}{\gg} U_2$ and $G = T_3 \overset{\overline{\delta_3}}{\gg} U_3$ such that $T_3 <: T_1$, $T_3 <: T_2$, $\overline{\delta_1} \subseteq \overline{\delta_3}$ and $\overline{\delta_2} \subseteq \overline{\delta_3}$.

  As $E \sqcup F$ is undefined, it follows that $T_1 \sqcap T_2$ is undefined. By Lemma A.9.3, there exists no lower bound of T_1 and T_2, which contradicts the existence of T_3 and G.

- $E = T_1 \gg U_2$, $F = T_2 \gg U_2$ and $G = T_3 \gg U_3$ such that $T_3 <: T_1$, $T_3 <: T_2$, $U_1 <: U_3$ and $U_2 <: U_3$.

  As $E \sqcup F$ is undefined, it follows that $T_1 \sqcap T_2$ is undefined or $U_1 \sqcup U_2$ is undefined. By Lemma A.9.2, $U_1 \sqcup U_2$ is always defined. Therefore $T_1 \sqcap T_2$ is undefined. By Lemma A.9.3, there exists no lower bound of T_1 and T_2, which contradicts the existence of T_3 and G.

□

Lemma A.9.5. *For all $O_1@S_1$ and $O_2@S_2$ such that $L(S_1', L_t(S_2')) \in R(\Delta_1, \Delta_2, S_1, S_2)$, there exists a trace δ such that $trmap(O_1@S_1, \delta) = O_1@S_1'$ and $trmap(O_2@S_2, \delta) = O_2@S_2'$.*

Proof. By induction on the derivation of $L(S_1', L_t(S_2')) \in R(\Delta_1, \Delta_2, S_1, S_2)$:

- by REACH_REFL. It follows that $S_1 = S_1'$ and $S_2 = S_2'$. Let $\delta = \epsilon$. By TR_MAP_EMPTY, $trmap(O_1@S_1, \delta) = O_1@S_1'$ and $trmap(O_2@S_2, \delta) = O_2@S_2'$.

- by REACH_ADJ. It follows that there exists two transitions $(S_a, m, T_a, S_a') \in \Delta_1$ and $(S_b, m, T_b, S_b') \in \Delta_2$ such that $L(S_a, L_t(S_b)) \in R(\Delta_1, \Delta_2, S_1, S_2)$ and $L(S_1, L_t(S_2')) \in R(\Delta_1, \Delta_2, S_a', S_b')$.

  By induction, there exists a trace δ_l such that $trmap(O_1@S_1, \delta_l) = O_1@S_a$ and $trmap(O_2@S_2, \delta_l) = O_2@S_b$. Also by induction, there exists a trace δ_r such that $trmap(O_1@S_a', \delta_r) = O_1@S_1'$ and $trmap(O_2@S_b', \delta_r) = O_2@S_2'$.

  Let $\delta_r' = (m, \top).\delta_r$. By TR_MAP_PREFIX, $trmap(O_1@S_a, \delta_r') = O_1@S_1'$ and $trmap(O_2@S_b, \delta'') = O_2@S_2'$.

  Let $\delta = \delta_l + \delta_r'$. By Lemma A.4.8, $trmap(O_1@S_1, \delta) = trmap(trmap(O_1@S_1, \delta_l), \delta_r')$, which is equivalent to $trmap(O_1@S_1, \delta) = O_1@S_1'$. Similarly, $trmap(O_2@S_2, \delta) = O_2@S_2'$.

□

A.10 Properties of typings

Lemma A.10.1. *Let* $v = \lambda(\overrightarrow{x_i : E_i}).t$ *and* $T = (\overrightarrow{E'_i}) \rightarrow V$ *such that* $\varnothing \rhd v : T \lhd \varnothing$. *It follows that* $\forall i.\ E_i \leq E'_i$*, and that* $\overline{x_i : in(E'_i)} \rhd t : V \lhd \overline{x_i : out(E'_i)}$.

Proof. By induction on the derivation of the typing of v:

- Derived by T_FUN_FL_DEF, meaning $\overline{x_i : in(E_i)} \rhd t : V \lhd \overline{x_i : out(E_i)}$ and $\forall i. E_i = E'_i$. By Lemma A.8.2, $\forall i.\ E_i \leq E'_i$. Directly, $\overline{x_i : in(E'_i)} \rhd t : V \lhd \overline{x_i : out(E_i)}$.
- Derived by T_SUB. It follows that there exists a $T' <: T$ such that $\varnothing \rhd v : T' \lhd \varnothing$. By Lemma A.5.1, $T' = (\overrightarrow{E''_i}) \rightarrow V''$ such that $\forall i.\ E''_i \leq E'_i$ and $V'' <: V$. By induction, $\forall i.\ E_i \leq E''_i$ and $\overline{x_i : in(E''_i)} \rhd t : V'' \lhd \overline{x_i : out(E''_i)}$. By application of T_SUBand repeated application of T_WIDEN_FL_EFFor T_WIDEN_UP_EFFas appropriate, $\overline{x_i : in(E'_i)} \rhd t : V \lhd \overline{x_i : out(E_i)}$. By SUB_EFF_TRANS, $\forall i.\ E_i \leq E'_i$.

□

Lemma A.10.2. *For any judgement* $\Gamma \rhd t : T \lhd \Gamma'$, $dom(\Gamma) = dom(\Gamma')$.

Proof. Straightforward induction on the typing derivation of t. □

Lemma A.10.3. *Let* v *be a value. It follows that for any typing* $\Gamma \rhd v : T \lhd \Gamma'$ *that* $\Gamma = \Gamma'$.

Proof. By induction on the typing derivation.

- v is typed by one of T_UNIT, T_TRUE, T_FALSE, T_OBJECT or T_FUN_FL_DEF. Directly, $\Gamma = \Gamma'$.
- v is typed by T_SUB. It follows that there $\Gamma \rhd t : U \lhd \Gamma'$ where $U <: T$. By induction, $\Gamma = \Gamma'$.

□

Lemma A.10.4. *Let* v *be a value such that* $\Gamma \rhd v : T \lhd \Gamma'$. *Let* Γ'' *be an arbitrary context. It follows that* $\Gamma'' \rhd v : T \lhd \Gamma''$.

Proof. As v is a value, it can be typed by one of T_TRUE, T_FALSE, T_UNIT, T_FUN_FL_DEF, T_FUN_UP_DEF or T_OBJECT. In each of these rules, the only requirement of Γ and Γ' is that they are equal. Therefore, $\Gamma'' \rhd t : T \lhd \Gamma''$ is a valid typing by one of these rules. □

Lemma A.10.5 (Substitution). *If* $\Gamma \rhd t : T \lhd \Gamma'$*, then* $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$ *where each* x_i *and* y_i *is distinct, and* $\overline{x_i} \cap \overline{y_i} = \varnothing$.

Proof. by induction on the typing derivation of t. $\Gamma \rhd t : T \lhd \Gamma'$ must have been derived by one of the following rules:

- T_UNIT, T_TRUE, T_FALSE, T_OBJECT, T_FUN_FL_DEF or T_FUN_UP_DEF. For each of these rules, $\Gamma = \Gamma'$. Therefore, $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} = \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$. and substitution has no effect on the value. The rule used for derivation has no specific requirements of either Γ or Γ' other than that they are equal. Therefore, $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd v\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_LET. It follows that $t = \textbf{let } x = t_v \textbf{ in } t_b$. Substitution is defined on this term such that

  $t\{\overline{{}^{x_i}/_{y_i}}\} \equiv \textbf{let } x = (t_v\{\overline{{}^{x_i}/_{y_i}}\}) \textbf{ in } (t_b\{\overline{{}^{x_j}/_{y_j}}\})$, where $\overline{y_j} = \overline{y_i} - \{x\}$.

  By rule T_LET, it follows that there exists T_v, T_v', Γ_1 such that $\Gamma \rhd t_v : T_v \lhd \Gamma_1$ and that $\Gamma_1, x : T_v \rhd t_b : T \lhd \Gamma', x : T_v'$. Additionally, $x \notin \mathit{dom}(\Gamma)$.

  By induction, t_v can be substituted such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_v\{\overline{{}^{x_i}/_{y_i}}\} : T_v \lhd \Gamma_1\{\overline{{}^{x_i}/_{y_i}}\}$. Additionally, t_b can be substituted such that $(\Gamma_1\{\overline{{}^{x_j}/_{y_j}}\}), x : T_v \rhd t_b\{\overline{{}^{x_j}/_{y_j}}\} : T \lhd \{(\overline{x_j/y_j}\}\Gamma'), x : T_v'$ as it is guaranteed that $x \notin \overline{y_j}$. As $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma)$, and that $x \notin \mathit{dom}(\Gamma)$, it follows that $\Gamma_1\{\overline{{}^{x_i}/_{y_i}}\} = (\Gamma_1\{\overline{{}^{x_j}/_{y_j}}\})$ and that $\overline{x_i/y_i}\}\Gamma' = \overline{x_j/y_j}\}\Gamma'$.

  Therefore, the requirements of T_LETare satisfied such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_SEQ. It follows that $t = t_l; t_r$, and there exists Γ'',T' such that $\Gamma \rhd t_l : T' \lhd \Gamma''$ and $\Gamma'' \rhd t_r : T \lhd \Gamma'$. Substitution is defined on this term such that $(t_l; t_r)\{\overline{{}^{x_i}/_{y_i}}\} \equiv t_l\{\overline{{}^{x_i}/_{y_i}}\}; t_r\{\overline{{}^{x_i}/_{y_i}}\}$.

  By induction, t_l and t_r can be typed such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_l\{\overline{{}^{x_i}/_{y_i}}\} : T' \lhd \Gamma''\{\overline{{}^{x_i}/_{y_i}}\}$ and $\Gamma''\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_r\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

  Therefore, t can by typed using T_SEQsuch that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_FUN_CALL. It follows that $t = z(\overrightarrow{z_k})$ and that $\Gamma = \Gamma_1, \overline{z_k : T_k}$, $\Gamma' = \Gamma_1, \overline{z_k : T_k'}$ where $\Gamma_1(z) = (\overrightarrow{E_k}) \to T$ with for each k, $T_k <: U_k$ and $T_k' = \mathit{remap}(T_k, E_k)$. Each variable in $\overline{z_k} \cup \{z\}$ is distinct.

  Substitution is defined on this term such that $z(\overrightarrow{z_k})\{\overline{{}^{x_i}/_{y_i}}\} \equiv z'(\overrightarrow{z_k'})$ where $z' = z$ if $z \notin \overline{y_i}$ or $z = x_i$ for the x_i such that $z = y_i$, and similarly for all z_i.

  The uniqueness of each substitution pair ensures that no variable will be passed twice as part of the function call. Therefore, the conditions of T_FUN_CALLare satisfied such that

  $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_METH_CALL. It follows that $t = x.m$ with $\Gamma = \Gamma_1, x : O@\overline{S}$ and $\Gamma' = \Gamma_1, x : O@\overline{S'}$ such that $m : T \Rightarrow \overline{S'} \in O@\overline{S}$.

  Substitution is defined on this term such that $x.m\{\overline{{}^{x_i}/_{y_i}}\} \equiv z.m$ where $z = x_j$ if $x = y_j$ for some $y_j \in \overline{y_i}$. Otherwise, $z = x$.

  The conditions of T_METH_CALL are satisfied such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_IF. It follows that $t = \textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f$, where there exists Γ_1, Γ_2, Γ_3, T_t, T_f such that $\Gamma \rhd t_c : \textbf{Bool} \lhd \Gamma_1$ and that $\Gamma_1 \rhd t_t : T_t \lhd \Gamma_2$ and that $\Gamma_1 \rhd t_f : T_f \lhd \Gamma_3$ where $\Gamma' = \Gamma_2 \sqcap \Gamma_3$ and $T = T_t \sqcup T_f$.

  Substitution is defined on this term such that
  $(\textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f)\{\overline{{}^{x_i}/_{y_i}}\} \equiv \textbf{if } t_c\{\overline{{}^{x_i}/_{y_i}}\} \textbf{ then } t_t\{\overline{{}^{x_i}/_{y_i}}\} \textbf{ else } t_f\{\overline{{}^{x_i}/_{y_i}}\}$.

  By induction substitution can be performed on t_c, t_t and t_f such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_c\{\overline{{}^{x_i}/_{y_i}}\} : \textbf{Bool} \lhd \Gamma_1\{\overline{{}^{x_i}/_{y_i}}\}$ and that $\Gamma_1\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_t\{\overline{{}^{x_i}/_{y_i}}\} : T_t \lhd \Gamma_2\{\overline{{}^{x_i}/_{y_i}}\}$ and $\Gamma_1\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_f\{\overline{{}^{x_i}/_{y_i}}\} : T_f \lhd \Gamma_3\{\overline{{}^{x_i}/_{y_i}}\}$. By the definition of substitution on contexts, $\Gamma_2\{\overline{{}^{x_i}/_{y_i}}\} \sqcap \Gamma_3\{\overline{{}^{x_i}/_{y_i}}\} = \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

  The conditions of the rule T_IF are satisfied such that
  $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd \textbf{if } t_c \textbf{ then } t_t \textbf{ else } t_f\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_WHILE_A. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$. Contexts Γ_1 through Γ_4 exist where $dom(\Gamma_1) = dom(\Gamma_2)$ and $dom(\Gamma_3) = dom(\Gamma_4)$, and there exists a type T_b such that $\Gamma_1 \rhd t_c : \textbf{Bool} \lhd \Gamma_2$ and $\Gamma_3 \rhd t_b : T_b \lhd \Gamma_4$.

  Let $E_c(x) = extract(x, \Gamma_1, \Gamma_2)$ and $E_b(x) = extract(x, \Gamma_3, \Gamma_4)$.

  Finally, $\Gamma' = \{x : remap(\Gamma(x), E_c(x) \cdot (E_b(x) \cdot E_c(x)){*}) \mid x \in dom(\Gamma_1)\}$ and $T = \textbf{Unit}$.

  Substitution is defined on this term such that
  $(\textbf{while } t_c \textbf{ do } t_b)\{\overline{{}^{x_i}/_{y_i}}\} \equiv \textbf{while } t_c\{\overline{{}^{x_i}/_{y_i}}\} \textbf{ do } t_b\{\overline{{}^{x_i}/_{y_i}}\}$.

  Let $\Gamma'_1 = \Gamma_1\{\overline{{}^{x_i}/_{y_i}}\}$, $\Gamma'_2 = \Gamma_2\{\overline{{}^{x_i}/_{y_i}}\}$, $\Gamma'_3 = \Gamma_3\{\overline{{}^{x_i}/_{y_i}}\}$ and $\Gamma'_4 = \Gamma_4\{\overline{{}^{x_i}/_{y_i}}\}$.

  By induction, t_c and t_b can by typed such that $\Gamma'_1 \rhd t_c\{\overline{{}^{x_i}/_{y_i}}\} : \textbf{Bool} \lhd \Gamma'_2$ and that $\Gamma'_3 \rhd t_b\{\overline{{}^{x_i}/_{y_i}}\} : T_b \lhd \Gamma'_4$.

  Let $\Gamma'_5 = \Gamma\{\overline{{}^{x_i}/_{y_i}}\}$. Let $E'_c(x) = extract(x, \Gamma'_1, \Gamma'_2)$ and $E'_b(x) = extract(x, \Gamma'_3, \Gamma'_4)$. Let $\Gamma'_6 = \{x : remap(\Gamma(x), E'_c(x) \cdot (E'_b(x) \cdot E'_c(x)){*}) \mid x \in dom(\Gamma'_5)\}$, which is equivalent to $\Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

  By T_WHILE_A, $\Gamma'_5 \rhd \textbf{while } t_c \textbf{ do } t_b\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'_6$, which is equivalent to $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd \textbf{while } t_c \textbf{ do } t_b\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_WHILE_B. It follows that $t = \textbf{while } t_c \textbf{ do } t_b$ and there exists a $\Gamma'' \geq \Gamma$ such that $\Gamma \rhd t_c : \textbf{Bool} \lhd \Gamma'$ and $\Gamma' \rhd t_b : T \lhd \Gamma''$.

  By induction, t_c and t_b can be typed such that $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_c\{\overline{{}^{x_i}/_{y_i}}\} : \textbf{Bool} \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$ and $\Gamma'\{\overline{{}^{x_i}/_{y_i}}\} \rhd t_c\{\overline{{}^{x_i}/_{y_i}}\} : \textbf{Bool} \lhd \Gamma''\{\overline{{}^{x_i}/_{y_i}}\}$.

  Therefore by T_WHILE_B, $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t : \textbf{Unit} \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_SUB. It follows that $\Gamma \rhd t : T' \lhd \Gamma'$ for some $T' <: T$. By induction, $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T' \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$. By rule T_SUB, $\Gamma\{\overline{{}^{x_i}/_{y_i}}\} \rhd t\{\overline{{}^{x_i}/_{y_i}}\} : T \lhd \Gamma'\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_WIDEN_FL_EFF. It follows that $\Gamma = \Gamma_1, x : U$ and $\Gamma' = \Gamma_2, x : V$ and there exists a U' and V' such that $\Gamma_1, x : U' \rhd t : T \lhd \Gamma_2, x : V'$ and $U' \gg V' \leq U \gg V$. By induction, $\Gamma_1, x : U'\{\overline{{}^{x_i}/_{y_i}}\} \rhd t : T \lhd \Gamma_2, x : V'\{\overline{{}^{x_i}/_{y_i}}\}$. By T_WIDEN_FL_EFF, $\Gamma_1, x : U\{\overline{{}^{x_i}/_{y_i}}\} \rhd t : T \lhd \Gamma_2, x : V\{\overline{{}^{x_i}/_{y_i}}\}$.

- T_WIDEN_UP_EFF— similar reasoning to T_WIDEN_FL_EFF.

□

A.10.1 Flow effect specific properties

Definition A.10.1 (Required properties of effect combinators)**.** The following properties of effect combinators are required and asserted due to the lack of a formal definition:

- If $T \gg U \cdot V \gg W = X \gg Y$, and $\mathit{remap}(X, T \gg U) = Z$, then $T \gg U \leq X \gg Z$ and $V \gg W \leq Z \gg Y$.
- If $T \gg U \mid V \gg W = X \gg Y$, then $T \gg U \leq X \gg Y$ and $V \gg W \leq X \gg Y$.
- $\mathit{remap}(T, (U \gg V)*) = T \sqcup \mathit{remap}(W, (U \gg V)*)$ where $W = \mathit{remap}(T, U \gg V)$.
- $\mathit{remap}(T, E \cdot F) <: \mathit{remap}(\mathit{remap}(T, E), F)$.

▲

Lemma A.10.6 (Upgrading, with flow effects)**.** *Let t be a term such that $\Gamma_1 \rhd t : U \lhd \Gamma_3$. If $\Gamma_2 \geq \Gamma_1$, then for $\Gamma_4 = \{\mathit{remap}(\Gamma_2(x), \Gamma_1(x) \gg \Gamma_3(x)) \mid x \in \mathit{dom}(\Gamma_3)\} \cup \{\Gamma_2(x) \mid x \notin \mathit{dom}(\Gamma_3)\}$, it follows that $\Gamma_2 \rhd t : U \lhd \Gamma_4$. Additionally, $\Gamma_4 \geq \Gamma_3$ and $\forall x \in \mathit{dom}(\Gamma_3).\Gamma_1(x) \gg \Gamma_3(x) \leq \Gamma_2(x) \gg \Gamma_4(x)$.*

Proof. As $\Gamma_2 \geq \Gamma_1$, there exists a Γ_2' such that $\Gamma_2 = \Gamma_2', \overline{x_i : T_i}$ where $dom(\Gamma_2') = dom(\Gamma_1)$ and $\forall x \in dom(\Gamma_1).\Gamma_2'(x) <: \Gamma_1'(x)$. The set $\overline{x_i : T_i}$ may be empty.

By Lemma A.6.1, $\forall x \in dom(\Gamma_3).remap(\Gamma_2(x), \Gamma_1(x) \gg \Gamma_3 x) <: \Gamma_3(x)$, which is equivalent to $\forall x \in dom(\Gamma_3).\Gamma_4(x) <: \Gamma_3(x)$. Therefore, $\Gamma_4 \geq \Gamma_3$.

By Lemma A.7.6, $\forall x \in dom(\Gamma_3).\Gamma_1(x) \gg \Gamma_3(x) \leq \Gamma_2(x) \gg \Gamma_4(x)$.

By Lemma A.10.2, $dom(\Gamma_1) = dom(\Gamma_3)$. By repeated application of T_WIDEN_FL_EFF and the weakening lemma (A.1.5) , $\Gamma_2 \rhd t : U \lhd \Gamma_4$. □

A.10.2 Upgrade effect specific properties

Lemma A.10.7 (Upgrading, with update effects)**.** *Let* $\Gamma_1 \leq \Gamma_2$. *Let* t *be a term such that* $\Gamma_1 \rhd t : U \lhd \Gamma_3$. *It follows that there exists a* Γ_4 *such that* $\Gamma_3 \leq \Gamma_4$ *and* $\Gamma_2 \rhd t : U' \lhd \Gamma_4$.

Proof. As $\Gamma_1 \leq \Gamma_2$, it follows that there exists a Γ_5 and $\overline{x_i : T_i}$ (which is potentially empty) such that $\Gamma_2 = \Gamma_5, \overline{x_i : T_i}$ where $dom(\Gamma_5) = dom(\Gamma_1)$ and $\forall x \in dom(\Gamma_1)$. $\Gamma_5(x) <: \Gamma_1(x)$. Let $\Gamma_4 = \Gamma_3, \overline{x_i : T_i}$. By T_WIDEN_UP_EFF, $\Gamma_5 \rhd t : U \lhd \Gamma_3$. By the weakening lemma (A.1.5) , $\Gamma_2 \rhd t : U \lhd \Gamma_4$. □

Bibliography

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects*. Springer, 1996.

[2] ALDRICH, J., ET AL. The Plaid language. URL: `http://www.cs.cmu.edu/~aldrich/plaid`.

[3] ALDRICH, J., SUNSHINE, J., SAINI, D., AND SPARKS, Z. Typestate-oriented programming. In *OOPSLA 2009*, ACM. `doi:10.1145/1639950.1640073`.

[4] ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., ET AL. Adding trace matching with free variables to AspectJ. In *OOPSLA 2005*, ACM. `doi:10.1145/1094811.1094839`.

[5] AMIN, N., MOORS, A., AND ODERSKY, M. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages (FOOL) 2012*.

[6] AUGUSTSSON, L., AND CARLSSON, M. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming, Gothenburg* (1999).

[7] AVGUSTINOV, P., TIBBLE, J., AND DE MOOR, O. Making trace monitors feasible. *ACM SIGPLAN Notices 42*, 10 (Oct. 2007), 589. `doi:10.1145/1297105.1297070`.

[8] BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software* (2001), Springer-Verlag New York, Inc., pp. 103–122.

[9] BARENDREGT, H. Introduction to Generalized Type Systems. *Journal of Functional Programming 1*, 2 (1991), 125–154.

[10] BARNETT, M., FÄHNDRICH, M., LEINO, K. R. M., ET AL. Specification and verification: the Spec# experience. *Communications of the ACM 54*, 6 (June 2011), 81–91. `doi:10.1145/1953122.1953145`.

[11] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The Spec# Programming System : An Overview. *Lecture Notes in Computer Science 3362* (2005), 49–69. doi:10.1007/978-3-540-30569-9_3.

[12] BECKMAN, N., BIERHOFF, K., AND ALDRICH, J. Verifying correct usage of atomic blocks and typestate. In *OOPSLA 2008*, ACM. doi:10.1145/1449955.1449783.

[13] BECKMAN, N. E., KIM, D., AND ALDRICH, J. An empirical study of object protocols in the wild. In *ECOOP 2011*, Springer. doi:10.1007/978-3-642-22655-7_2.

[14] BECKMAN, N. E., AND NORI, A. V. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI 2011*, ACM. doi:10.1145/1993316.1993524.

[15] BETTINI, L., COPPO, M., DANTONI, L., DE LUCA, M., DEZANI-CIANCAGLINI, M., AND YOSHIDA, N. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008*. Springer. doi:10.1007/978-3-540-85361-9_33.

[16] BIERHOFF, K., BECKMAN, N., AND ALDRICH, J. Practical API protocol checking with access permissions. In *ECOOP 2009*, Springer. doi:10.1007/978-3-642-03013-0_10.

[17] BLACKWELL, A. F., AND GREEN, T. R. A cognitive dimensions questionnaire optimised for users. In *The Psychology of Programming Interest Group (PPIG) Workshop* (2000).

[18] BLOCH, J. *Effect Java, Second Edition*. Addison-Wesley, 2008.

[19] BODDEN, E. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, 2009.

[20] BODDEN, E., HENDREN, L., AND LHOTÁK, O. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 525–549.

[21] BODDEN, E., LAM, P., AND HENDREN, L. Clara: a framework for statically evaluating finite-state runtime monitors. In *Runtime Verification (RV) 2010*, Springer. doi:10.1007/978-3-642-16612-9_15.

[22] BONELLI, E., AND COMPAGNONI, A. Multipoint session types for a distributed calculus. In *Trustworthy Global Computing*. Springer, 2008. doi:10.1007/978-3-540-78663-4_17.

[23] BOVE, A., DYBJER, P., AND NORELL, U. A brief overview of Agda — a functional language with dependent types. In *TPHOLs 2009*. Springer. doi:10.1007/978-3-642-03359-9_6.

[24] BOYLAND, J. Checking interference with fractional permissions. In *SAS 2003*, Springer. doi:10.1007/3-540-44898-5_4.

[25] BRADY, E. C. Idris: systems programming meets full dependent types. In *PLPV 2011*, ACM. doi:10.1145/1929529.1929536.

[26] BRADY, E. C. Programming and reasoning with algebraic effects and dependent types. In *ICFP 2013 (to appear)*.

[27] BROOKS, R. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies 9*, 6 (1977), 737–751. doi:10.1016/S0020-7373(77)80039-4.

[28] BROOKS, R. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies 18*, 6 (1983), 543–554. doi:10.1016/S0020-7373(83)80031-5.

[29] CAPECCHI, S., COPPO, M., DEZANI-CIANCAGLINI, M., DROSSOPOULOU, S., AND GIACHINO, E. Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science 410*, 2 (2009), 142–167. doi:10.1016/j.tcs.2008.09.016.

[30] CHALIN, P., KINIRY, J., LEAVENS, G., AND POLL, E. Beyond assertions: advanced specification and verification with JML and ESC/Java2. In *FMCO 2006*, Springer. doi:10.1007/11804192_16.

[31] CHEN, F., AND ROŞU, G. Mop: an efficient and generic runtime verification framework. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 569–588.

[32] CLARKE, S. Evaluating a new programming language. In *The Psychology of Programming Interest Group (PPIG) Workshop* (2001).

[33] COK, D. R. Specifying java iterators with JML and ESC/Java2. In *SAVCBS 2006*, ACM. doi:10.1145/1181195.1181210.

[34] COLLINGBOURNE, P., AND KELLY, P. H. Inference of session types from control flow. In *FESCA 2008*, Elsevier. doi:10.1016/j.entcs.2010.06.003.

[35] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *POPL 1982*, ACM. doi:10.1145/582153.582176.

[36] DeLine, R., and Fähndrich, M. Enforcing high-level protocols in low-level software. In *PLDI 2001*, ACM. `doi:10.1145/378795.378811`.

[37] DeLine, R., and Fähndrich, M. Typestates for objects. In *ECOOP 2004*, Springer. `doi:10.1007/978-3-540-24851-4_21`.

[38] DeLine, R., and Fähndrich, M. The Fugue protocol checker: Is your software baroque. Tech. Rep. MSR-TR-2004-07, Microsoft Research, 2004.

[39] Deniélou, P.-M., and Yoshida, N. Multiparty session types meet communicating automata. In *ETAPS 2012*. Springer. `doi:10.1007/978-3-642-28869-2_10`.

[40] Dezani-Ciancaglini, M., Giachino, E., Drossopoulou, S., and Yoshida, N. Bounded session types for object oriented languages. In *FMCO 2006*, Springer. `doi:10.1007/978-3-540-74792-5_10`.

[41] Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., and Drossopoulou, S. Session types for object-oriented languages. In *ECOOP 2006*. Springer. `doi:10.1007/11785477_20`.

[42] Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., and Drossopoulou, S. A distributed object-oriented language with session types. In *Trustworthy Global Computing*. Springer, 2005. `doi:10.1007/11580850_16`.

[43] Duke, R., Salzman, E., Burmeister, J., Poon, J., and Murray, L. Teaching programming to beginners-choosing the language is just the first step. In *Proceedings of the Australasian conference on Computing education* (2000), ACM, pp. 79–86.

[44] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. Patterns in property specifications for finite-state verification. *ICSE '99* (1999), 411–420. `doi:10.1145/302405.302672`.

[45] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (2001), 99–123. `doi:10.1109/32.908957`.

[46] Fähndrich, M. Static verification for code contracts. In *SAS 2010*. Springer. `doi:10.1007/978-3-642-15769-1_2`.

[47] Fähndrich, M., Barnett, M., and Logozzo, F. Embedded contract languages. In *SAC 2010*, ACM. `doi:10.1145/1774088.1774531`.

[48] FINK, S. J., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol. 17*, 2 (2008), 1–34. `doi:10.1145/1348250.1348255`.

[49] FLOYD, R. W. Assigning meaning to programs. In *Mathematical aspects of computer science*, vol. 19. American Mathematical Society, 1967.

[50] FUH, Y.-C., AND MISHRA, P. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT 1989*, Springer. `doi:10.1007/3-540-50940-2_35`.

[51] FUH, Y.-C., AND MISHRA, P. Type inference with subtypes. In *ESOP 1988*, Springer. `doi:10.1007/3-540-19027-9_7`.

[52] FULGHAM, B., BAGLEY, D., ET AL. The computer language benchmarks game. URL: `http://benchmarksgame.alioth.debian.org/`.

[53] GALIL, Z., AND ITALIANO, G. F. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR) 23*, 3 (1991), 319–344. `doi:10.1145/116873.116878`.

[54] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[55] GANNON, J. D., AND HORNING, J. J. The impact of language design on the production of reliable software. *ACM SIGPLAN Notices 10*, 6 (1975), 10. `doi:10.1145/390016.808420`.

[56] GAY, S., AND HOLE, M. Subtyping for session types in the pi calculus. *Acta Informatica 42*, 2-3 (2005), 191–225. `doi:10.1007/s00236-005-0177-z`.

[57] GAY, S. J., AND VASCONCELOS, V. T. Linear type theory for asynchronous session types. *Journal of Functional Programming 20*, 01 (2010), 19–50. `doi:10.1017/S0956796809990268`.

[58] GAY, S. J., VASCONCELOS, V. T., RAVARA, A., GESBERT, N., AND CALDEIRA, A. Z. Modular session types for distributed object-oriented programming. In *POPL 2010*. `doi:10.1145/1706299.1706335`.

[59] GIRARD, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–101. `doi:10.1016/0304-3975(87)90045-4`.

[60] GOPINATHAN, M., AND RAJAMANI, S. K. Enforcing object protocols by combining static and runtime analysis. In *OOPSLA 2008*. `doi:10.1145/1449764.1449784`.

[61] Graham, S. L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM (JACM) 23*, 1 (1976), 172–202. `doi:10.1145/321921.321939`.

[62] Green, T., Petre, M., and Bellamy, R. Comprehensibility of visual and textual programs: A test of superlativism against thematch-mismatchconjecture. *ESP 91*, 743 (1991), 121–146.

[63] Green, T. R. G. Cognitive dimensions of notations. In *People and Computers V* (1989), pp. 443–460.

[64] Green, T. R. G., and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing 7*, 2 (1996), 131–174. `doi:10.1006/jvlc.1996.0009`.

[65] Gupta, D. What is a good first programming language? *Crossroads 10*, 4 (2004), 7–7.

[66] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987), 231–274. `doi:10.1016/0167-6423(87)90035-9`.

[67] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580. `doi:10.1145/363235.363259`.

[68] Hoare, C. A. R. Null references: The billion dollar mistake. Presented at QCon London. URL: `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`.

[69] Honda, K., and Brown, G. The Scribble language. URL: `http://www.jboss.org/scribble`.

[70] Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., and Yoshida, N. Scribbling interactions with a formal foundation. In *The 7th International Conference on Distributed Computing and Internet Technology (ICDIT)*. Springer, 2011. `doi:10.1007/978-3-642-19056-8_4`.

[71] Honda, K., Vasconcelos, V., and Kubo, M. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, Springer. `doi:10.1007/BFb0053567`.

[72] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. In *POPL 2008*, ACM. `doi:10.1145/1328438.1328472`.

[73] HU, R., KOUZAPAS, D., PERNET, O., YOSHIDA, N., AND HONDA, K. Type-safe eventful sessions in Java. In *ECOOP 2010*. Springer. `doi:10.1007/978-3-642-14107-2_16`.

[74] HU, R., YOSHIDA, N., AND HONDA, K. Session-based distributed programming in java. In *ECOOP 2008*. Springer, 2008. `doi:10.1007/978-3-540-70592-5_22`.

[75] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 3 (2001), 396–450. `doi:10.1145/503502.503505`.

[76] IHAKA, R., AND GENTLEMAN, R. R: A language for data analysis and graphics. *Journal of computational and graphical statistics 5*, 3 (1996), 299–314. `doi:10.1080/10618600.1996.10474713`.

[77] IHAKA, R., GENTLEMAN, R., ET AL. The R project for statistical computing. URL: `http://www.r-project.org/`.

[78] JASPAN, C., AND ALDRICH, J. Are object protocols burdensome?: an empirical study of developer forums. In *PLATEAU 2011*, ACM. `doi:10.1145/2089155.2089168`.

[79] JASPAN, C., AND ALDRICH, J. Checking framework interactions with relationships. In *ECOOP 2009*, Springer. `doi:10.1007/978-3-642-03013-0_3`.

[80] JIM, T. What are principal typings and what are they good for? In *POPL 1996*, ACM. `doi:10.1145/237721.237728`.

[81] KADODA, G. A cognitive dimensions view of the differences between designers and users of theorem proving assistants. In *The Psychology of Programming Interest Group (PPIG) Workshop* (2000).

[82] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *ECOOP 2001*, Springer. `doi:10.1007/3-540-45337-7_18`.

[83] KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. From uncertainty to belief: Inferring the specification within. In *OSDI 2006*, USENIX Association, pp. 161–176.

[84] LEAVENS, G. T., BAKER, A. L., AND RUBY, C. JML: A Java modeling language. In *Formal Underpinnings of Java Workshop at OOPSLA 1998*.

[85] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MÜLLER, P., KINIRY, J., CHALIN, P., AND ZIMMERMAN, D. M. JML Reference Manual. 2009.

[86] LEE, T., ET AL. The Netty Project. URL: `http://netty.io/`.

[87] MALAYERI, D., AND ALDRICH, J. CZ: multiple inheritance without diamonds. In *OOPSLA 2009*, ACM. `doi:10.1145/1640089.1640092`.

[88] MALAYERI, D., AND ALDRICH, J. Integrating nominal and structural subtyping. In *ECOOP 2008*. Springer. `doi:10.1007/978-3-540-70592-5_12`.

[89] MANN, H. B., AND WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics 18*, 1 (1947), 50–60. `doi:10.1214/aoms/1177730491`.

[90] MARKSTRUM, S. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *PLATEAU 2010*, ACM. `doi:10.1145/1937117.1937124`.

[91] MARTELLI, A., AND MONTANARI, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 2 (1982), 258–282. `doi:10.1145/357162.357169`.

[92] MCBRIDE, C., AND PATERSON, R. Applicative programming with effects. *Journal of Functional Programming 18*, 01 (2007), 1–13. `doi:10.1017/S0956796807006326`.

[93] MEIJER, E., BECKMAN, B., AND BIERMAN, G. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD 2006*, ACM. `doi:10.1145/1142473.1142552`.

[94] MEYER, B. Eiffel: A language and environment for software engineering. *Journal of Systems and Software 8*, 3 (June 1988), 199–246. `doi:10.1016/0164-1212(88)90022-2`.

[95] MICROSOFT. Code Contracts. URL: `http://research.microsoft.com/en-us/projects/contracts/`.

[96] MILITÃO, F., ALDRICH, J., AND CAIRES, L. Aliasing control with view-based typestate. In *FTFJP 2010*, ACM. `doi:10.1145/1924520.1924527`.

[97] MILNER, R. *Communicating and mobile systems: the pi calculus*. Cambridge University Press, 1999.

[98] MINSKY, Y., MADHAVAPEDDY, A., AND MINSKY, Y. *Real World OCaml*. O'Reilly Media, Inc., 2013.

[99] MITCHELL, J. C. Coercion and type inference. In *POPL 1984*, ACM. `doi:10.1145/800017.800529`.

[100] MITCHELL, J. C. Type inference with simple subtypes. *Journal of functional programming 1*, 03 (1991), 245–285. `doi:10.1017/S0956796800000113`.

[101] MOORE, E. F. Gedanken-experiments on sequential machines. *Automata studies 34*, 129–153.

[102] MORRISETT, G. L^3: A linear language with locations. *Typed Lambda Calculi and Applications* (2005), 293–307. `doi:10.1007/11417170\_22`.

[103] MOSTROUS, D., AND YOSHIDA, N. A session object calculus for structured communication based-programming. Tech. rep., University of Lisbon, 2008.

[104] NADEN, K., BOCCHINO, R., ALDRICH, J., AND BIERHOFF, K. A type system for borrowing permissions. In *POPL 2012*, ACM. `doi:10.1145/2103656.2103722`.

[105] NAEEM, N. A., AND LHOTAK, O. Typestate-like analysis of multiple interacting objects. In *OOPSLA 2008*, ACM. `doi:10.1145/1449764.1449792`.

[106] NANEVSKI, A., MORRISETT, G., SHINNAR, A., GOVEREAU, P., AND BIRKEDAL, L. Ynot: dependent types for imperative programs. In *ICFP 2008*, ACM. `doi:10.1145/1411204.1411237`.

[107] NEUBAUER, M., AND THIEMANN, P. An implementation of session types. In *Practical Aspects of Declarative Languages (PADL) 2004*. Springer. `doi:10.1007/978-3-540-24836-1_5`.

[108] NG, N., YOSHIDA, N., PERNET, O., HU, R., AND KRYFTIS, Y. Safe parallel programming with session java. In *Coordination Models and Languages (COORDINATION) 2011*, Springer. `doi:10.1007/978-3-642-21464-6_8`.

[109] NIELSON, F., AND NIELSON, H. R. Type and effect systems. In *Correct System Design*. Springer, 1999. `doi:10.1007/3-540-48092-7_6`.

[110] NIERSTRASZ, O. Regular types for active objects. In *OOPSLA 1993*, ACM. `doi:10.1145/165854.167976`.

[111] ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. An overview of the Scala programming language. Tech. rep., 2004.

[112] ODERSKY, M., CREMET, V., RÖCKL, C., AND ZENGER, M. A nominal theory of objects with dependent types. In *ECOOP 2003*, Springer. doi:10.1007/978-3-540-45070-2_10.

[113] ODERSKY, M., AND WADLER, P. Pizza into Java: Translating theory into practice. In *POPL 1997* (1997), ACM. doi:10.1145/263699.263715.

[114] O'SULLIVAN, B., GOERZEN, J., AND STEWART, D. B. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[115] PARNIN, C. A cognitive neuroscience perspective on memory for programming tasks. In *The Psychology of Programming Interest Group (PPIG) Workshop* (2010).

[116] PEARSON, K. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5 50*, 302 (1900), 157–175. doi:10.1080/14786440009463897.

[117] PETRE, M. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM 38*, 6 (June 1995), 33–44. URL: http://doi.acm.org/10.1145/203241.203251, doi:10.1145/203241.203251.

[118] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, 2002.

[119] PIERCE, B. C. *Advanced topics in types and programming languages*. The MIT Press, 2005.

[120] PIERCE, B. C., AND TURNER, D. N. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS) 22*, 1 (2000), 1–44. doi:10.1145/345099.345100.

[121] PIKE, R., GRIESEMER, R., THOMPSON, K., ET AL. The Go language. URL: http://golang.org/.

[122] PLASMEIJER, R., AND EEKELEN, M. V. Keep it clean: a unique approach to functional programming. *ACM SIGPLAN Notices 34*, 6 (1999), 23–31. doi:10.1145/606666.606670.

[123] PRABHAKAR, B., LITECKY, C. R., AND ARNETT, K. It skills in a tough job market. *Communications of the ACM 48*, 10 (2005), 91–94.

[124] Pucella, R., and Tov, J. A. Haskell session types with (almost) no class. *ACM SIGPLAN Notices 44*, 2 (2009), 25–36. doi:10.1145/1543134.1411290.

[125] Puntigam, F. Coordination requirements expressed in types for active objects. In *ECOOP 1997*, Springer. doi:10.1007/BFb0053387.

[126] Puntigam, F. Synchronization expressed in types of communication channels. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)* (1996), Springer. doi:10.1007/3-540-61626-8_99.

[127] Rajlich, V., and Wilde, N. The role of concepts in program comprehension. In *The 10th International Workshop on Program Comprehension* (2002), IEEE. doi:10.1109/WPC.2002.1021348.

[128] Ramanathan, M. K., Grama, A., and Jagannathan, S. Static specification inference using predicate mining. In *PLDI 2007*, ACM. doi:10.1145/1250734.1250749.

[129] Ravara, A. *Typing Non-Uniform Concurrent Objects*. PhD thesis, University of Lisbon, 2000.

[130] Ravara, A., and Vasconcelos, V. T. Typing non-uniform concurrent objects. In *CONCUR 2000*, Springer. doi:10.1007/3-540-44618-4_34.

[131] Reynolds, J. C. Syntactic control of interference. In *POPL 1978*, ACM. doi:10.1145/512760.512766.

[132] Reynolds, J. C. Syntactic control of interference part 2. In *Automata, Languages and Programming*. Springer, 1989. doi:10.1007/BFb0035793.

[133] Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 1 (1965), 23–41. doi:10.1145/321250.321253.

[134] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[135] Sakarovitch, J. *Elements of automata theory*. Cambridge University Press, 2009.

[136] Siek, J., and Taha, W. Gradual typing for objects. In *ECOOP 2007*. Springer. doi:10.1007/978-3-540-73589-2_2.

[137] Siek, J. G., and Taha, W. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop* (2006), vol. 6, pp. 81–92.

[138] SIMONET, V. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Programming Languages and Systems*. Springer, 2003. doi:10.1007/978-3-540-40018-9_19.

[139] SLOANE, A. M. Lightweight language processing in kiama. In *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011. doi:10.1007/978-3-642-18023-1_12.

[140] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering 12*, 1 (1986), 157–171. doi:10.1109/TSE.1986.6312929.

[141] SUNSHINE, J., NADEN, K., STORK, S., ALDRICH, J., AND TANTER, É. First-class state change in Plaid. In *OOPSLA 2011*, ACM. doi:10.1145/2048066.2048122.

[142] TAFT, S., DUFF, R., BRUKARDT, R., PLOEDEREDER, E., AND LEROY, P. *Ada 2005 Reference Manual: Language and Standard Libraries: International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*. Springer-Verlag, 2006.

[143] TAKEUCHI, K., HONDA, K., AND KUBO, M. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe (PARLE)*. Springer, 1994. doi:10.1007/3-540-58184-7_118.

[144] TARJAN, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*, 2 (1975), 215–225. doi:10.1145/321879.321884.

[145] TOV, J. A., AND PUCELLA, R. Practical affine types. In *POPL 2011*, ACM. doi:10.1145/1926385.1926436.

[146] TOV, J. A., AND PUCELLA, R. Stateful contracts for affine types. In *ESOP 2010*. Springer. doi:10.1007/978-3-642-11957-6_29.

[147] VASCONCELOS, V. T., GAY, S. J., RAVARA, A., GESBERT, N., AND CALDEIRA, A. Z. Dynamic Interfaces. *Workshop on Foundations of Object-Oriented Languages (FOOL) 2009*.

[148] WADLER, P. Linear types can change the world. In *IFIP TC Working Conference on Programming Concepts and Methods* (1990), vol. 2, pp. 347–359.

[149] WELLS, J. B. The essence of principal typings. In *Automata, Languages and Programming, 29th International Colloquium (ICALP)*. Springer, 2002. doi:10.1007/3-540-45465-9_78.

[150] WELSH, M., CULLER, D., AND BREWER, E. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP 2001*, ACM. doi:10.1145/502034.502057.

[151] WESTHEIDE, D. The neophyte's guide to scala, part 5: The option type. http://danielwestheide.com/blog/2012/12/19/the-neophytes-guide-to-scala-part-5-the-option-type.html, 2012.

[152] WHITLEY, K. N. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing 8*, 1 (1997), 109–142.

[153] WOLFF, R., GARCIA, R., TANTER, E., AND ALDRICH, J. Gradual typestate. In *ECOOP 2011*. Springer. doi:10.1007/978-3-642-22655-7_22.

[154] WRIGHT, A. K. Typing references by effect inference. In *ESOP 1992*, Springer. doi:10.1007/3-540-55253-7_28.

[155] XI, H. Imperative programming with dependent types. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science* (2000), IEEE. doi:10.1109/LICS.2000.855785.

[156] XI, H., AND PFENNING, F. Dependent types in practical programming. ACM. doi:10.1145/292540.292560.

[157] XI, H., AND PFENNING, F. Eliminating array bound checking through dependent types. In *PLDI 1998*, ACM. doi:10.1145/277650.277732.

[158] YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: Mining Temporal API Rules from Imperfect Traces. ACM. doi:10.1145/1134285.1134325.

[159] YOSHIDA, N., DENIÉLOU, P.-M., BEJLERI, A., AND HU, R. Parameterised multi-party session types. In *Foundations of Software Science and Computational Structures (FOSSACS)*. Springer, 2010. doi:10.1007/978-3-642-12032-9_10.

[160] ZOOK, D., HUANG, S. S., AND SMARAGDAKIS, Y. Generating aspectj programs with meta-aspectj. In *Generative Programming and Component Engineering* (2004), Springer, pp. 1–18.