UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

LINEAR AND SHARED OBJECTS IN CONCURRENT PROGRAMMING

Joana Correia Campos

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2010

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

LINEAR AND SHARED OBJECTS IN CONCURRENT PROGRAMMING

Joana Correia Campos

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2010

Abstract

Although mainstream object-oriented languages, like Java, are currently able to detect and prevent many programming errors by static type-checking, common usage-related errors are not captured and signalled to programmers. In general, no (formal) support is available in these languages for ensuring that an object is used according to the protocol which the programmer had in mind when describing the behaviour of a class. The file reader protocol is a simple but clarifying example: first a file must be opened, then it can be read multiple times (though not beyond the end-of-file), and finally it must be closed. As client code is not checked for protocol conformance, trying to read the file without first opening it, or when it is closed, are simple disregards caught only by runtime exceptions, assuming the language is equipped with built-in support to handle errors and exceptional events.

The MOOL programming language presented in this work is an attempt to formalise object usage and access. It consists in a simple class-based object-oriented language that includes standard primitives found in most object-oriented language formalisms. Additionally, the language offers constructs that can be attached at class definitions for specifying (1) the available methods based on an object state, and (2) how methods may be called in that state – by a single client, in which case we say that the object has a linear status, or without restrictions, in which case we say it has a shared one. We refer to this abstract view that defines an object state and status the *class usage type*. We formalise the language syntax, the operational semantics, and a type system that enforces by static typing that methods are called only when available, and by a single client if so specified in the class usage type. We illustrate the language capabilities by encoding in MOOL the protocols of two well-known examples: the file reader and the auction system. We have built a prototype compiler to implement our ideas, and its architecture is also described. Finally, we anticipate some of the related topics which we are interested in pursuing in future work.

Keywords: Object-oriented programming, concurrency, type systems, linear objects, session types

Resumo

As linguagens de programação centradas em objectos, como o Java, conseguem detectar antecipadamente em tempo de compilação, e assim prevenir, muitos dos erros de tipificação e de inicialização de atributos e variáveis que os programadores inadvertidamente introduzem nos seus códigos. Contudo, estas linguagens são actualmente incapazes de capturar e assinalar erros relacionados com a utilização incorrecta de objectos. Em geral, as linguagens comerciais não dispõem de suporte formal e verificável que garanta que uma instância de uma classe é usada de acordo com a intenção do programador que escreveu essa classe. O protocolo que descreve a leitura de um ficheiro serve de motivação: primeiro o ficheiro deve ser aberto, depois as suas linhas podem ser lidas uma por uma, e, antes de o programa terminar, o ficheiro deve ser fechado. Não havendo qualquer verificação do código cliente a este nível, é inevitável que o protocolo seja quebrado, e que uma tentativa (indevida) de leitura seja efectuada antes de invocada a operação que abre o ficheiro, ou que uma outra esbarre com um ficheiro fechado. Em linguagens como o Java, este tipo de erros só é tarde detectado, quando uma excepção é lançada em tempo de execução.

A linguagem de programação MOOL proposta nesta tese representa uma tentativa de formalizar a utilização e o acesso a objectos. Trata-se de uma linguagem baseada em classes, que inclui primitivas disponíveis na maioria dos formalismos de linguagens centradas em objectos. Adicionalmente, a linguagem dispõe de construções sintácticas que permitem ao programador especificar como é que um objecto de uma classe deve ser usado. Através desta especificação, o programador pode (1) definir os métodos disponíveis em função do estado do objecto, e (2) indicar como o objecto pode ser referenciado quando se encontra nesse estado – por uma única referência, tratando-se assim de um objecto linear, ou sem restrições no caso de um objecto que pode ser partilhado por vários clientes. Neste trabalho, a visão abstracta que os clientes têm do estado dos objectos é designada de *tipo de utilização da classe* (ou *class usage type*). Ao contrário dos tipos comuns, na linguagem proposta os tipos dos objectos são dinâmicos, variando com o seu estado. Gay *et al.* [38] designam de interface dinâmica a especificação global de métodos disponíveis numa classe com base no estado do objecto, distinguindo-a da interface estática do Java. Objectos que mudam dinamicamente o conjunto de métodos disponíveis em função do seu estado são também conhecidos na literatura como objectos

não uniformes. Nierstrasz [30] foi o primeiro a estudar o comportamento de objectos não uniformes, ou activos, em sistemas concorrentes. A linguagem MOOL oferece ainda um mecanismo de concorrência que permite a criação de *threads*, além da primitiva sync que permite a sincronização de métodos quando, em objectos partilhados, a comunicação deve ser efectuada em exclusão mútua.

A linguagem formal que é apresentada nesta tese é baseada na linguagem proposta por Gay *et al.* [15], que formaliza uma abordagem designada de *tipos de sessão modulares*. A teoria dos tipos de sessão foi proposta para a verificação em tempo de compilação de programas cuja comunicação se processa por canais tipificados. Os tipos de sessão são propostos como forma de impor que as implementações dos canais obedecem às sequências e tipos das mensagens especificados nos protocolos de comunicação. Em geral, os protocolos podem ser expressos por tipos de sessão, independentemente da linguagem de programação em que são integrados. Os tipos de sessão têm sido integrados em vários paradigmas de linguagens de programação: cálculo pi, linguagens funcionais, linguagens centradas em objectos, CORBA, etc. O trabalho sobre tipos de sessão modulares combina, numa linguagem centrada em objectos distribuída, os tipos de sessão e a ideia de disponibilidade dos métodos de uma classe em função do estado do objecto. A modularidade da abordagem resulta de a implementação do tipo de sessão poder ser decomposta pelos vários métodos da classe, contrastando com trabalhos anteriores, no contexto de linguagens centradas em objectos, em que o tipo de sessão é implementado no corpo de um único método.

O conceito de tipos de utilização desenvolvido nesta tese foi inspirado nos tipos de sessão modulares, e adaptado a um modelo de comunicação mais simples – a troca de mensagens através de chamadas de métodos. O estilo de programação proposto pretende ser simples e intuitivo, não tendo o programador de lidar com distribuição (a concorrência é efectuada por memória partilhada), mas tirando proveito das propriedades de segurança associadas aos tipos de sessão.

As contribuições desta tese materializam-se na formalização da sintaxe, da semântica operacional e do sistema de tipos, que verifica as especificações em tempo de compilação, e na implementação de um compilador de MOOL, cuja arquitectura também se descreve.

No contexto dos tipos de sessão, as contribuições desta tese podem resumir-se nos seguintes pontos:

- Ao contrário de abordagens anteriores, o modelo de comunicação baseia-se exclusivamente na chamada de métodos;

- As classes são anotadas com uma especificação de utilização que estrutura a sequência de métodos que os clientes podem invocar em função do estado do objecto, a qual é enriquecida com qualificadores lin/un que permitem controlar se um objecto é linear, e uma única referência o pode usar, ou se é partilhado, não existindo restrições quanto ao número de clientes;

- Ao contrário de abordagens anteriores, os canais partilhados são substituídos pela primitiva de sincronização convencional que permite que certas operações num objecto sejam acedidas sem interferência de outras *threads*.

A tese encontra-se organizada em seis capítulos. Na introdução, o exemplo do leitor de ficheiros é apresentado como motivação dos principais aspectos da linguagem e, em particular, da especificação de tipos de utilização. Segue-se um capítulo onde se apresenta um exemplo mais extenso – o sistema de leilões – que ilustra a introdução de tipos de utilização em protocolos mais complexos. No capítulo da linguagem formal, é descrita a sintaxe, a semântica operacional e o sistema de tipos. Alguns exemplos de derivação de tipos ilustram o funcionamento das regras e as mudanças operadas no tipo dos atributos à medida que a derivação avança. A prova formal da linguagem MOOL não é apresentada nesta tese, ficando adiada para um trabalho futuro. No entanto, sendo este sistema baseado num sistema de tipos [15] para o qual é apresentada a prova formal por indução, tudo aponta para que não seja difícil provar os resultados também para a linguagem MOOL. No capítulo da implementação, são descritas as tecnologias utilizadas e a arquitectura do compilador. Desenvolvido na linguagem Java, o compilador assenta na *framework* do SableCC, que gera automaticamente analisadores léxicos e classes que implementam o padrão visitante, o que permitiu reduzir o esforço de implementação ao desenvolvimento da componente semântica, através da extensão destas classes. A arquitectura do compilador distingue duas fases: a fase de análise, onde, com base na árvore abstracta do programa, é construída a tabela de símbolos e efectuada a verificação de tipos com base num algoritmo de tipificação guiado pelos tipos de utilização, e a fase de síntese, onde o código de um programa MOOL válido é traduzido para uma linguagem intermédia. O resultado desta traduçao é convertido em *bytecode* pela plataforma *open source* Mono, tendo como alvo a Common Language Runtime (CLR). No capítulo do trabalho relacionado, discute-se o estado da arte em tipos de sessão, focando o contexto das linguagens centradas em objectos, em *typestates*, uma abordagem com a qual as ideias desta tese apresentam pontos de contacto, e em objectos lineares. Finalmente, na conclusão antecipa-se o trabalho futuro, nomeadamente a apresentação dos resultados da linguagem MOOL e o estudo de técnicas que introduzam alguma flexibilidade na utilização de objectos lineares.

Palavras-chave: Programação centrada em objectos, concorrência, sistemas de tipos, objectos lineares, tipos de sessão

Contents

List of Figures

Listings

Chapter 1

Introduction

As software complexity and size increase, more effective techniques for building reliable systems are needed. In particular, large applications require both modular development strategies and formal behavioural specifications that assist programmers in validating the correctness of their code.

Concurrent programming usually introduces further complexity into the already complex world of traditional sequential programming. Reasoning about concurrent programs can be difficult, since thread interference should be considered at several program points. Race conditions and deadlocks are some common concurrency-related problems in mainstream programming languages.

Mainstream object-oriented languages, like Java, can automatically detect and prevent (standard) type and initialisation errors through compile-time checks. However, they cannot detect and prevent errors related to usage protocols (usually only described in informal documentation), which end up revealing themselves as runtime exceptions in languages equipped with built-in support to handle errors and exceptional events. For example, reading from a file should follow the rules of a well-known usage protocol: first a file must be opened, then it can be read multiple times (without reading beyond the end-of-file), and finally it must be closed. Nevertheless, in mainstream object-oriented languages, any disregard of this specification is only detected when it eventually causes a runtime exception. As a result, one of the major topics of computer science today is the study of strategies that can verify sequential and concurrent programs against precise interface specifications.

The design of languages conceived especially for reasoning about program correctness via specification and verification has received great attention since the Floyd–Hoare logic [13, 21] in the late 1960's. Since then, several programming languages were developed with mechanisms to prove program correctness and, in particular, with support for statically enforcing specifications which programmers can write to express their intentions about how a program should work. This support for recording and enforcing assumptions about programs creates a new programming methodology in which developers are also

called to reason about their code. However, the complexity of some specification models is still a deterrent to the diffusion of these techniques to mainstream languages. Often, readability and ease of understanding are design considerations that are not regarded as important as correctness and safety ones, and software development practices continue to be subjected to numerous errors that could easily be avoided by the use of these safer languages.

Program development in the industry could greatly benefit from having specification and verification techniques fully integrated in mainstream programming languages. Benefits would not just be long-term, with the reduction in costs of software maintenance and the extension of software lifespan. Immediate benefits would also come of having programmers use a methodology that allows them to automatically detect any usage errors as early as compile-time.

In this context, we propose MOOL, a mini object-oriented language in a Java-like style. Our programming language offers constructs that programmers can attach at class definitions to capture usage protocols, describing how an object should be used and accessed. In particular, programmers can use these constructs to specify (1) the available methods and (2) the existence of aliasing restrictions, both of which depend on an object state. When only one client can reference an object, we say that the object has a linear status. When multiple clients can reference the same object (i.e no aliasing restrictions are imposed), we say that the object has an unrestricted, or shared, status. This combination of properties (state and status) provides an abstract view of an object as seen by clients, which we call the *class usage type*. As opposed to standard types, usage types are dynamic, varying over time with the object state. Based on an object usage type, the MOOL type system can statically enforce that (1) methods are called in the specified order and that (2) aliasing is compliant with the object specified status. MOOL also includes a simple concurrency mechanism for thread spawning, and a standard mutual exclusion facility to be used when access to certain operations on shared objects needs to be performed without thread interference.

Our formal language is based on a recent work by Gay *et al.* [15], which provides a new approach to session types within the distributed object-oriented paradigm, referred to as *modular session types*. Session types have been proposed to enhance the verification of programs at compile-time by addressing the issue of typed communication over channels. Typically, all communication takes place within the context of sessions. In practice, sessions are just blocks of code in which a channel is made available for the communication between two or more participants. Session types provide a means to enforce that channel implementations obey the sequences and types of messages specified in communication protocols, and are thus of great assistance to programmers who want to verify the correctness of their programs. This theory has proved effective when applied to static type-checking concurrent and distributed object-oriented languages, revealing both

the expressiveness and complexity needed for modelling several common computing scenarios, such as client-server and peer-to-peer protocols.

In the recent approach described in the work on modular session types [15], the modularity comes from partitioning the session implementation into several methods, as opposed to previous approaches in the context of object-oriented languages in which the session is implemented in a single method body. Moreover, there is a successful attempt at treating session-typed communication channels as objects by hiding channel primitive operations in an API from where clients can call methods. In this thesis, we take the overall concept on which session types are founded, namely that communication should proceed according to specified protocols, which are statically enforced, and use it to address the issue of typed object usage in a programming language that relies on a simpler communication model – message passing in the form of method calls. We follow the programming style of Java, and propose a language where the programmer does not have to deal with distribution (we use shared memory concurrency) nor channels, while still taking advantage of the type-safety properties that are usually associated with session types. The design of our language was guided by the attempt to make it not only type-safe at compile-time, and for that we introduce statically enforced specification constructs, but also simple for object-oriented practitioners, which we accomplish by introducing an intuitive specification in a simple class-based object-oriented programming language.

1.1 Motivation

There are several reasons why a type system should enforce that an object, and its methods, should only be available under specific conditions. For example, an attempt to pop an element from an empty stack, or to advance an iterator beyond the end of a list, or a file that is freely manipulated by several clients, are simple uses of objects that can lead to unexpected behaviour, and even loss of data.

Other approaches that check that an object is correctly used based on method availability, namely systems of typestates for object-oriented languages [7, 10], rely on pre- and post-condition annotations on method definitions, describing what an object state should be before and after each one of its methods is called. In our language, we do not use method annotations; instead, we attach a *usage* specification at the class level to define how an instance of the class should be used.

Gay *et al.* [38] call the global specification of available methods the object *dynamic interface* to distinguish from the interface offered by Java. In the literature, objects that dynamically change the set of available methods are also known as *non-uniform objects*. Nierstrasz [30] was the first to study the behaviour of non-uniform, or active, objects in concurrent systems.

We follow a similar approach in which the available methods depend on an object

state. Additionally, we make aliasing depend on its status. To introduce the concepts that support our ideas, we present a simple example. Consider a FileReader that uses an object of class File, whose behaviour has already been informally described, and supports multi-threaded access once the contents of the file has been obtained. A FileReader hides the File protocol from clients, revealing its own: it starts by reading an entire file into a string; once the reading is concluded, the object can be shared, and several threads can have references to the object in order to obtain the file contents. A counter tracks the number of accesses made to the object. During the reading process, there is a choice of obtaining the file contents read up to that point.

Listing 1.1 formalises the File protocol, which the FileReader class (the client side) in Listing 1.2 implements. Each class usage type formally describes the informal descriptions provided above. The FileReader uses an object of class File, conforming to its protocol. Similarly, a FileReader client should use an object of this class taking into consideration its formalised protocol. The usage specifications attached to the class definitions indicate the order in which methods are to be called and the aliasing restrictions, if any, imposed on clients. An object of class File is linear from beginning to end, while an object of class FileReader begins by imposing restrictions on aliasing, but then evolves into state Done where the specified methods may be called by multiple threads. Although a detailed explanation of the usage specification is provided later in Section 2.2.1 and Chapter 3 of this thesis, below we comment briefly on some specific features.

Notice first that the **lin** qualifier (used to indicate linearity) tracks the object status which is controlled by its current type. For example, after opening the file in line 10 of class FileReader, the field f changes its type to File[Read] (see the File usage), where the next available method is eof, annotated with the **lin** descriptor. Read is a symbolic name representing the object state or current type. The FileReader type accompanies the changes in the File type. After calling open on a FileReader object, its type changes to FileReader[Next].

The entire File protocol from state Read to state **end**, where there are no more available methods, takes place in the method next body of class FileReader (lines 17-22). The form $\langle \ldots + \ldots \rangle$, which we call a variant type, in the File usage specification, indicates that method eof returns a result of type **boolean** on which depends the object subsequent state. So, in the FileReader implementation, a test is performed in order to find out whether the end-of-file has been reached, and the file has to be closed by invoking method close, or the next string can be read by invoking method read. Syntactically the (;) binds stronger than the (+), which separates the two possible results (the true result is described on the left hand-side, while the false one is described on the right hand-side). Because a variant is implicitly linear, the **lin** is omitted in the type that starts each variant.

The FileReader example also includes a variant type. The true variant, represented by

```
1 class File {
2   usage lin open; Read
3     where Read = lin eof; ⟨close; end + read; Read⟩;
4   ...
5 }
```

Listing 1.1: A file usage type

```
1 class FileReader {
2   usage lin open; Next
3     where Next = lin{next; ⟨Next + Done⟩ +
4                      toString; Next};
5           Done = *{toString + getCounter};
6
7   File f; string s; int counter;
8
9   unit open(string name) {
10    File f = new File(name);
11    f.open("r");
12    s = "";
13    counter = 0;
14  }
15
16  boolean next() {
17    if(f.eof()) {
18      f.close();
19      false; // return false
20    } else {
21      s += f.read();
22      true; // return true
23    }
24  }
25
26  string toString() {
27    count();
28    s; // return s
29  }
30
31  int getCounter() {
32    counter; // return counter
33  }
34
35  sync unit count() {
36    counter++;
37  }
38 }
```

Listing 1.2: A file reader

state Next, defines a recursive type, showing a similar behaviour to the one of the recursive type Read in the false variant of the File class usage type. While this recursive type eventually ends (method next returns **false** once the File protocols ends), the *anonymous* recursive type denoted by (*) in line 5 of class FileReader reveals a different behaviour of a type that never ends. The state Done marks the change in the object status, which evolves from linear to shared. It is not imposed by the MOOL type system, but rather it emerges as a property that the only form of unrestricted useful types, apart from **end** (that is **un**{}), are of this form. In a branch type, the (+) assumes a different meaning, indicating a choice: when in state Done any of the methods toString and getCounter can be called an infinite number of times, in no particular order. Formally, this type defines a recursive branch type of the form μX.**un**{toString.X + getCounter.X}, where the **un** stands for unrestricted (or shared), and, to lighten the syntax, can be omitted as we consider it to be an object default status. Because of the particular form of this recursive type, calling any of these methods on an instance of class FileReader does not change the object state nor the set of available methods.

Finally, notice that the method count, defined in line 35, is not referred in the FileReader usage type. This means that we can regard it as a *private* method, because in MOOL clients can only *view* (and call) methods defined in the class usage specification. This method is **sync**-annotated, but when called in state Next, the method annotation has no effect, as the **lin** qualifier in the class usage type, by forbidding aliasing, already enforces mutual exclusion.

There are several remarks to be made regarding this motivating example:

- The class usage type in the File and FileReader examples prevents common pitfalls in the usage of a file object by enforcing a sequence of legal method calls. For example, clients can no longer read from a file without first opening it, or read beyond the end-of-file. Moreover, the class usage type documents and formalises the specification which our compiler ensures at compile-time that clients correctly implement.

- This simple example also illustrates a problem of *shared* accesses which may arise in both sequential and concurrent object-oriented programming. In mainstream languages, there is no way to avoid the potential interference of two clients with a reference bound to the same file object. For example, one reference could be used to make the file unreadable (by invoking method close on it) between the test on eof and the reading operation made by another reference. This might result in inconsistencies in the clients' views of the file, and it illustrates a situation that can happen both in single- and multi-threaded settings. Our solution to this problem is to introduce a status (linear or unrestricted) attached to an object type. In the example, the field f declared in class FileReader is the only reference allowed to the File object, thus ensuring the safeness of the protocol. The FileReader object is also

linear while the File type has not been consumed to the end. Once the file has been closed, we can allow that several threads hold references to the same FileReader instance.

Foregoing work on session types for objects deals with linear types only. The introduction of shared objects is dealt with in the implementation mentioned in modular session types [15], while defining two distinct categories to treat linear and unrestricted objects. In our approach to object aliasing control, we define a single category for objects, as opposed to distinct categories for linear and for shared objects. We let the current status of an object to be governed by its type, allowing linear objects to evolve into shared ones (*cf.* [35, 37]). The opposite is not possible, as we do not keep track of the number of references to a given object.

- Finally, not all programming problems can be solved by the introduction of linearity. However, forbidding aliasing would be a too restrictive approach. Because we do not track the number of references existing to a shared object, we never let an object with an unrestricted type to go back to being linear. As a result, some additional mechanism has to used to allow for concurrent access in mutual exclusion. Since our main focus are linear objects, to enforce serialised access to certain methods that manipulate shared data, we adopt a standard and straightforward solution similar to the *synchronized* mechanism used in Java to prevent thread interference. The need for this mechanism should become clear from regarding method count in the example, where the **sync** keyword modifies the method in order to ensure that no updates to the counter are lost when the method is accessed by multiple threads.

1.2 Contributions

Building on previous work by Gay *et al.* [15, 38], we propose a core concurrent object oriented-language that includes specification constructs based on the idea of an object usage protocol. We formalise the operational semantics and type system. Finally, we implement these ideas in a prototype compiler.

Regarding session types, the contributions of our language are as follows:

- In contrast to other works on session types, we elect method invocation as the only communication model, both in concurrent and sequential programming;

- We annotate classes with a usage descriptor to structure how clients can call methods, and we enhance it with lin/un qualifiers for aliasing control, thus defining a single category for objects that may evolve from a linear status into an unrestricted one;

- In contrast to other works on session types, we replace the well-known shared channel primitive by a conventional synchronization primitive for mutual exclusion ac-

cess in concurrent settings. We introduce a sync method modifier to describe those operations in shared objects that must be accessed without thread interference.

1.3 Thesis Outline

The current chapter introduces our work and its main motivations. The remaining chapters are organised as follows:

Chapter 2 describes the programming language via an extended example, which illustrates the main features of the specification language;

Chapter 3 formally presents the core language, describing its syntax, operational semantics and type system. Its safeness is not proved in this thesis; we leave it for future work. Because our type system relies on many of the premises upon which the work on modular session types [15] is built, which provides a full formal treatment for its language, we are confident that MOOL inherits similar type-safety properties, and that we will be able to formally demonstrate them in future work;

Chapter 4 describes the prototype compiler implementation, introducing the main technologies upon which the compiler was built, as well as a view of its architecture. As far as technologies are concerned, the compiler is written in Java, and the SableCC framework provides a parser generator and tree-walkers, which are extended in order to implement the analysis on the input MOOL programs. The compiler targets Mono, the open-source *clone* of the Common Language Runtime (CLR);

Chapter 5 discusses the state of the art on session types, in particular in the context of object-oriented languages, typestates, and unique ownership of objects;

Chapter 6 presents our conclusions and points out lines for future work.

Chapter 2

Example: The Auction System

In this chapter, we introduce an example of a full system written in the MOOL programming language in order to illustrate its syntax and main capabilities. Our main focus are the usage specification details, apart from which the language is a standard class-based object-oriented language. All the key technical ideas should already become clear in this chapter (see Chapter 3 for a more technical introduction).

The example models an auction system. We begin by explaining the system overall requirements and describing the usage protocols using an informal representation (Section 2.1). Then, we provide our solution encoded in MOOL (Section 2.2) in which we formalise the described protocols in usage types, and discuss the decisions taken. Besides demonstrating a practical use of MOOL, we reveal in this extended example the potentialities of our approach, which can be summarised in the following features:

- Expressiveness: the usage specification can be used for modelling relatively complex scenarios;
- Small learning curve: it is very easy to understand the usage specification and start writing types using it;
- Safer code: the programmer specifies in the class usage type how objects should be used and accessed (namely, the available methods, and the aliasing restrictions), and our type-checker verifies that client implementations comply with the usage type.

2.1 The Auction Usage Protocol

Our auction system, adapted from [34, 35], features three kinds of participants: the auctioneer, the sellers and the bidders. Sellers sell items for a minimum price. Bidders place bids in order to buy some item for the best possible price. The auctioneer controls these interactions.

The system is best described by the UML sequence diagrams in Figures 2.1 and 2.2, modelling the two main scenarios through a sequence of messages exchanged between objects. The first diagram describes the scenario for a seller, while the second one describes the scenario for a bidder. We are mainly interested in visualising the interaction between the different players; we do not represent concurrent communication, even though it should not be hard to imagine several seller and bidder threads concurrently making requests to the same auctioneer object.

The sequence diagrams show, as parallel vertical dotted lines (called *lifelines*), the different objects that participate in the protocol, and as horizontal arrows, the messages exchanged between them, in the order in which they occur. Activation boxes, or method-call boxes, are the rectangles drawn on top of lifelines to represent the execution of an operation in response to a message. To lighten the representation, we omit the usual dashed open-arrowed line at the end of an activation box indicating the return from a message and its result. Instead, we simply annotate sent messages with the returned value, if any. A conditional message is shown by preceding the message by a conditional clause in square brackets. The message is only sent if the clause evaluates to true.

The first diagram (Figure 2.1) depicts how a Seller initiates the interaction with the Auctioneer indicating the item to auction and its price. The Auctioneer, after creating an Auction object, where the bids for the item being sold are going to be placed, delegates the service to a new Selling object. Once the auction is set, the auctioneer can start receiving bids for that item. We signal with a note the moment when the two scenarios interweave.

In the interaction initiated by a Bidder (Figure 2.2), the Auctioneer receives a request for the item searched and, if the item is being auctioned, it delegates the service to the Bidding object. Still in the bidder scenario, we can see that a Bidder holds its own Bidding object from which it can obtain the item initial price as defined by the Seller. Based on the returned value, the Bidder decides whether to make a bid by calling the appropriate method on the Bidding object, or else do nothing. Returning to the seller scenario, if the sale is successful, the Selling object allows a Seller to obtain the sale final price by invoking method getFinalPrice.

Both scenarios depict a similar pattern, and it is not difficult to conclude from the diagrams that the usage protocols should be specified in the Selling and Bidding classes, and that these protocols should describe linear types. This is the only way we can guarantee that clients (sellers and bidders) fully obey the specification, by enforcing that their types are consumed to the end. Notice in the diagrams that a fresh Selling (and Bidding) object is created at the beginning of each selling (and bidding) interaction, and is implicitly destroyed at the end. We signal object destruction in the diagram to increase the expressiveness of the representation; the type system provides crucial information to object deallocation.

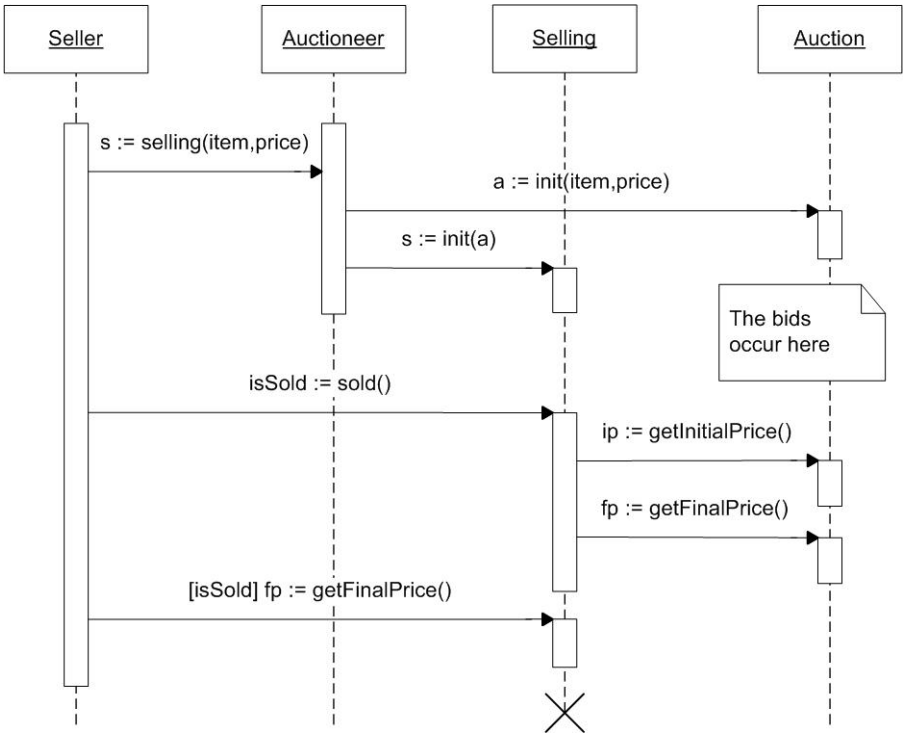


Figure 2.1: The scenario for a seller

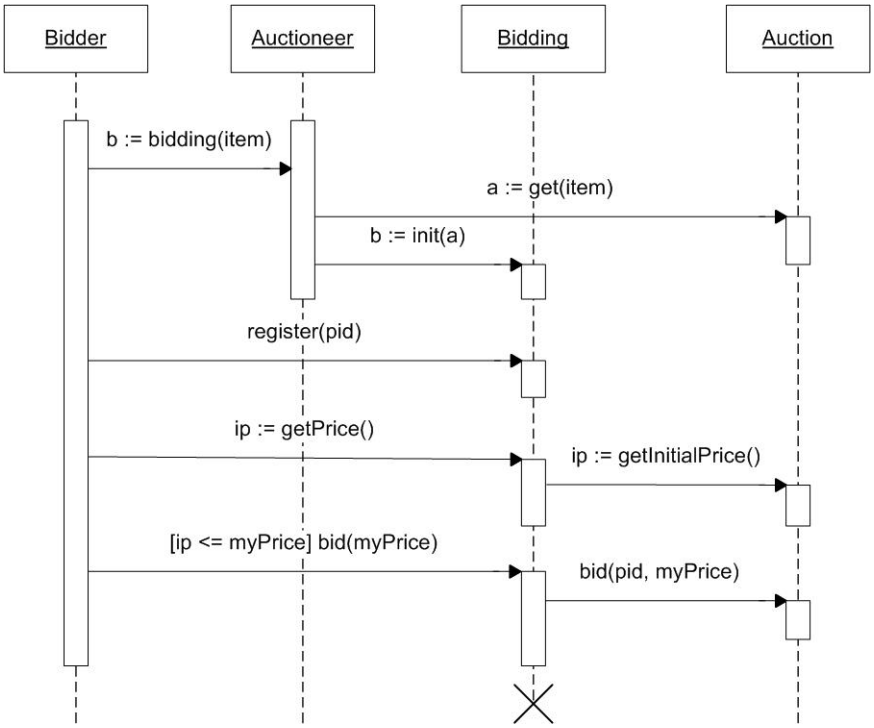


Figure 2.2: The scenario for a bidder

2.2 Programming with Usage Types

Given the above informal description of the system usage protocols, it is straightforward to write the code for each object identified in the two UML sequence diagrams. A possible implementation is sketched in Listings 2.1-2.7. Each class begins by specifying in a class usage descriptor the sequence of available methods and the aliasing restrictions, if any, with which clients should comply. No additional specification is needed since, as conventionally established, method signatures convey all the information clients need (the number and type of parameters, and the return types).

2.2.1 Usage syntactic details

Before analysing each class and its usage in detail, we introduce some less obvious syntactic details, in order to complete the brief description provided in Section 1.1. The usage formalises how clients should use an object of a given class, and no other usage is possible. Although our language does not support access level modifiers, all methods that are not referred in the usage specification are not visible to clients. For the same reason, class fields are also private, and cannot be used directly.

If a program defines a conventional class named C with methods m1, m2 and m3, and no usage declaration, our compiler will insert **usage** $*\{$m1 + m2 + m3$\}$ as the class default usage type, where each method (m1, m2 and m3) is always available. Formally, this usage defines a recursive branch type of the form μ X.**un**$\{$m1.X + m2.X + m3.X$\}$. The **un** qualifiers are omitted in the examples. A choice between calling one of the three available methods is indicated by (+). Because of the particular form of the recursive type, calling any of these methods on an instance of class C will not change the object state nor the set of available methods.

A typical usage declaration for a linear object is a sequential composition of available methods. If class C is linear, **usage lin** m1; **lin** m2; **lin** m3; **end**; is a possible usage declaration. Calling methods in the prescribed order on an instance of this class changes the object state and the set of available methods. State **end** is an abbreviation for **un** $\{\}$. When an object is in this state, it means that the sequence of available methods is empty (the usage protocol is finished).

A variant type, denoted by $\langle \ldots + \ldots \rangle$, is indexed by the two values of the **boolean** type returned by the method to which the variant is bound. A client should test the result of the call: if **true** is returned, the new object state, and the available methods, are to be found in the left-hand side of the variant; if **false** is returned, it is the right-hand side to dictate the object state and available methods.

```
1 class Auctioneer {
2   usage lin init; *{selling + bidding};
3
4   AuctionMap map;
5
6   unit init() {
7     map = new AuctionMap();
8   }
9
10  Selling[Sold] selling(string item, int price) {
11    Auction a = new Auction();
12    a.init(item, price);
13    map.put(item, a);
14    Selling s = new Selling();
15    s.init(a);
16    s; // return s
17  }
18
19  Bidding[Register] bidding(string item) {
20    Bidding b = new Bidding();
21    b.init(map.get(item));
22    b; // return b
23  }
24 }
```

Listing 2.1: An auctioneer

```
1 class Auction {
2   usage lin init;
3             *{bid + getInitialPrice +
4               getMaxBid + getBidder};
5
6   string item; int initPrice;
7   int bidder; int maxBid;
8
9   unit init(string item, int initPrice) {
10    ... // initialize fields
11  }
12
13  sync unit bid(int pid, int bid) {
14    if(maxBid <= bid) {
15      bidder = pid;
16      maxBid = bid;
17    }
18  }
19  ... // the getters
20 }
```

Listing 2.2: An auction

2.2.2 The Auctioneer

Consider now the usage specification in Listing 2.1, which provides an implementation of class Auctioneer. When an object of this class is created using the explicit constructor init , only one client can reference it, but then the object evolves into an unrestricted type, allowing several sellers and bidders to hold references to the same Auctioneer instance. Notice that we have defined a recursive (shared) type.

Each client can do one of two things:

- it can call method selling to obtain an object that provides an implementation of the selling activity on the Auctioneer; or
- it can call method bidding and obtain an object that implements the buying activity on the Auctioneer.

Then, it can repeat the interaction all over again: a seller can lower the price of an item with no bids, and start a new sale; a bidder can bid a higher price. The type never ends, and this illustrates why, in any program, we cannot keep track of the number of references to a shared object. The return types of these two methods are Selling [Sold] and Bidding[Register] respectively, because they need to be consistent with the state of the Selling and Bidding objects after the method bodies are executed and advance the types. In the example, the state Sold is short for **lin** sold; ⟨getPrice; **end** + done; **end**⟩; (lines 3 and 4 in Listing 2.4), and is used for convenience to replace the full type above by a name that represents the abstract state.

Notice, still in Listing 2.1, that when a new selling request is made, a new Auction object is created (line 11). The code of class Auction is shown in Listing 2.2. This instance is then added to the AuctionMap object (whose class we omit), where the Auctioneer keeps all the auctions (line 13), and is passed to the constructors of both the Selling and Bidding objects. In method selling , the reference to the Auction is created locally within the method body and assigned to a variable (line 11), while in method bidding, it must be fetched from the AuctionMap object (line 21). It is through reading and writing to this shared Auction object that the protocol takes place.

The **sync** modifier of method bid (line 13 in Listing 2.2) is used to control concurrent bids made by separate Bidder threads via their Bidding objects (see Listing 2.6). This annotation also qualifies the put and get operations on class AuctionMap.

2.2.3 The Seller

Listings 2.3 and 2.4 implement two linear types. The usage declaration in class Seller is an abbreviation for the nested composition of branch types **lin**{ init ; **lin**{run; **un**{}}}. Also

```
1 class Seller {
2   usage lin init; lin run; end;
3
4   string item; int price;
5   Auctioneer a;
6
7   unit init(Auctioneer a, string item, int price) {
8     ... // initialize fields
9   }
10
11  unit run() {
12    Selling s = a.selling(item, price);
13    ... // wait for the auction to end
14    if(s.sold())
15      print("made " +
16        s.getPrice() + " euros!");
17    else {
18      s.done();
19        if(lowerPrice())
20          run();
21    }
22  }
23
24  boolean lowerPrice() {
25    ... // implementation omitted
26  }
27 }
```

Listing 2.3: A seller

```
1 class Selling {
2   usage lin init; Sold
3     where Sold = lin sold;
4           ⟨getPrice; end + done; end⟩;
5
6   Auction a; int finalPrice;
7
8   unit init(Auction a) {
9     ... // initialize fields
10  }
11
12  boolean sold() {
13    finalPrice = a.getFinalPrice();
14    finalPrice >= a.getInitialPrice();
15  }
16
17  int getPrice() {
18    finalPrice;
19  }
20
21  unit done() {
22    // dummy method
23  }
24 }
```

Listing 2.4: A selling protocol

```
1 class Bidder {
2   usage lin init; lin run; end;
3
4   int pid; int myPrice; Auctioneer a;
5
6   unit init(Auctioneer a, int pid, int myPrice) {
7     this.a = a;
8     this.pid = pid;
9     this.myPrice = myPrice;
10  }
11
12  unit run() {
13    Bidding b = a.bidding();
14    b.register(pid);
15    if(b.getInitialPrice() <= myPrice)
16      b.bid(myPrice);
17    else
18      b.done();
19  }
20 }
```

Listing 2.5: A bidder

```
1 class Bidding {
2   usage lin init; Register
3     where Register = lin register;
4       lin getPrice; lin{bid; end + done; end};
5
6   Auction a; int myBidder; int price;
7
8   unit init(Auction a) {
9     this.a = a;
10    price = 0;
11  }
12
13  unit register(int myBidder) {
14    this.myBidder = myBidder;
15  }
16
17  int getPrice() {
18    if(price == 0)
19      price = a.getInitialPrice();
20    price;
21  }
22
23  unit bid(int price) {
24    a.bid(myBidder, price);
25  }
26
27  unit done() {
28    // dummy method
29  }
30 }
```

Listing 2.6: A bidding protocol

notice an example of a variant type in class Selling (line 4 of Listing 2.4). The initial type of each variant must be linear, so the redundant **lin** qualifier can be omitted. This variant type requires that the client tests the **boolean** result of method sold in order to determine the next available method: if the value evaluates to **true**, then the caller can obtain the price (because the item was sold) via method getPrice, otherwise the protocol ends.

The Seller implementation completely complies with the Selling protocol (lines 14-16) on what concerns reference s declared in line 12: the result of method sold is tested on the condition of the **if** expression, and the price of the sale is printed if it evaluates to **true**, otherwise the protocols ends, and the Selling object is deallocated through the *dummy* method done. This method only purpose is marking the end of the protocol (and of the object). However, if method lowerPrice returns **true**, the protocol can restart, a new Selling object is created, and memory is allocated to it.

The method lowerPrice in class Seller (line 24 in Listing 2.3) is an example of a method not referred in the usage specification (and consequently omitted from the object type). Our type system ensures that only methods in the usage type are visible to clients (*cf.* [15, 38]). All methods not specified in the usage type can be accessed in their own classes, but never alter the type (otherwise client views of the object could become inconsistent).

2.2.4 The Bidder

The Bidder and Bidding classes in Listings 2.5-2.6 are similar in terms of structure to the classes described above for the seller protocol. Again, the usage type specified in class Bidder shows an explicit constructor init and a run method (not recursive in this implementation). In the method run body, the Bidding object is used according to its usage type, starting at state Register.

In order to bid, a bidder must register its process number; then, it must get the item price; and finally, it can either make a bid or do nothing, and the protocol ends. Notice the slight variation of the usage configuration in line 4 of Listing 2.6 in which the type offers a choice of two different calls (bid or done) based on the result of a method with a non-boolean return type. In line 15 of Listing 2.5, a Bidder object makes its choice based on the price it is willing to pay.

2.2.5 Putting all together

The Main class in Listing 2.7 is the glue: it creates an Auctioneer object that controls the auction, and spawns a separate thread for a Seller and two Bidder objects, using a Java-like technique for thread creation.

```
1 class Main {
2   usage lin main; end;
3
4   unit main() {
5     Auctioneer a = new Auctioneer();
6     a.init();
7     Seller seller = new Seller();
8     seller.init(a, "psp", 100);
9     spawn seller.run();
10    Bidder bidder1 = new Bidder();
11    bidder1.init(a, 1, 100);
12    spawn bidder1.run();
13    Bidder bidder2 = new Bidder();
14    bidder2.init(a, 2, 100);
15    spawn bidder2.run();
16  }
17 }
```

Listing 2.7: The main class

Chapter 3

The Core Language

In this chapter, we present the technical details of the MOOL core language, an essential subset of the language used in the example (see Chapter 2) and implemented in a prototype compiler (see Chapter 4). The subset consists of those features that define MOOL, distinguishing it from similar languages, as well as a handful of other features that support it as a practical language. We start with the formal syntax in Section 3.1, then we define the operational semantics in Section 3.2 and subtyping in Section 3.3, and finally we formalise the type system in Section 3.4. The main simplification in our language, as opposed to previous approaches that work with session type concepts within the object-oriented paradigm, is that it does not include channels; instead, only message passing in the form of method calls is used. In particular, the set of expressions defined in our language syntax are constructed based exclusively on standard primitives found in most object-oriented language formalisms. The only unusual element is the *usage type*, specified in each class, which provides an abstract view of the protocol that clients must follow.

Most of the technicalities that we present are based on the core language from modular session types by Gay *et al.* [15], and on the linear type system presented by Vasconcelos [37]. Inspired by these approaches on sessions types, our main challenge was the attempt to formalise the convergence of channels and objects in a simple concurrent object-oriented language.

3.1 Syntax

In MOOL, we distinguish between the user syntax and the runtime syntax. The former is the programmer's language, and is defined in Figure 3.1; the latter is only required by the type system and operational semantics, and appears in Figure 3.2.

The metavariables C, f, and m range over class, field, and method names, respectively. We write $\vec{F}$ as short for $F_1; \ldots; F_n;$ (a sequence of field declarations), $\vec{M}$ as short for $M_1 \ldots M_n$ (a sequence of method declarations), where $n \geq 0$ in all cases. We abbreviate all sort of sequences using a similar pattern.

We assume that class identifiers in a sequence of declarations $\vec{D}$ are all distinct, and that the set of method and field names declared in each class contains only distinct names as well. Object references o include the keyword this, which references the current instance. In the examples, field accesses omit the prefix this; the compiler can insert it when needed.

In order to simplify the static and dynamic semantics, as well as the type system, in which we focus our analysis later in this chapter, we introduce the following restrictions on the syntax of specific constructs:

(i) only assignment to fields is defined;

(ii) a method call is only defined on a field of an object reference (not on an arbitrary expression);

(iii) it follows from the above that calling a method on a parameter requires assigning it first to a field;

(iv) a method accepts only one parameter.

The rules in the type system and operational semantics can easily be extended to methods accepting multiple parameters by recursively considering each of the parameters, instead of just one. The remaining restrictions are mainly related to the limited use of parameters in the core language. Passing a parameter to a method call, and assigning it to a field are the only two ways to use this kind of reference in the core language. While being a restriction, it supports good programming practices, since calling a method on a parameter changes the object type, and hence its assignment to a field, instead of its direct use in a call, should be preferred.

In the implementation, the syntax has been extended to include local variables. The rules for fields are easily adapted to variables, the main difference being that a variable has a method-level scope, whereas a field is defined at the class level. The resulting implications on linearity are taken into consideration. Self-calls (made on this) are defined in the syntax and the operational semantics, and have also been implemented. However, since self-calls do not modify the object type, we omit from the type system the typing rule for this construct.

3.1.1 User syntax

Classes, Fields and Methods The syntax of a class declaration D defines a class named C that encloses a usage descriptor u, and a set of fields and methods within its scope. A field declaration F simply defines a field named f with a type t. In the case of a field of an object reference, type t is dynamic, which means that it may vary at runtime. The

$$
\begin{array}{rrcl}
\text{(class declarations)} & D & ::= & \mathsf{class}\ C\ \{u; \vec{F}; \vec{M}\} \\
\text{(field declarations)} & F & ::= & t\ f \\
\text{(method declarations)} & M & ::= & s\ t\ m(t\ x)\ \{e\} \\
\text{(method qualifiers)} & s & ::= & \varepsilon \mid \mathsf{sync} \\
\text{(types)} & t & ::= & \mathsf{unit} \mid \mathsf{boolean} \mid C[u] \\
\text{(values)} & v & ::= & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid o \\
\text{(expressions)} & e & ::= & v \mid x \mid o.f \\
 & & & \mid e; e \mid o.f = e \\
 & & & \mid \mathsf{new}\ C() \mid o.m(e) \mid o.f.m(e) \\
 & & & \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \mid \mathsf{while}\ (e)\ e \\
 & & & \mid \mathsf{spawn}\ e \\
\text{(class usage types)} & u & ::= & q\ \{m_i; u_i\}_{i \in I} \mid \langle u + u \rangle \mid \\
 & & & \mid X \mid \mu X.u \\
\text{(type qualifiers)} & q & ::= & \mathsf{lin} \mid \mathsf{un}
\end{array}
$$

Figure 3.1: User syntax

general syntax for defining a method declaration M is also standard, denoting a method named m that returns a value of type t, taking a parameter x with type t. The body of the method is represented by the expression e.

Method modifiers are optional, which we indicate in the syntax by the empty string ε. A method declaration can be annotated with the sync qualifier to prevent race conditions from arising when separate threads attempt concurrent calls on the same method. This means that a sync method is evaluated in a similar way to Java's synchronized methods: the object for which the method is invoked has an associated monitor lock, that is acquired by the synchronized method before its body is executed [19].

Types and Values The MOOL syntax defines non-object and object types. Non-object types include the primitive unit and boolean types. The unit type has the single value unit, while the boolean type can have one of two values: true and false.

The type of an object is $C[u]$ in the style of modular session types [15], ranging over a class named C with usage type u. From a client class perspective, it prescribes the structure of method invocation: in which order/how methods should be called. Notice that only methods (not fields) are visible from outside, since methods are in the object type (and fields are not), as discussed in the examples from the previous chapters. It follows that methods not referred in the type of an object can be regarded as *private* (*cf.* [15, 38]), even though our language does not support access level modifiers.

Expressions The MOOL expressions are standard in object-oriented languages. We have defined (in order of appearance) values, parameters, fields, the sequential expression composition, assignment to fields, object creation, the self-call, the method call on fields, control flow expressions, and thread creation.

Values and references have been introduced above. Assignments, object creation and method calls have a syntax similar to the one found in most object-oriented languages, except for the aforementioned restrictions. Conditional and loop expressions are standard. Thread creation uses the spawn primitive. The form of the spawn expression means that the inner expression e will be evaluated in a newly created thread.

Class usage types In the core language, only the abstract syntax of usage types is defined, omitting the **usage** and **where** clauses, and the branch choice (+) and recursive (*) operators that have been introduced in the examples. However, it is not difficult to translate from the syntax of the examples into the core language syntax. The recursive type of the Auctioneer with the form **usage lin** init ; $*\{$selling + bidding$\}$; (see Section 2.2.2) takes the following configuration in the core language:

$$\mathsf{lin}\ \{init; \mu X.\mathsf{un}\ \{selling.X, bidding.X\}\}$$

The usage type defined in class Selling (see Section 2.2.3) provides an example of a variant type. The form **usage lin** init ; Sold **where** Sold = **lin** sold; $\langle$getPrice; **end** + done; **end**$\rangle$; is equivalent in the core language syntax to:

$$\mathsf{lin}\ \{init; \mathsf{lin}\ \{sold; \langle \mathsf{lin}\ \{getPrice; \mathsf{un}\ \{\}\} + \mathsf{lin}\ \{done; \mathsf{un}\ \{\}\}\rangle\}\}$$

Branch types are denoted by $\{m_i; u_i\}_{i \in I}$, defining available methods m_i and their continuations u_i. Variant types are denoted by $\langle u + u \rangle$, and are indexed by the set of the two boolean values returned by the preceding method in the specified usage type. In order to resolve the type, the caller must perform a test on its result before calling further methods. For simplicity, we use binary-only variants as in typestates [32], but more generous variants using enumerations can be found in the literature [15]. Recursive types are indicated in the syntax by $\mu X.u$, and need to be *contractive*, which means that no subexpression of the form $\mu X_1. \cdots \mu X_n.X_1$ is allowed [15]. These types are treated in an equi-recursive discipline, which means that we regard $\mu X.u$ and its unrolling $u\{^{\mu X.u}/_X\}$ as equal types.

A usage branch type is annotated with a qualifier q for aliasing control. The lin qualifier describes the status of objects that can be referenced by a single client. The un qualifier stands for unrestricted, or shared, and governs the status of objects that can be referenced in multiple threads. Recall that we define a single category for objects, as opposed to distinct categories for linear and for unrestricted objects, allowing linear objects to evolve into a status where they can be shared by multiple clients (*cf.* [35, 37]). The opposite is not possible, as we do not keep track of the number of references to a given object.

$$
\begin{array}{rrcl}
\text{(values)} & v & ::= & \ldots \mid \mathsf{uninit}_{\mathsf{C[u]}} \\
\text{(expressions)} & e & ::= & \ldots \mid \mathsf{insync}\ o\ e \\
\text{(contexts)} & \mathcal{E} & ::= & [\_] \mid \mathcal{E}; e \mid o.f = \mathcal{E} \\
 & & & \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E}) \\
 & & & \mid \mathsf{if}\ (\mathcal{E})\ e\ \mathsf{else}\ e \mid \mathsf{while}\ (\mathcal{E})\ e \\
\text{(lock flags)} & l & ::= & 0 \mid 1 \\
\text{(object records)} & R & ::= & (t, l, \vec{f} = \vec{v}) \\
\text{(heaps)} & h & ::= & \emptyset \mid h, o = R \\
\text{(states)} & S & ::= & (h;\ e_1 | \ldots | e_n)
\end{array}
$$

Figure 3.2: Runtime syntax

Notice that the type system, which we introduce later in this chapter, imposes some further restrictions on usage types, namely that:

(i) the initial class usage type can only be a branch, because a variant type is bound to the result of a method call;

(ii) in a variant type configuration $\langle u + u \rangle$, each u corresponds to a lin-qualified branch (and that is why the qualifier is always omitted); and

(iii) an object final status is always unrestricted, whether because it evolves into an end state (with an empty set of alternatives), or because, being recursive, it never ends, as illustrated in the examples.

3.1.2 Runtime syntax

Figure 3.2 introduces additional elements required by the type system and operational semantics, but not available in the user syntax.

In the runtime syntax, *values* include $\mathsf{uninit}_{\mathsf{C[u]}}$, not in the programmer syntax, which is used by our compiler to initialise fields of type $C[u]$ when an object is created. *Expressions* are extended with the insync expression in the style of Flanagan and Abadi [12], that denotes that the subexpression e is currently being evaluated while the lock is held on object o.

Contexts, *lock flags*, *object records*, *heaps* and *states* presented next are required by the operational semantics. Evaluation contexts are denoted by $\mathcal{E}$, and are expressions with one hole. Contexts specify the order in which expressions are evaluated, defining where reduction rules can be applied. The contexts which we define ensure that all field accesses and method calls occur after the receiver has been resolved, and that sequences

of expressions are evaluated in a specific order, namely from left to right. In the case of control flow expressions, the contexts guarantee that a branch in a conditional construct can only be executed after the condition has reduced to a value of a boolean type, and the same is guaranteed in the execution of the body of a while loop.

A heap is viewed as a map (a partial function of finite domain) from object identifiers o into records R. The operation $h, o = R$ adds an entry to the heap h, if o does not yet exist, and is considered to be associative and commutative, which means that a heap is an unordered set of bindings [15]. Object records are instances of usage-typed classes and are represented by the triple $(t, l, \vec{f} = \vec{v})$, where t is the object current type, l represents the class lock flag, and $\vec{f} \mapsto \vec{v}$ is a mapping from field identifiers to their values. The *lock* can be either in a locked or unlocked state, denoted by the 1 and 0 flag, respectively. The record $(\_, 1, \_)$ means that some thread currently holds the lock associated with the current instance. Initially, an object is created with the flag set to 0. States consist of two components: a heap and a parallel composition of expressions sharing the same heap.

3.1.3 Programs

A MOOL program is composed of a collection of class declarations, which include the program entry point – a starting class named Main with a special method called main. A program starts with an initial state of the form $(o : \mathsf{Main}[u], 0, \vec{f} = \mathsf{init}(\vec{t})), o.\mathsf{main}()$. Intuitively, though not syntactically correct, we could simplify and just say that a program starts with the expression new Main().main().

In the rest of this chapter, we present the reduction and typing rules of the MOOL core language. We use the notation in Figure 3.3 to access the components defined in class declarations.

3.2 Operational Semantics

An operational semantics for MOOL is described in this section, modelling program execution. The rules are defined in terms of state transitions. The evaluation of configurations starts with an empty heap and a set of expressions, and proceeds according to the various rules that specify the behaviour of each expression.

To simplify the reduction rules, we define a set of operations on the heap. If o is in the domain of h, then $h(o)$ returns the record R associated with o in h. If $R = (C[u], l, \vec{f} = \vec{v})$, the auxiliary functions in Figure 3.3 are used to access and update object record components.

The functions used to access elements of object records are straightforward. Functions for record update are used to modify an object component. We show how lock and field values are updated. In both cases, the operation changes the initial heap h_1 into the final

Accessing components in class $C\ \{u; \vec{F}; \vec{M}\}$

- $C.\mathsf{usage} = u$
- $C.\mathsf{fields} = \vec{F}$
- $C.\mathsf{methods} = \vec{M}$

Accessing components in record $(C[u], l, \vec{f} = \vec{v})$

- $h(o).\mathsf{class} = C$
- $h(o).\mathsf{usage} = u$
- $h(o).\mathsf{lock} = l$
- $h(o).f_i = v_i$

Updating record components in heap $h = h_1, o : (C[u], l, \vec{f} = \vec{v}), h_2$

- $h\{o.\mathsf{lock} \mapsto l'\} = h_1, o : (C[u], l', \vec{f} = \vec{v}), h_2$
- $h\{o.f_i \mapsto v\} = h_1, o : (C[u], l, (\vec{f_1} = \vec{v}_1, f_i = v, \vec{f_n} = \vec{v}_n)), h_2$ where $\vec{f} = f_1 \ldots f_n$ and $\vec{v} = v_1 \ldots v_n$

Figure 3.3: Auxiliary functions for class and heap components

one h_2. We also use the notation $e\{^o/_{\mathsf{this}}\}\{^v/_x\}$ to denote the substitution in expression e of this for o and of x for v.

The rest of the section is devoted to presenting the reduction rules for MOOL and explaining each one of them in detail. The only unusual rules are the ones that have to deal with linear control of objects; the remaining ones are typical of most object-oriented languages.

3.2.1 Reduction rules for states

In Figure 3.4 we define the reduction rules for states, and below we comment on each one of them. These rules take the general form:

$$(h;\ e_1|\ldots|e_n) \longrightarrow (h;\ e_1|\ldots|e_n)$$

$$\text{(R-CONTEXT)}\ \frac{(h;\ e_1|\ldots|e|\ldots|e_n) \longrightarrow (h';\ e_1|\ldots|e'|\ldots|e_n)}{(h;\ e_1|\ldots|\mathcal{E}[e]|\ldots|e_n) \longrightarrow (h';\ e_1|\ldots|\mathcal{E}[e']|\ldots|e_n)}$$

$$\text{(R-SPAWN)}\ \ (h;\ e_1|\ldots|\mathcal{E}[\mathsf{spawn}\ e]|\ldots|e_n) \longrightarrow (h;\ e_1|\ldots|\mathcal{E}[\mathsf{unit}]|e|\ldots|e_n)$$

Figure 3.4: Reduction rules for states

R-CONTEXT This rule is standard, defining which expression can be evaluated next in the program execution. It is the rule that invokes all the other reduction rules for expressions (see Figure 3.6).

R-SPAWN When spawn e is evaluated in an arbitrary context, the original thread does not evaluate the expression, but instead detaches it to a new thread running in parallel. After spawn, the original thread proceeds to evaluate its next instruction. The value of the spawn expression is unit, as the result of evaluating e is not transferred back to the original thread.

3.2.2 Reduction rules for expressions

The rules for expressions in Figure 3.6 have a similar form, with the difference that the state contains only one expression (as opposed to a parallel composition of expressions):

$$(h;\ e) \longrightarrow (h;\ e)$$

The rules use in their definitions the predicates defined in Figure 3.5. Predicate $q(v)$ determines the type status (linear or unrestricted) given a value v. If v is a value of a primitive type, then its status is always unrestricted. If v is an object, then its status depends on its current type, which can be fetch from the usage component of the record associated with o in heap h. A variant type is always linear, and the status of a branch type is defined by the programmer. Predicate $q(t)$ follows a similar pattern, with the difference that it takes a type as its argument. Finally, function $\mathsf{init}(t)$ takes a type as its argument and returns the type default value.

R-SEQ This rule reduces the result to the second part of the sequence of expressions, discarding the first part only after it has become a value.

R-LINFIELD, R-UNFIELD Access to a field involves extracting the value of the field from an object in the heap and determining whether the value belongs to an unrestricted

---

Predicates $\mathsf{lin}(v)$ **and** $\mathsf{un}(v)$ **with** q $::=$ lin | un

- if $v = \mathsf{unit}, \mathsf{true}, \mathsf{false}$ then $\mathsf{un}(v)$
- if $v = o$ and $h(o).\mathsf{usage} = \langle \_ + \_ \rangle$ then $\mathsf{lin}(v)$
- if $v = o$ and $h(o).\mathsf{usage} = q\ \{\ldots\}$ then $q(v)$

Predicates $\mathsf{lin}(t)$ **and** $\mathsf{un}(t)$ **with** q $::=$ lin | un

- if $t = \mathsf{unit}, \mathsf{boolean}$ then $\mathsf{un}(t)$
- if $t = C[\langle \_ + \_ \rangle]$ then $\mathsf{lin}(t)$
- if $t = C[q\ \{\ldots\}]$ then $q(t)$

Function $\mathsf{init}(t)$

- $\mathsf{init}(\mathsf{boolean}) = \mathsf{false}$
- $\mathsf{init}(\mathsf{unit}) = \mathsf{unit}$
- $\mathsf{init}(C[u]) = \mathsf{uninit}_{\mathsf{C[u]}}$

Figure 3.5: Auxiliary functions for values and types

or a linear type, using functions $\mathsf{un}(v)$ and $\mathsf{lin}(v)$, respectively (defined in Figure 3.5). After access, the value of an unrestricted field is v as in Java, but the value of a linear field must be unit to ensure that only one reference to an object with a linear type exists at any given moment.

R-ASSIGN Assignment to a field updates the value of the field with value v. The value of the entire expression needs to be unit (as opposed to v), and this is again a linearity constraint to enforce that a reference to an object with a linear type goes out of scope after appearing on the right-hand side of an assignment.

R-NEW The creation of a new object requires generating a fresh object identifier and obtaining the field names and types from the set of the class fields. Notice that a new object record, indexed by object identifier o, is added to the heap. Also notice that the object starts in an unlocked state 0, and that fields are initialised to default values based on their declared types (see Figure 3.5).

$$\text{(R-SEQ)}\ \ (h;\ v; e) \longrightarrow (h;\ e)$$

$$\text{(R-LINFIELD)}\ \frac{h(o).f = v \qquad \mathsf{lin}(v)}{(h;\ o.f) \longrightarrow (h\{o.f \mapsto \mathsf{unit}\};\ v)}$$

$$\text{(R-UNFIELD)}\ \frac{h(o).f = v \qquad \mathsf{un}(v)}{(h;\ o.f) \longrightarrow (h;\ v)}$$

$$\text{(R-ASSIGN)}\ \ (h;\ o.f\ =\ v) \longrightarrow (h\{o.f \mapsto v\};\ \mathsf{unit})$$

$$\text{(R-NEW)}\ \frac{o\ \text{fresh} \qquad C.\mathsf{fields} = \vec{t}\,\vec{f}}{(h;\ \mathsf{new}\ C()) \longrightarrow (h, o = (C, C.\mathsf{usage}, 0)\vec{f} = \mathsf{init}(\vec{t});\ o)}$$

$$\text{(R-SELFCALL)}\ \frac{\_\, m(\_\, x)\ \{e\} \in (h(o).\mathsf{class}).\mathsf{methods}}{(h;\ o.m(v)) \longrightarrow (h;\ e\{^{o}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-CALL)}\ \frac{m \in h(o).f.\mathsf{usage} \qquad \_\, m(\_\, x)\ \{e\} \in (h(o).f.\mathsf{class}).\mathsf{methods}}{(h;\ o.f.m(v)) \longrightarrow (h;\ e\{^{o.f}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-SCALL)}\ \frac{\begin{array}{c} h(o).f.\mathsf{lock} = 0 \qquad m \in h(o).f.\mathsf{usage} \\ \mathsf{sync}\ \_\, m(\_\, x)\ \{e\} \in (h(o).f.\mathsf{class}).\mathsf{methods} \end{array}}{(h;\ o.f.m(v)) \longrightarrow (h\{(o).f.\mathsf{lock} \mapsto 1\};\ \mathsf{insync}\ o'\ e\{^{o.f}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-INSYNC)}\ \ (h;\ \mathsf{insync}\ o\ v) \longrightarrow (h\{(o).\mathsf{lock} \mapsto 0\};\ v)$$

$$\text{(R-IFTRUE)}\ \ (h;\ \mathsf{if}\ (\mathsf{true})\ e'\ \mathsf{else}\ e'') \longrightarrow (h;\ e')$$

$$\text{(R-IFFALSE)}\ \ (h;\ \mathsf{if}\ (\mathsf{false})\ e'\ \mathsf{else}\ e'') \longrightarrow (h;\ e'')$$

$$\text{(R-WHILE)}\ \ (h;\ \mathsf{while}\ (e)\ e') \longrightarrow (h;\ \mathsf{if}\ (e)\ \{e'; \mathsf{while}\ (e)\ e'\}\ \mathsf{else}\ \mathsf{unit})$$

Figure 3.6: Reduction rules for expressions

R-SELFCALL, R-CALL The first rule is for method calls directly on an object reference, which typically result from calls on this. These calls do not depend on the usage type, unlike rule R-CALL. In the latter case, the invoked method not only has to be defined in the class of the receiver, but it must also be referred in receiver usage type. Notice that both rules are for non-synchronized method calls. For synchronized calls we have defined a specific rule (R-SCALL), which we introduce below. In either case, if the hypothesis holds, then the method body is prepared by replacing occurrences of this by the receiver heap address (so that occurrences of this within the method body refer to the receiver instance) and the actual parameter by the formal one. The resulting expression is the method body with the above substitutions that can be further reduced using the appropriate rules.

R-SCALL, R-INSYNC These two rules are used when synchronizing method calls. They were inspired by the operational semantics proposed by Flanagan and Abadi [12] for a concurrent, imperative language. The new element here, as opposed to the standard reduction rules for method calls above (R-SELFCALL and R-CALL), is the lock acquired on the receiver object. The method body reduces to the insync expression that indicates that the method body specified by e is currently being evaluated as a synchronized block. Rule R-INSYNC says that once the method returns a value, the lock on the object is released, and this we represent by updating the lock component with the flag 0.

R-IFTRUE, R-IFFALSE These two rules implement the conditional construct, making each branch depend on the boolean value that controls the condition, and which has been reduced by other rules.

R-WHILE The behaviour of the while loop is defined by rewriting it to a nested conditional expression. Iteration is reproduced by evaluating the condition within a conditional: if the test succeeds, an iteration will be performed; if it fails, the iteration is cancelled, and execution proceeds to an empty (unit) branch.

3.3 Subtyping

The subtyping definition for the MOOL language is similar to the one described for the language in [15]. The subtype usage relation is coinductively defined as follows, where $<:$ stands for the largest relation on class usage types:

- If $\{m_i : u_i\}_{i \in I} <: u'$ then $u' = \{m_j : u'_j\}_{j \in J}$ with $J \subseteq I$ and $\forall j \in J, u_j <: u'_j$
- If $\langle u' + u'' \rangle <: u$ then $u = u'$ with $u' <: u$ or $u = u''$ with $u'' <: u$

The branch type is the source of subtyping by allowing that the possible usage choices can safely be used in the context where the superusage is expected. The proof that the usage relation is a pre-order (i.e. a relation that is reflexive and transitive) is not provided here, but can be adapted from [16].

The subtyping usage rule relation is extended to the types of our language as the smallest reflexive relation which includes the following:

- $C[u] <: C[u']$ if $u <: u'$

This means that implicitly the subtyping relation for the two other types in our language is defined as boolean $<:$ boolean and unit $<:$ unit.

3.4 Type System

The purpose of any type system is to filter some of the programs that are generated by the context-free grammar, seeking to guarantee that these programs make sense in view of a set of well-defined rules. In MOOL, type safety is achieved by making sure that:

(i) client code calls methods in the order specified in the class usage type;

(ii) client code tests method results, if applicable, before proceeding to the next call; and

(iii) references to linear objects are consumed to the end before being discarded.

The type checker relies on a set of typing rules that formalise these properties. Most of the rules are adapted from modular session types [15]. The reconstruction of session types in [37] gave us insight into how to approach lin/un qualifiers.

The inference rules that define the type system use two typing environments, Θ and Γ. We consider typing environments as maps (or partial functions of finite domain) similar to heaps. The typing environment Θ is a map from usage types u to typing environments Γ. It records the field typing environment associated with usage type u. It is used to keep track of visited usage types, thus preventing cycles in the presence of recursive types.

The typing assumptions for environment Γ have the form:

$$\Gamma \ ::= \ \Sigma \ | \ \langle \Gamma + \Gamma \rangle$$

where Σ is a map from fields $o.f$ and parameters x to types t. Environment $\langle \Gamma + \Gamma \rangle$ represents a pair of maps: the map on the left is used for the true variant, while the map on the right is for the false one.

Below are the typing judgements where these environments are used. The typing judgement for usage types is presented on the left, and the one on the right is used for expressions:

$$\Theta; \Gamma \triangleright_C u \triangleleft \Gamma' \qquad\qquad \Gamma \triangleright e : t \triangleleft \Gamma'$$

$$\text{(T-PROGRAM)}\ \frac{\vdash D_1 \quad \ldots \quad \vdash D_n}{\vdash D_1 \ldots D_n}$$

$$\text{(T-CLASS)}\ \frac{\emptyset; \mathsf{this}.\vec{f} : \vec{t} \triangleright_C u \triangleleft \Sigma \qquad \mathsf{un}(\Sigma)}{\vdash \mathsf{class}\ C\ \{u; \vec{t}\ \vec{f}; \_\}}$$

Figure 3.7: Typing rules for programs

The judgements have two parts: the input of the judgement (on the left-hand side of the first triangle) and its output (on the right-hand side of the second triangle). In the centre is the syntactic construct being evaluated.

The typing judgement for usage types reads: "Usage type u is valid for class C starting from field and parameter types in Γ, the initial environment, and ending with field and parameter types in Γ', the final environment". The initial and final environments may be different, because object field types in Γ may have changed after each method described in u has been type-checked.

The typing judgement for expressions follows a similar pattern. Γ is the initial typing environment before type-checking expression e, and Γ' is the final one, after associating e with type t. Again, the initial and final environments may be different, because of side-effects the expression being checked may cause on environment Γ, turning it into environment Γ'. In particular, identifiers may become unavailable in the case of references to linear objects, and object types may change as a result of the evaluation of some language constructs, namely method calls and control flow expressions.

Using a similar notation to the one used in Section 3.2 to update the heap, if $\Gamma(o.f)$ is defined, i.e. if $o.f \in dom(\Gamma)$, then $\Gamma\{o.f \mapsto t\}$ denotes the environment obtained by updating the type of field $o.f$ to type t.

3.4.1 Type checking programs

The typing rules defined for MOOL have a clear algorithmic configuration. They should be read starting from the judgement in the conclusion, and considering that it holds given the specified premises. Moreover, type-checking is modular and performed following a top-down strategy: a program checking is conducted by checking each class separately, which in turn conducts the checking of each method within the class *in the order in which it appears in the specified usage type*. In what follows, rules are presented in the order in which they are evaluated (and in the order in which they should be read).

Figure 3.7 defines rules for typing programs, and below we comment on them.

T-PROGRAM When checking that a program is well-formed, this rule simply checks that each class defined in it is well-formed.

T-CLASS The rule for typing classes constructs the field arguments in the initial typing environment[1]. Notice that when starting to type-check a class, the environment Θ is empty as no usage types have yet been visited. T-CLASS is responsible for calling the rules for type-checking usage type constructs (see Figure 3.8). After type-checking u, the object fields in Σ are a subset of the original ones, that have been consumed, inside the object, to the end. Those which are missing have been passed as parameters, or returned from methods, and thus removed from the environment. This means that the object fields in a class final environment have always unrestricted types, either recursive or un end. Function $\mathsf{un}(\Sigma)$ recursively calls predicate $q(t)$ in Figure 3.5, passing each field type in Σ as argument, in order to guarantee that no linear fields exist in the class final environment.

3.4.2 Type checking classes

The rules in Figure 3.8 describe how the four usage constructs (*cf.* Figure 3.1) are typed in MOOL. In Figure 3.9, we exemplify in a derivation tree how the rules are applied to a slightly simplified FileReader usage type from Section 1.1, starting at state Next. To lighten the representation, we drop the subscript in the usage type judgements, which identifies the class being type-checked. Γ_1 is the initial environment, containing the field f typing when a FileReader object is at state Next. Field f references a linear File object, and its type (omitted) changes throughout the FileReader class type-checking, altering the class environment configuration. Environments Γ_1 to Γ_4 represent these changes. Θ_1 and Θ_2 keep track of visited usage types by storing references to recursive states Next and Done, mapped to their respective environments. Because the other two fields in class FileReader (s and counter) have standard, non-object types, that remain unchanged throughout the class type-checking, we simplify and omit them from the environments represented in the derivation.

The example makes a reference to the derivation in Figure 3.13, where we illustrate the method next body being type-checked and revealing how f changes its type as the File type is consumed. T-IFV final environment Γ_2 (in the form of a pair of environments) is T-BRANCH initial one. The true environment Γ_1 is chosen for the type-checking of those methods described in state Next, while the false one Γ_4 is chosen for the type-checking of the methods specified in state Done. The class final environment is empty as field f has been removed by T-BRANCH after its type has been consumed to the end in the method next body. The example establishes that μNext.**lin** next;... is a valid usage type by directly applying the appropriate usage type rule at each step in the derivation tree. The descriptions that we provide below clarify the semantics of the rules.

[1]The initial typing environment cannot be Γ, because variant types are bound to the result of a method call, as discussed in Section 3.1.1.

$$(\text{T-BRANCH})\ \frac{\forall i \in I \qquad s_i\ t_i\ m_i(t'_i\ x_i)\ \{e_i\} \in C.\mathsf{methods} \qquad \Sigma, x_i : t'_i \triangleright e_i : t_i \triangleleft \Gamma \quad x_i : t''_i \in \Gamma \Rightarrow \mathsf{un}(t''_i) \quad \Gamma = \langle \_ + \_ \rangle \Rightarrow t_i = \mathsf{boolean} \quad \Theta; \Gamma \backslash x_i \triangleright_C u_i \triangleleft \Gamma'}{\Theta; \Sigma \triangleright_C q\ \{m_i; u_i\} \triangleleft \Gamma'}$$

$$(\text{T-VARIANT})\ \frac{\Theta; \Gamma' \triangleright_C u_t \triangleleft \Gamma \qquad \Theta; \Gamma'' \triangleright_C u_f \triangleleft \Gamma}{\Theta; \langle \Gamma' + \Gamma'' \rangle \triangleright_C \langle u_t + u_f \rangle \triangleleft \Gamma}$$

$$(\text{T-USAGEVAR})\ \ (\Theta, X : \Gamma); \Gamma \triangleright_C X \triangleleft \Gamma'$$

$$(\text{T-REC})\ \frac{(\Theta, X : \Gamma); \Gamma \triangleright_C u \triangleleft \Gamma'}{\Theta; \Gamma \triangleright_C \mu X.u \triangleleft \Gamma'}$$

Figure 3.8: Typing rules for classes

T-BRANCH The rule for branch usage types conducts the type-checking of the methods defined in the class usage type in the order in which they appear in the specified sequence. The appropriate rules for expressions (see Figure 3.11) are called by T-BRANCH so that each method body e_i can be type-checked.

The initial environment is a single map of fields and parameters, denoted by Σ, while the final one may be a pair of maps if the method m_i returns a value of type boolean. Thus Γ is used instead. When exiting the method body, the parameter type in Γ may have changed from the initial type t'_i to the final one t''_i. Additionally, the rule requires that t''_i is unrestricted, so function $\mathsf{un}(t''_i)$ checks that the parameter has been consumed to the end. Finally, the parameter entry must be removed from environment Γ before the rule advances to type-checking the method continuations u_i, where again the field typing may change, modifying the initial environment Γ to the final one Γ'.

Extending the rule to methods accepting multiple parameters means simply that each parameter in the environment is recursively checked, and recursively removed from the final environment afterwards.

T-VARIANT In checking variant types, the rule requires that there is a consistency between the variants final environment Γ, after passing Γ', the left environment, to type the true variant, and Γ'', the right environment, to type the false variant.

T-USAGEVAR, T-REC The first rule, T-USAGEVAR, simply reads the type variable X from map Θ. Because X represents an infinite branch in the usage type tree that no program can ever consume to the end (see an example in states Next and Done in Figure 3.9), the rule cannot define the final environment as being the same as the initial one, which

```
class FileReader {
  usage lin open; Next
    where Next = lin next; ⟨Next + Done⟩;
          Done = *{toString + getCounter};
  ...
}
```

$$(*)\ \frac{\vdots}{\Gamma_4 \triangleright \text{getCounter} : \text{int} \triangleleft \Gamma_4} \qquad \frac{}{\Theta_2, \Gamma_4 \triangleright \text{Done} \triangleleft \emptyset}\ (\text{T-UsageVar})$$

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma_4 \triangleright \text{toString} : \text{string} \triangleleft \Gamma_4} \qquad \dfrac{}{\Theta_2, \Gamma_4 \triangleright \text{Done} \triangleleft \emptyset}\ (\text{T-UsageVar}) \qquad (*)}{\Theta_2, \Gamma_4 \triangleright \{\text{toString; Done + getCounter; Done}\} \triangleleft \emptyset}\ (\text{T-Branch})}{\Theta_1; \Gamma_4 \triangleright \mu\text{Done}.\{\text{toString; Done + getCounter; Done}\} \triangleleft \emptyset}\ (**)\ (\text{T-Rec})$$

$$\frac{\dfrac{\dfrac{\text{(see Fig. 3.13)}}{\Gamma_1 \triangleright \textbf{if} \ldots : \ldots \triangleleft \Gamma_2}\ (\text{T-IfV}) \qquad \dfrac{\dfrac{}{\Theta_1; \Gamma_1 \triangleright \text{Next} \triangleleft \emptyset}\ (\text{T-UsageVar})}{\Theta_1; \Gamma_2 \triangleright \langle \text{Next} + \text{Done} \rangle \triangleleft \emptyset}\ (**)\ (\text{T-Variant})}{\Theta_1; \Gamma_1 \triangleright \textbf{lin}\ \text{next}; \ldots \triangleleft \emptyset}\ (\text{T-Branch})}{\emptyset; \Gamma_1 \triangleright \mu\text{Next}.\textbf{lin}\ \text{next}; \ldots \triangleleft \emptyset}\ (\text{T-Rec})$$

- $\Theta_1 = \text{Next} : \Gamma_1$
- $\Theta_2 = \Theta_1, \text{Done} : \Gamma_4$
- $\Gamma_1 = \textbf{this}.\text{f} : \text{File[Read]}$
- $\Gamma_2 = \langle \Gamma_1 + \Gamma_4 \rangle = \langle \textbf{this}.\text{f} : \text{File[Read]} + \textbf{this}.\text{f} : \text{File}[\textbf{end}] \rangle$
- $\Gamma_3 = \textbf{this}.\text{f} : \text{File}[\langle \text{close; } \textbf{end} + \text{read; Read} \rangle]$
- $\Gamma_4 = \textbf{this}.\text{f} : \text{File}[\textbf{end}]$

Figure 3.9: Example of the FileReader usage type derivation

$$\text{(T-UNIT)}\ \ \Gamma \triangleright \mathsf{unit} : \mathsf{unit} \triangleleft \Gamma$$

$$\text{(T-TRUE)}\ \ \Gamma \triangleright \mathsf{true} : \mathsf{boolean} \triangleleft \Gamma$$

$$\text{(T-FALSE)}\ \ \Gamma \triangleright \mathsf{false} : \mathsf{boolean} \triangleleft \Gamma$$

Figure 3.10: Typing rules for values

means that the final environment can be whichever we want. The rule for recursive usage types, T-REC, not only reads the type from map Θ, but also checks the type against the class, using T-USAGEVAR to type usage variables X, and the other two rules, T-BRANCH and T-VARIANT, to type branch and variant constructs. The final environment Γ' may be different, because field types may have changed, after type-checking expressions in method bodies (recall that T-BRANCH calls the appropriate rules for expressions).

3.4.3 Type checking expressions

We now proceed to the rules for expressions. In order to make them more readable, we use the single environment Σ everywhere a map is needed to index a field or a parameter; otherwise the more general environment Γ is used. Figure 3.10 types values. The rules are mostly standard, evaluating the types that values can assume.

Figures 3.11 and 3.12 define the rules for the remaining expressions of the top level language. The rest of the section is devoted to discussing each one of them in detail.

T-SEQ Typing the sequential composition of expressions is simple: the rule considers the typing of the second subexpression taking into account the effects of the first one on the environment, which may be different when e represents a method call or a control flow expression.

T-LINVAR, T-UNVAR These two rules evaluate the types of method parameters, distinguishing linear from unrestricted ones. Recall that, as we cannot call methods on parameters, they must be first assigned to fields. In the case of parameters with linear types, the type system requires that they are removed from the environment, so that they can no longer be used after assignment. Predicates $\mathsf{lin}(t)$ and $\mathsf{un}(t)$, which have been defined earlier in Figure 3.5, are used to determine the status of a given type. In practice, primitive types (unit and boolean) are always unrestricted; for object types the current status varies with the type.

$$\text{(T-SEQ)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma'' \qquad \Gamma'' \triangleright e' : t' \triangleleft \Gamma'}{\Gamma \triangleright e; e' : t' \triangleleft \Gamma'}$$

$$\text{(T-LINVAR)}\ \frac{\mathsf{lin}(t)}{\Sigma, x : t \triangleright x : t \triangleleft \Sigma} \qquad \text{(T-UNVAR)}\ \frac{\mathsf{un}(t)}{\Sigma, x : t \triangleright x : t \triangleleft \Sigma, x : t}$$

$$\text{(T-LINFIELD)}\ \frac{\mathsf{lin}(t)}{\Sigma, o.f : t \triangleright o.f : t \triangleleft \Sigma} \quad \text{(T-UNFIELD)}\ \frac{\mathsf{un}(t)}{\Sigma, o.f : t \triangleright o.f : t \triangleleft \Sigma, o.f : t}$$

$$\text{(T-ASSIGN)}\ \frac{\Sigma \triangleright e : t \triangleleft \Sigma' \qquad \Sigma'(o.f) = t \qquad \mathsf{un}(t)}{\Sigma \triangleright o.f \ = \ e : \mathsf{unit} \triangleleft \Sigma'}$$

$$\text{(T-NEW)}\ \ \Gamma \triangleright \mathsf{new}\ C() : C[C.\mathsf{usage}] \triangleleft \Gamma$$

$$\text{(T-SPAWN)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma' \qquad \mathsf{un}(t)}{\Gamma \triangleright \mathsf{spawn}\ \ e : \mathsf{unit} \triangleleft \Gamma'}$$

Figure 3.11: Typing rules for simple expressions

T-LINFIELD, T-UNFIELD The rules for fields are similar to the ones defined above for parameters, removing fields with linear types from the environment.

T-ASSIGN This rule formalises assignments to fields. The type of the reference and the expression must be consistent and, because of linearity, it must be unrestricted. The type of the entire assignment expression is unit (as opposed to t) to enforce that a linear reference goes out of scope after appearing on the right-hand side of the assignment.

T-NEW The rule for object creation simply states that a new object has the initial usage type declared by its class.

T-SPAWN The type-checking rule for a spawn expression requires not only that the thread body is typable, but also that it is not of a linear type; otherwise one could create a linear reference and not use it to the end (suppose we wrote **spawn new** C()). The type of the entire spawn expression is unit, because the evaluation is performed only for its effect (creating a new thread); no result is ever returned (recall the reduction rule in Section 3.2.1).

$$\text{(T-CALL)}\ \frac{\Gamma \triangleright e : t \triangleleft \Sigma \qquad \Sigma(o.f) = C[\_\ \{m_i; u_i\}_{i \in I}] \qquad j \in I \qquad t\ m_j(t\,x)\ \{\_\} \in C.\mathsf{methods}}{\Gamma \triangleright o.f.m_j(e) : t \triangleleft \Sigma\{o.f \mapsto C[u_j]\}}$$

$$\text{(T-IFV)}\ \frac{\Gamma \triangleright o.f.m(e) : \mathsf{boolean} \triangleleft \Sigma \qquad \Sigma(o.f) = C[\langle u_t + u_f \rangle] \qquad \Sigma\{o.f \mapsto C[u_t]\} \triangleright e' : t \triangleleft \Gamma' \qquad \Sigma\{o.f \mapsto C[u_f]\} \triangleright e'' : t \triangleleft \Gamma'}{\Gamma \triangleright \mathsf{if}\ (o.f.m(e))\ e'\ \mathsf{else}\ e'' : t \triangleleft \Gamma'}$$

$$\text{(T-WHILEV)}\ \frac{\Gamma \triangleright o.f.m(e) : \mathsf{boolean} \triangleleft \Sigma \qquad \Sigma(o.f) = C[\langle u_t + u_f \rangle] \qquad \Sigma\{o.f \mapsto C[u_t]\} \triangleright e' : t \triangleleft \Gamma}{\Gamma \triangleright \mathsf{while}\ (o.f.m(e))\ e' : \mathsf{unit} \triangleleft \Sigma\{o.f \mapsto C[u_f]\}}$$

$$\text{(T-IF)}\ \frac{\Gamma \triangleright e : \mathsf{boolean} \triangleleft \Gamma' \qquad \Gamma' \triangleright e' : t \triangleleft \Gamma'' \qquad \Gamma' \triangleright e'' : t \triangleleft \Gamma''}{\Gamma \triangleright \mathsf{if}\ (e)\ e'\ \mathsf{else}\ e'' : t \triangleleft \Gamma''}$$

$$\text{(T-WHILE)}\ \frac{\Gamma \triangleright e : \mathsf{boolean} \triangleleft \Gamma' \qquad \Gamma' \triangleright e' : t \triangleleft \Gamma}{\Gamma \triangleright \mathsf{while}\ (e)\ e' : \mathsf{unit} \triangleleft \Gamma'}$$

$$\text{(T-INJL)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma'}{\Gamma \triangleright e : t \triangleleft \langle \Gamma' + \Gamma'' \rangle} \qquad \text{(T-INJR)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma''}{\Gamma \triangleright e : t \triangleleft \langle \Gamma' + \Gamma'' \rangle}$$

$$\text{(T-SUB)}\ \frac{\Gamma \triangleright e : C[u] \triangleleft \Gamma' \qquad C[u] <: C[u']}{\Gamma \triangleright e : C[u'] \triangleleft \Gamma'} \quad \text{(T-SUBENV)}\ \frac{\Gamma \triangleright e : t \triangleleft \Sigma \qquad \Sigma <: \Sigma'}{\Gamma \triangleright e : t \triangleleft \Sigma'}$$

Figure 3.12: Typing rules for calls and control flow expressions

T-CALL Besides enforcing that the type of the argument matches the parameter type in the method signature, and that the method is defined in the set of the receiver methods, calls on fields require that the receiver has an appropriate usage type. The call results in the receiver changing its type to $C[u_j]$, with the final environment Σ being updated accordingly, while at the same time the type advances from $\{m_i; u_i\}$ to m_j.

T-IFV, T-WHILEV As opposed to the rules presented thus far, the remaining rules that we introduce in this section are not syntax-directed, which means that to implement them additional information is needed as they are evaluated. The next chapter informally describes the algorithm used in the implementation of the type-checker, explaining how that information is tracked.

To simplify the rules, for an object to have type variant, the method call on which the variant type depends must be performed on the condition of a control flow expression (and not on an arbitrary assignment). Recall that a variant type is tied to the result of a method and, depending on the value returned, an object type may be modified differently.

The type system requires the value to be tested in the condition of the expression, so as to have the type immediately deduced. We believe that this restriction does not impair current object-oriented programming practices, considering that what we propose reflects how a variant type is typically given to an object.

T-IFV and T-WHILEV are particular cases of the more general rules for control flow expressions that we describe below. The order in which we present them here corresponds to the actual order in which the type-checker evaluates them. Both rules depend on T-CALL to deduce the return type of the method tested on the condition.

In the case of T-IFV, both branches use Σ, the environment that results from the call, as their initial environment. The method returned value defines how $o.f$ type is updated, thus determining which branch (the then or the else) shall be executed. However, because only one of the branches can be executed, the rule enforces that the two branches should be consistent, sharing the same final environment Γ'.

T-WHILEV follows a similar pattern: a while expression can be reduced to an if that is repeated while a particular condition returns true. As in T-IFV, the rule for the loop expression requires that the method called on the condition has a boolean type, and that the receiver has a variant type. When the condition evaluates to true, the receiver type becomes $C[u_t]$ in Σ, the loop body is executed, while preserving at the end the original Γ. This is required because the condition continues being tested until it eventually evaluates to false. In the conclusion, the final environment Σ is modified as the receiver type is updated to the type of the false variant $C[u_f]$. The type of the entire loop expression is unit, because its result can never be used.

T-IF, T-WHILE These are standard rules for control flow expressions, and should be used only after the above rules have been tried.

T-INJL, T-INJR The two injection rules build a variant environment as follows: an environment Γ becomes $\langle \Gamma + \Gamma' \rangle$ by injecting Γ on the left using rule T-INJL, and becomes $\langle \Gamma' + \Gamma \rangle$ by injecting on the right using rule T-INJR.

Typically, these rules build the final environment of a boolean method bound to a variant type. The example in Figure 3.13 depicts the typing derivation of the method next body from the FileReader class in Section 1.1. So that the example can be followed easily, we accompany it with the FileReader usage type and the method implementation, as well as with the File usage type, allowing that the changes in field f of FileReader, which is bound to the File object, can be tracked in the respective usage specification.

T-IFV is at the root of the derivation tree evaluating the conditional expression corresponding to the method body, starting with the environment configuration represented by Γ_1. The example shows the environment configuration in which each subexpression within the conditional expression is typed, and how the field f typing changes as the derivation proceeds. Recall that the other two fields defined in class FileReader are omitted, as their non-object types remain unchanged throughout the type-checking.

Calling f.eof() changes the type of field f to the variant type in Γ_3, which is the initial environment of each of the if expression branches. Calling f.close() and f.read() changes again the type of f, differently in each call, resulting in two different environment configurations, Γ_4 and Γ_1, respectively. Using the appropriate injection rules, each environment is then injected into the pair of environments Γ_2, introducing consistency in the branches final environment, and thus in the conditional expression final environment. Recall from the derivation of Figure 3.13 presented earlier that Γ_2 is used as the input environment of rule T-VARIANT. In that example, we show the appropriate environment being chosen in order to proceed with the type-checking of the remaining methods of class FileReader, not yet type-checked.

T-SUB, T-SUBENV The two subsumption rules are similar to the ones described for the language presented in modular session types [15]. T-SUB is a standard subsumption rule that simply says that whenever we can prove that type t' is a subtype of t, we can treat t' as if it were type t. T-SUBENV allows subsumption in the final environment, and is used for the branches of control flow expressions.

Consider in Figure 3.14 class F usage type, a variation of the File usage type written exclusively for the sake of illustrating subsumption in MOOL. Class C defines a field f referencing an object of class F. The method m body represents the MOOL way of writing **if** (!f.eof) f.read(); After calling the condition f.eof(), there is a choice: the type of f can either advance to type F[⟨Close + read; Read⟩] or to F[Close]. In the example, the then

```
class File {
  usage lin open; Read
    where Read = lin eof; ⟨close; end + read; Read⟩;
    ...
}

class FileReader {
  usage lin open; Next
    where Next = lin next; ⟨Next + Done⟩;
          Done = *{toString + getCounter};

  File f;
  ...
  boolean next() {
    if(f.eof()) {
      f.close();
      false; // return false
    } else {
      s += f.read();
      true; // return true
    }
  }
  ...
}
```

$$(*)\ \frac{\vdots}{\Gamma_3 \triangleright \text{f.close()} : \textbf{unit} \triangleleft \Gamma_4}{\Gamma_5 \triangleright \text{f.close(); } \textbf{false} : \textbf{boolean} \triangleleft \Gamma_2} \qquad (**)\ \frac{\vdots}{\Gamma_3 \triangleright \text{f.read()} : \textbf{unit} \triangleleft \Gamma_1}{\Gamma_6 \triangleright \text{f.read(); } \textbf{true} : \textbf{boolean} \triangleleft \Gamma_2}$$

$$\frac{\dfrac{\cdots}{\Gamma_1 \triangleright \text{f.eof()} : \textbf{boolean} \triangleleft \Gamma_3}\ (\text{T-CALL}) \qquad (*) \qquad (**)}{\Gamma_1 \triangleright \textbf{if}\ (\text{f.eof()})\ \{\text{f.close(); ... }\}\ \textbf{else}\ \{\text{... f.read(); ... }\} : \textbf{boolean} \triangleleft \Gamma_2}\ (\text{T-IFV})$$

- $\Gamma_1 = \textbf{this}.\text{f} : \text{File[Read]}$
- $\Gamma_2 = \langle \Gamma_1 + \Gamma_4 \rangle = \langle \textbf{this}.\text{f} : \text{File[Read]} + \textbf{this}.\text{f} : \text{File[}\textbf{end}\text{]} \rangle$
- $\Gamma_3 = \textbf{this}.\text{f} : \text{File}[\langle \text{close; } \textbf{end} + \text{read; Read} \rangle]$
- $\Gamma_4 = \textbf{this}.\text{f} : \text{File[}\textbf{end}\text{]}$
- $\Gamma_5 = \textbf{this}.\text{f} : \text{File[close; }\textbf{end}\text{]}$
- $\Gamma_6 = \textbf{this}.\text{f} : \text{File[read; Read]}$

Figure 3.13: Example of a FileReader method derivation

branch body simply returns unit, so the type of f remains unchanged. On the contrary, the else advances the type of f to F[Read]. Because T-IFV defines that the two branches final environment must be consistent, T-SUBENV allows that the type in Γ_2 subsumes the one in Γ_1, thus reconciling the final types of both branches.

3.5 Additional Details

In the previous section, we have presented a type system that can be unfolded in two: the first one containing the syntax-directed rules, which can be implemented directly, no additional information being required, in the prototype compiler type-checker, and the second one containing the rules which are not syntax-directed, namely the rules for the control flow expressions, and those for the injection and subsumption. The latter type system requires that some information is kept as the algorithm, described in the next chapter, proceeds with the evaluation. However, to prove the main results of our type system, some additional technical rules are required for the syntactic constructions described in Figure 3.2, namely rules for the insync expression, for object record R, for heaps h and for states S.

Based on the full formal treatment provided for the language of modular session types [15], the main results we are expecting to prove are:

Type Preservation Well-typedness remains invariant under evaluation.

No Stuck States A MOOL program never goes into a state that is not defined by the rules.

Conformance When executing a typed program, the call trace of every object is one of the traces of the initial usage type of its class [15].

The properties *type preservation* and *no stuck states* are the usual properties for proving type-safety in object-oriented languages. The *conformance* property, however, is specific of our type system as it determines conformance of sequences of method calls to usage types. In MOOL, call traces on an object o is a sequence of method names $m_1 m_2 \ldots$, where each m_i is a method name, excluding self-calls (for they do alter an object type). A usage type defines a set of call traces, which corresponds to the set of paths in a *labelled directed graph* (see an example in Figure 4.3).

We do not present the technical rules in this thesis, nor do we prove the main results, leaving this analysis for future work. Nevertheless, we are confidant that with minor adjustments we will be able to provide a full formal treatment for the MOOL language, using subject reduction as a proof technique by induction, and having as reference the strategy adopted by the work on modular session types.

```
class F {
  usage lin open; Read
    where Read = lin{eof; ⟨Close + read; Read⟩ +
                     Close}
          Close = close; end;
    ...
}

class C {
  ...
  F f;
  ...
  unit m() {
    if(f.eof())
      unit;
    else
      f.read();
  }
  ...
}
```

$$(*)\ \frac{}{\Gamma_2 \triangleright \textbf{unit} : \textbf{unit} \triangleleft \Gamma_2}\ \text{(T-UNIT)}$$

$$(**)\ \frac{\dfrac{}{\Gamma_4 \triangleright \text{f.read()} : \textbf{unit} \triangleleft \Gamma_1}\ \text{(T-CALL)}}{\Gamma_4 \triangleright \text{f.read()} : \textbf{unit} \triangleleft \Gamma_2}\ \text{(T-SUBENV)}$$

$$\frac{\dfrac{}{\Gamma_1 \triangleright \text{f.eof()} : \textbf{boolean} \triangleleft \Gamma_3}\ \text{(T-CALL)} \qquad (*) \qquad (**)}{\Gamma_1 \triangleright \textbf{if}\ (\text{f.eof()})\ \textbf{unit};\ \textbf{else}\ \text{f.read()} : \textbf{unit} \triangleleft \Gamma_2}\ \text{(T-IFV)}$$

- $\Gamma_1 = \textbf{this}.\text{f} : \text{File[Read]}$
- $\Gamma_2 = \textbf{this}.\text{f} : \text{File[Close]}$
- $\Gamma_3 = \textbf{this}.\text{f} : \text{File}[\langle \text{Close} + \text{read; Read}\rangle + \text{Close}]$
- $\Gamma_4 = \textbf{this}.\text{f} : \text{File[read; Read]}$

Figure 3.14: Example of a derivation with subtyping

Chapter 4

Implementation

Along with the core language introduced in the previous chapter, we have developed a prototype compiler for MOOL. In this chapter, we start by giving an account of the technologies upon which our compiler is built (Section 4.1); then, we confront the language formalisation with its implementation in the prototype compiler, highlighting the main differences; and finally, we present the compiler overall architecture (Section 4.3), distinguishing between the analysis and the synthesis phases.

4.1 Technologies

The MOOL compiler takes as input a program in the form of a set of .mool source code files. The contents of those files are parsed and analysed against the specification of the MOOL grammar, after which the compiler outputs an equivalent program in an intermediate language. The MOOL compiler is implemented in Java, it uses SableCC as a parser generator, and the Mono *ilasm* Assembler tool to assemble the generated output into bytecode, which is just-in-time compiled to machine code for execution by a Common Language Runtime (CLR).

4.1.1 The SableCC framework

SableCC is an object-oriented framework, fully written in Java, that generates compilers and interpreters. We only provide here a brief overview of the framework, referring the reader to [14] for more information. The compiler compiler takes as input a .grammar file containing the grammatical specification of the source language, and generates as output a set of Java classes, which include a lexer, a parser with an Abstract Syntax Tree (AST) constructor, and an analysis framework to visit the AST nodes.

A compiler built upon SableCC gets the following features for free:

- lexical and syntactical analysis of the input program;
- automatic AST construction of the compiled program;

- tree-walker classes to perform analysis and transformations on the AST; and
- storage in the form of internal hashtables that can be used to add specific analysis information; no alteration of node types is involved, since these hashtables are kept in the tree-walker classes.

A distinguishing feature of SableCC is that there are no embedded semantic actions associated with each alternative of a grammar production in the specification file. Actions can be added by extending classes in the tree-walker generated framework. This not only increases the grammar readability, but also simplifies the development process, since the effort of implementing the analysis on the input program is reduced to the implementation of the classes that perform the semantic analysis.

SableCC takes full advantage of the Java type system to generate strongly typed ASTs. For each production of the grammar, the tool outputs a class hierarchy, consisting of an abstract class representing the production and a concrete class for each of its named alternatives. These child classes provide accessor methods for their elements. SableCC also makes extensive use of typed linked lists in elements with repetitions, using the Java Collection API. The main advantage of having a generated framework strongly typed is that the AST can be handled in a safer way, with no corruptions occurring in the tree.

SableCC enforces some additional *safeguards*. It is referred in [14] that it is not possible to create *directed acyclic graphs* (DAG). If an AST node is attached to a new parent, the link between the node and its old parent is automatically deleted, since, in the SableCC generated tree, each node can only have one parent.

4.1.2 The Mono IL assembler

Mono is an open source development platform based on the Microsoft .NET framework, that allows the development of cross-platform applications. Mono's implementation is based on the ECMA standards for C# and the Common Language Infrastructure (CLI). Mono contains the core development libraries, as well as the development and deployment tools, offered within the .NET environment.

Our compiler does not compile to native code, but to the Common Intermediate Language (CIL), or simply Intermediate Language (IL), which is an object-oriented assembly language, entirely stack-based, generated by all .NET languages. Because Mono accepts any language that compiles to IL, our prototype compiler outputs `.il` files, containing a textual representation of .NET assemblies. An assembly in the .NET world is a deployment unit, containing all the information required by the runtime to execute the application. The Mono assembler tool, *ilasm*[1], assembles IL instructions into bytecode, creating

[1]There is a corresponding *monodis* tool, which is Mono disassembler of bytecode. For more information about Mono platform, refer to Mono web page (`http://mono-project.com`).

assemblies, which are stored in Portable Executable (PE) files with a .dll or .exe extension, the difference being that a .exe assembly contains an entry point for an application (a Main method). Assemblies thus generated can be executed by any Mono machine, the execution of the program being performed by a virtual CLR.

4.2 Formalisation vs. Implementation

There are some differences between the formalisation of the core language, introduced in the previous chapter, and its implementation in the prototype compiler. The latter does not contradict any of the assumptions of the formalisation, but instead expands the core language, removing some of the syntactic restrictions that were introduced in order to simplify the typing rules.

Below is a description of the differences visible to the programmer:

- The reference to the current object this is optional; when omitted, the compiler automatically inserts it.
- The MOOL language grammar was extended in order to include local variables, as opposed to only fields and parameters in the core language (the examples in Chapter 2 make use of these constructs). In terms of the reduction and typing rules, the implications are not significant as variables share the same behaviour as fields, except for the scope that is limited to the method.
- There are two additional usage type configurations which the programmer can write: (1) the anonymous recursive usage type, mentioned in the examples, and (2) the **end** state that abbreviates an unrestricted branch with no available methods.
- A default usage type, configuring a recursive anonymous usage containing all the methods defined in the class, is automatically inserted by the compiler when no explicit specification is defined by the programmer.

4.3 Architecture

By building the protoype compiler upon the SableCC framework, the effort of implementing the analysis of MOOL programs is reduced to the semantic component. After being executed on the grammatical specification of MOOL, SableCC automatically generates four packages: analysis, lexer, node and parser. The MOOL compiler relies on SableCC classes implemented in the lexer and parser packages to parse the input files and generate the program AST, using the appropriate classes in the node package. The tree-walker classes in the analysis package are extended in order to perform the specific analysis on the program tree representation.

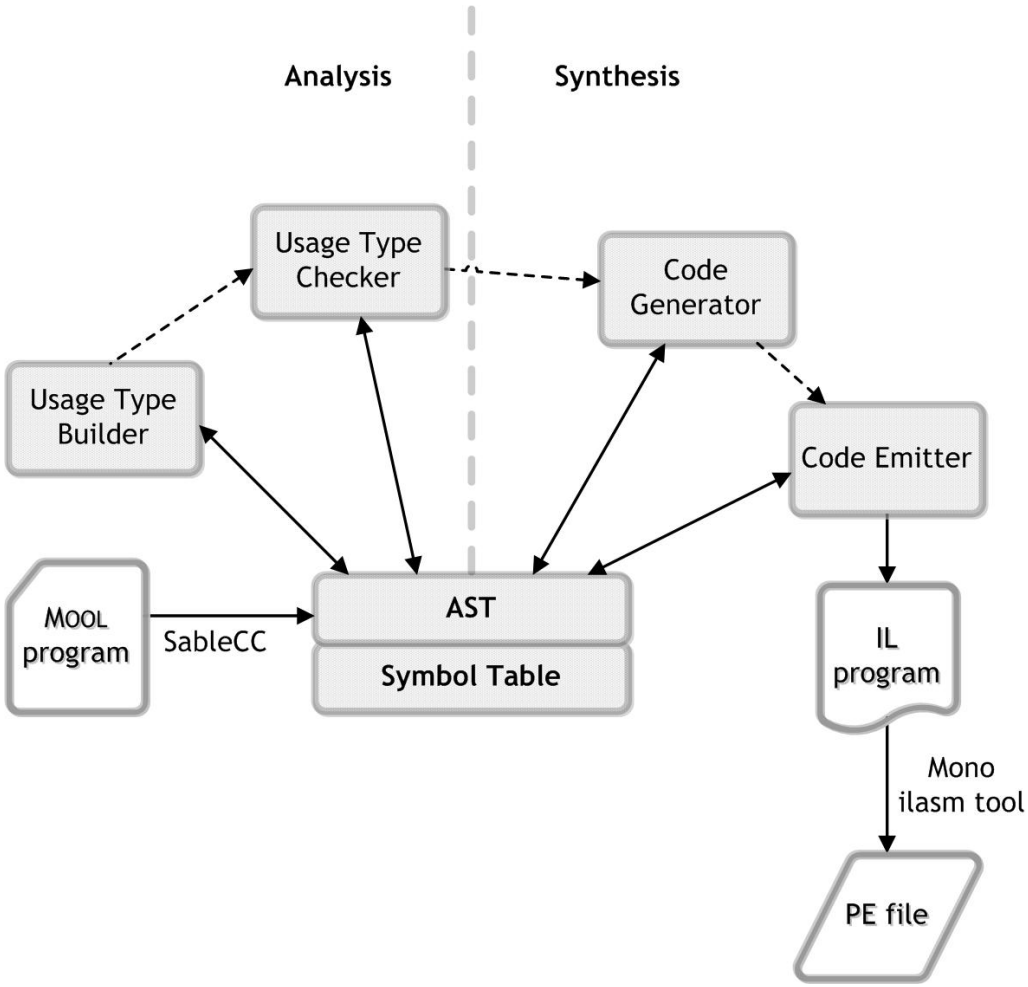


Figure 4.1: The MOOL compiler architecture

The MOOL prototype compiler is organised in the logical components represented in Figure 4.1. The view depicts on the centre the AST and the symbol table structure as the two shared internal representations on which the components from the analysis-synthesis model operate. The dotted connections between the modules denote control flow; the stronger connections between the modules and the AST/symbol table denote data access and update. The input of the compiler is a MOOL program, which, after the analysis process, is converted into an equivalent IL program. The Mono IL Assembler then assembles this into a Portable Executable (PE) file.

Each component in the analysis/synthesis phases corresponds to one or more passes over the tree, visiting the AST relevant nodes. In what follows, we give an overview of each of the modules that compose the MOOL compiler.

4.3.1 Analysis Phase

If every file of a MOOL program conforms to the MOOL grammar, no parsing errors being detected, and the corresponding AST is created, this means that the program is

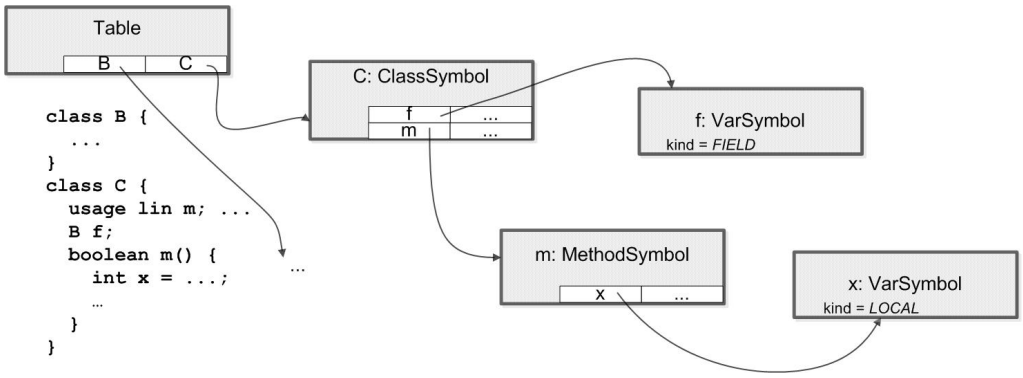
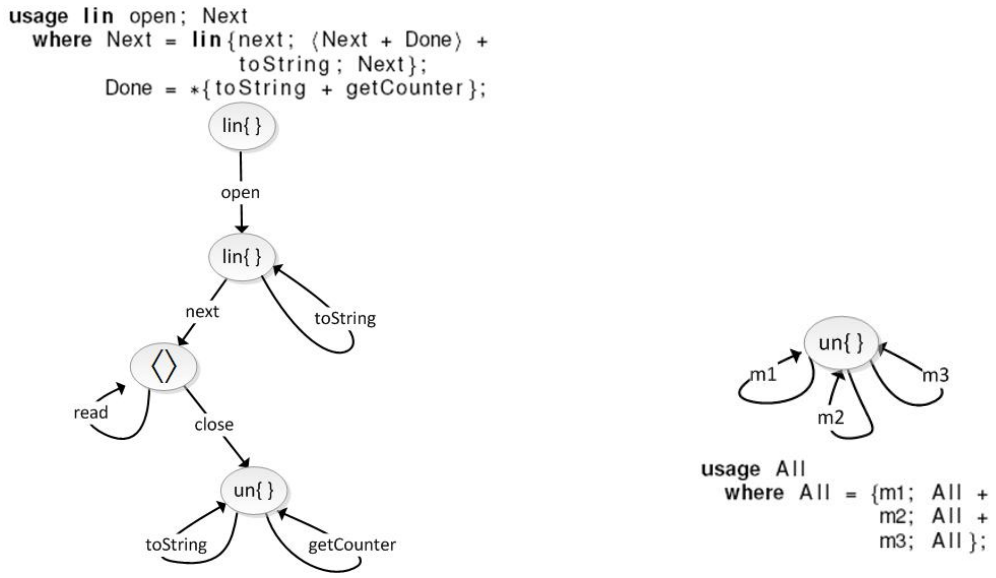


Figure 4.2: A program and its symbol table

syntactically correct. As in a standard compiler, identifiers and their uses must be checked for consistency in order to build the symbol table. This corresponds to the first pass of this phase. The second pass builds an internal representation of a class usage type, that is attached to the class definition already in the symbol table. The order of the passes is not irrelevant as only methods in the symbol table should be considered in the internal usage type representation. Finally, the third and last pass of this phase type-checks each method defined in the class based on its usage type.

Symbol table The semantic part that concerns identifier analysis usually collects information from the input program and stores it in a data structure called a symbol table. This structure maps each identifier to its semantics, and is used in all subsequent passes of the compiler. In MOOL the different kinds of symbols are: class, field, method, formal, and local variable. The symbol table must also provide information about the scope of a symbol (where in the program the identifier is visible). The MOOL language does not have inheritance nor nested methods, so the lexical scope rules are easy, having a maximal depth of two. Each class in the MOOL language defines a scope that covers every method in the class. Each method also defines a scope where every variable (local and parameter) binding is active.

The symbol table design uses the Java inheritance mechanism. It is composed of an abstract Symbol class, and a concrete subclass for each kind of symbol: class, method (which are symbols with scope) and variable in which a field indicates the kind – field, local or parameter. Scopes are implemented by setting up a separate table in the corresponding symbol class, creating a multilevel structure. At the top-level, a table provides the interface for the compiler, with an entry for each class. Under this level, each class has its own tables, one with entries for fields and another one with entries for methods within the class, and finally each method has also a table, with entries for variables (locals and parameters) within the method. Hashtables are used to implement these tables. Figure 4.2 shows a program and its symbol table (other attributes of Symbol are omitted).



(a) The FileReader usage type

(b) A default usage type

Figure 4.3: The external and internal representation of two usage types

This structure directly mirrors the language lexical scopes, defining a separate logical table and associating with its scope. The symbol table is built by one pass over the AST that visits node declarations and binds identifiers to their semantic properties in the symbol table. In subsequent passes, identifiers are looked up in the table. For example, assuming a variable is encountered in one of these visits, because the closest scope is seen first, the compiler first looks it up in the method table of variables, and, if the identifier is not found, it then looks it up in the class table of fields.

Usage type builder The four usage type configurations defined in the MOOL core language (see Figure 3.1), and the two extra configurations mentioned in Section 4.2 are transformed in an internal representation consisting of a labelled directed graph with only two type of nodes, branches and variants. The six initial constructs available to the programmer are thus internally reduced to only two in order to easily implement the equivalence of regular infinite trees.

The usage type internal representation is built using a separate hierarchy of three types to represent a directed graph: a usage type abstract class and the two subclasses for branch and variant types. A pass over the SableCC AST visits the six usage type configurations, transforming them in this data structure, which is then attached to the respective class in the symbol table. Figure 4.3 depicts the external and internal representations of two usage types: (a) represents the FileReader usage type introduced in Section 1.1; and (b) represents a default usage type, automatically generated by the compiler. The textual type that accompanies graph (b) is equivalent to writing the anonymous recursive type

usage $*\{$m1 + m2 + m3$\}$;

Custom visitors were also implemented in order to visit the nodes of the graph representation and perform specific analysis. A visitor class was designed to compare usage types for equality by implementing the union-find algorithm adapted from [1]. The union operation adds an element to an equivalence class, while the find operation checks whether an element belongs to the same equivalence class of another element. To prevent infinite cycles on recursive types, a set data structure is used as a control mechanism that tracks visited usage types.

Usage type-checker The purpose of this module is to perform the semantic analysis on types. A MOOL program type-checking starts with T-PROGRAM, which calls T-CLASS for every class in the program, that in turn triggers the class usage type walker. The pass over the class usage type then triggers the pass over the expression visitors.

The type-checker implements a type-checking algorithm adapted from the one defined in modular session types [15]. A top-level algorithm implements the usage type rules, calling an algorithm that implements the expression rules for the type-checking of each method body. A map of field and variable types is used, representing the environment in the typing rules. Additionally, a flag indicating whether or not the rule is in a variant mode is also needed. Initially, the variant mode flag is set to false.

At the end of the pass over the expressions that define each method body, T-BRANCH checks that variables have been consumed to the end, removing them from the map. Similarly, at the end of the pass over each class usage type, T-CLASS checks that only fields with unrestricted types are in the final map configuration.

Below is a description of each algorithm that composes the overall type-checking algorithm:

Algorithm for checking subtyping Function $\mathsf{sup}(u, u')$ and function $\mathsf{sup}(C[u], C[u'])$, its extension to types, with the same C in both types, are used by the algorithm that checks subtyping. The function is defined by taking the intersection of sets of methods and the upper bound of their continuations [15]. An example of subtyping has been given in Figure 3.14.

Algorithm for checking expressions The tree-walker that visits expression nodes uses a stack as a structure for supporting type-checking expressions, where the type of each expression is pushed after visiting its node. The use of a stack of types greatly simplifies this task. Since the walker implements a depth-first traversal of the tree, when a new expression is visited, the types needed have already been pushed into the stack by previous expression visitors. All that the type-checker needs to do is pop the types out of the stack, check the expression against those types, and then push the type of the expression being evaluated into the stack.

For the syntax-directed rules (see Figures 3.10-3.11 and the rule for calls in Figure 3.12), the algorithm follows their definitions. The remaining rules (see remaining rules in Figure 3.12) are not syntax-directed, which means that the algorithm for checking subtyping and an additional constructor have to be invoked. This new constructor is the compositional constructor of variant maps $\Gamma_1 \uplus \Gamma_2$, which implements the injection rules, combining two maps in a pair of maps $\langle \Gamma_1 + \Gamma_2 \rangle$ and checking for consistency. Typically, this constructor is used by T-IFV and T-WHILEV. An example has been given in Figure 3.13.

The decision as to set the variant mode flag to true is taken after evaluating the condition: if the receiver has a variant type, the flag is set to true. This is the point where the distinction between the two variant control flow expressions, T-IFV and T-WHILEV, and the two standard ones, T-IF and T-WHILE, is made. The flag is again set to false by T-VARIANT in the top-level algorithm, when checking the true and false variants.

Algorithm for checking class usage types Rules T-BRANCH and T-VARIANT are syntax-directed, which means that the algorithm simply follows their definitions as given in Figure 3.8. This is the top-level algorithm that triggers the above ones.

4.3.2 Synthesis Phase

This phase generates IL code for the CLR. The compiler converts almost directly from the AST to this high-level assembly language, using the information stored in the symbol table over the previous passes. Even though as of writing the compiler does not perform optimisations, namely it does not implement a peephole code optimiser (the synthesis part of the compiler is not the focus of this thesis), it should not be difficult to add one, since part of the effort went to construct an intermediate representation for instructions that we briefly describe below.

Code generator IL is a stack-based object-oriented language, which means that every instruction takes something from the top of the stack and puts something onto the stack. IL comprises two sets of instructions: base instructions, and object model instructions. The base instructions are analogous to native CPU instructions. Examples include call, add, ldloc (from the load and store groups of instruction), etc. The object model instructions provide an instruction set commonly used in high-level languages, namely ldobj, initiobj and ldstr. The .NET assembly reference library (mscorlib.dll), which is loaded by the CLR, is used in IL instructions. The mscorlib provides the basic class implementations for .NET Framework, such as Object, String and Thread. The MOOL compiler uses only a small subset of the available base and object instructions. IL is also strongly typed.

The mapping between the MOOL standard types and the IL types is almost direct. For a detailed description of IL, refer to [20] and [26].

A pass over the AST builds a sequence of IL instructions. For the instruction representation a hierarchy is built, having Instr as an abstract class, and every required IL instruction defined as a subclass. We implement this hierarchy in Java in order to easily translate each instruction in the MOOL language to its equivalent in IL to be run on the CLR. The pass over the AST is straightforward, except in the presence of *branching instructions*. To handle these expressions, we implement the *backpatching* one-pass technique, also described in [1]. When the branching instructions are generated, the targets of the jumps are left unspecified. These instructions are stored in a specific list for jumps. When the proper label is determined, the jumps are completed with the proper label.

Code emitter Code emission recovers the instruction sequences in the intermediate representation built by the previous pass, and emits IL code. Headers for program, class, field and method declarations also need to be generated by this component, and possibly the maxStack directive, which gives the maximal evaluation stack depth in slots (not in bytes). The directive indicates the maximum number of variables that may be pushed onto the stack at any given time during the execution of a method. The default value is 8 slots, and it should be enough for the vast majority of methods. The compiler only has to explicitly define a value for the directive if it determines a bigger stack slot. For each method defined in a class, a previous pass visits expression nodes counting the resources needed, by considering expressions that push variables onto the stack and those which pop them, and then attaches that information to the method table in the symbol table, so that it can be easily fetched by the code emitter component.

Chapter 5

Related Work

This chapter introduces a more extensive discussion of related work. It reviews the main lines of research which have directly inspired this thesis. The topics addressed are session types (Section 5.1), typestates (Section 5.2), and unique ownership of objects (Section 5.3). Each section concludes with a discussion, commenting on the similarities and differences of the described approaches regarding our work.

5.1 Session types

Introduced in [23, 24, 33], session types are, in their simplest form, structured sequences of types representing the continuous interaction in the communication between two parties. Originally developed for dyadic sessions, the concept was extended to multi-party sessions. Typically, all communication takes place within the context of sessions in which channels are made available for exchanging values between the participants according to the specified types. Channels as conceived by session types are special entities that carry messages of different types, bi-directionally, in a specific sequence between two end points.

To capture the flavour of session type channel specifications as typically found in the literature, we present below a brief example. A simple session type describing the communication between two processes A and B could have the following configuration as seen from process A perspective:

$$\oplus\{\mathsf{req} : !\,[\mathsf{Int}]\,.\,?\,[\mathsf{Bool}]\,,\mathsf{quit} : \mathsf{end}\}$$

Input and output of a value are represented by (!) and (?), respectively, while ($\oplus$) indicates the choices available and ($\&$) denotes the alternatives provided. In this example, process A has the choice of sending an integer and receiving a boolean in response, or quitting. The keyword end marks the end of the communication.

The same protocol from process B perspective would be:

$$\&\{\mathsf{req} : ?[\mathsf{Int}] \,.\, ![\mathsf{Bool}] \,,\, \mathsf{quit} : \mathsf{end}\}$$

Notice that a condition of this example is that the two processes communicate over a channel that is shared by both, and that the types described are *dual*. Duality means that the types are compatible: an input of process A has a corresponding output in process B. Another condition that the type system may enforce is that the communication between the two processes proceeds without interference. For this to happen, channels must be linear, guaranteeing that at any given moment each end of the channel may be owned by exactly one process. Communication channels can also be unrestricted (or shared); the reconstruction of session types in [37] provides insight into how to deal with a language combining linear and unrestricted channels.

Communication protocols can be expressed as types using session type theory, independently of the programming language in which session types are embedded. Proving this, programming languages that implement session types come in all flavours: pi calculus, an idealised concurrent programming language in the context of which the original concepts were developed, functional languages [17, 29, 35, 39], CORBA [34], object-oriented languages [8, 9, 15, 28, 38].

Session types for object-oriented programming Session types have gained much attention in the area of object-oriented languages. The work by Dezani-Ciancaglini *et al.* on Moose [8, 9], a multithreaded object-oriented calculus with session types, was the first attempt to marriage the object-oriented paradigm and session types. In this approach, channel-based communication is integrated in a Java-like syntax, extended with *session expressions*. For example, connect u s{e} starts a session on channel u, and the expression u.receive receives a value on that same channel. Expressions also exist for *conditional* and *iterative communication*: u. receiveIf {e}{e'} and u.receiveWhile{e} are the expressions defined for receiving a value through these control flow expressions.

This and all subsequent work has been developed around the idea of channel creation and specification within a single method, allowing delegation to another method, but on the condition that the channel must be consumed to the end.

A recent line of research combines session types and the idea taken from typestate theory that the availability of methods depends on object states. First presented by Gay *et al.* [38], describing how the interface of an object evolves based on the state of the object, subsequent work focused on modularising a session over different methods of a class [15]. The idea presented is that channel primitive operations can be hidden in an API from where clients can call methods transparently. These approaches distinguish themselves from previous works on session types by defining a global specification of method

availability at the class level, no longer constraining channel creation and specification within a single method. Another key feature is that the state of an object may depend on the result of a method with an enumerated return type. Inheritance is also included in the Java-like style; a subtyping relation on session types defines method availability in a subclass in relation to its superclass.

Discussion Our work is closer to the ideas introduced in the two recent approaches to session types [15, 38]. We also define object types based on method availability, and make object states depend on method results. However, communication channels specified by session types are typically implemented in a socket-like style, even though they can be hidden in an API [15]. Our language aims at a simpler programming style, analogous to the one in [38]. Because working with channels is less convenient than using a communication model exclusively based on method calls, we borrow from session types the idea of typed communication protocols but we apply it to usage (instead of session) protocols.

Another difference between our work and these recent approaches [15, 38] is that we use binary-only variant types, making choices depend on the boolean values, and not on the values of an arbitrary enumerated type. To simplify our language, we also impose some restrictions on the use of variant types (which must be tested on the condition of control flow expressions, and cannot be assigned), while the language in [15] provides a more flexible solution, defining a special type which keeps track of a location with a variant type.

Another important distinction is that we define a single category for linear and unrestricted (or shared) objects, as opposed to two separate ones, letting an object evolve from linear to unrestricted.

5.2 Typestates

In the literature, several lines of research can be found that reveal similarities with session type theory. One of these lines, started by Strom and Yemini, introduces the concept of typestate [32] in which the state of the object in some particular context determines the set of available operations in that context. Objects, by nature, can be in different states throughout their life cycle. The concept involves static analysis of programs at compile-time so that all the possible states of an object and associated legal operations can be tracked at each point in the program text, based on pre- and post-conditions. The authors describe an algorithm for typestate tracking in a program graph.

The concept was initially proposed for imperative languages, introduced in the language NIL, and then incorporated into several programming languages [6, 7, 10], and some ideas relate very closely to session type recent approach on modularity.

DeLine and Fähndrich propose the theory to objects in a tool called Fugue that spec-

ifies and checks typestates on Microsoft .NET-based programs. Fugue allows subclasses to define additional states. Bierhoff and Aldrich [4] extend Fugue's approach with the concepts of state refinement, which ensures subtype substitutability, and specification inheritance. The work is further extended with *access permissions* that combine typestate and object aliasing information and can guarantee the absence of protocol violations at runtime [5]. These access permissions allow developers to specify different patterns of aliasing determining how a reference might be used.

Discussion Some ideas explored by typestates relate closely to the more recent work on session types, and to our approach, namely the fact that methods should only be available on certain states, and that method availability might depend on the result of a preceding method. However, there are two main structural differences between the approaches described and our own:

(i) typestates use annotations at the method level, while we capture the entire behaviour in a class level specification; and

(ii) typestates allow a more complete aliasing control than ours (recall that our main concern are linear types), at the expense of a more complex system and set of annotations, namely by using *access permissions* [5] which capture several patterns of aliasing through a combination of allowed read and write accesses.

5.3 Unique Ownership of Objects

Baker [3] was one of the first authors to discuss the idea of linear objects and "use-once" variables in the context of concurrent computation models. Several advantages are pointed out to the introduction of controlled ownership of objects in concurrent programming, namely the possibility of using linear objects, instead of standard synchronization mechanisms, with less costs. However, there are limitations as to their use, as identified by other authors, in particular because linear objects cannot be stored in unrestricted objects, and an unrestricted object cannot evolve to a linear one.

There have been several works that attempt to introduce flexibility in linearity: the works of Hogg [22] and Almeida [2] are two of the most influential. Hogg's Islands attach to objects' interfaces aliasing mode annotations, while Almeida's Balloons uses abstract interpretation. In both systems, through different mechanisms, aliasing of normal objects is unrestricted within Islands and Balloons, but statically prevented by external references.

Discussion Although linearity presents many advantages, it also introduces limitations, namely when used within the object-oriented paradigm. In MOOL, these restrictions are also present, and proof of this is the need to introduce a standard synchronized mechanism

for accessing shared objects in mutual exclusion. The approaches referred above are more complex attempts than ours to address aliasing in object oriented languages. Our focus in this thesis are linear objects, but in future work we intend to investigate more sophisticated techniques to integrate linear and shared objects within the object-oriented paradigm.

Chapter 6

Conclusion

This thesis formalises a mini concurrent object-oriented programming language with support for the specification and static checking of usage protocols. Our ideas have been implemented in a prototype compiler, which we also describe. From the beginning, we have attempted to strike a balance between simplicity, expressiveness and safety in the design of our language usage specifications.

6.1 Achievements

Although our approach is a combination of existing approaches, we believe that we have achieved a distinct and effective framework of programming that arises from the treatment which we provide to protocol control within the object-oriented paradigm. The safety features that we borrow from (modular) session types are integrated in a simple programming language with an intuitive specification descriptor which formalises object usage. The file reader and the auction system examples, introduced in previous chapters, illustrate our key technical ideas.

The language We have defined a clean class-based language that uses primitives consistent with most object-oriented languages. The distinguishing feature is the usage specification which is able to describe both method availability, based on an object state, and aliasing restrictions, based on an object status. As existing work on modular session types and typestates, a key feature in MOOL is the possibility of making a state depend on the result of a method call. Concurrency is also included in the language, and, within this setting, linearity can efficiently and elegantly address safety issues in many usage protocols. However, some scenarios do not naturally integrate linear objects, namely when objects must be shared and provide operations accessed without thread interference. For those, a standard synchronisation mechanism has been included in the language.

Operational semantics and type system We have designed a simple operational semantics and a static type system which checks client code conformance to method call sequences and behaviourally constrained aliasing, as expressed by programmers in class usage types. Environments are used to track consumption of linear references. Our rules are attractive for implementation, since for the most part they are syntax-directed, and can be implemented in a type-checker with little effort.

The compiler We have implemented our ideas in a prototype compiler written in Java, built upon SableCC framework, and targeting the Mono virtual machine. By creating a working compiler, we were able to refine our type system, and to identify some of the limitations in our type system treatment. Some of these limitations were overcome in the course of our work, others should be refined in future work.

6.2 Future Work

We intend to improve some of the features of the MOOL language, and to find out ways to tackle some of its limitations. In what follows, we anticipate some of the topics which we are interested in pursuing:

- Our most immediate goal is to provide a full formal treatment of the MOOL language, proving that it has the property of subject reduction, as discussed in Section 3.5.
- Aliasing control is a topic we would like to further explore. In our work, we introduce linearity in the world of shared objects, thus relaxing a strictly linear control of objects. The use of linear objects solves important aliasing and typing issues. In the traditional linearity model, linear objects may reference other linear or unrestricted objects, and unrestricted objects may reference other unrestricted objects, but they may not reference linear ones. While emerging naturally from our type system, this restriction can be limiting in practice. Existing work [11] proposes a type system that relaxes this and other restrictions related to the division between linear and unrestricted types. It would be interesting to look at this and similar approaches and try to integrate them in the MOOL programming language.
- Modern object-oriented languages, like Java, offer sophisticated polymorphic features. In this work, usage types are introduced in a simpler language. We would also like to integrate our specifications in more complex usage patterns.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, volume 1241, pages 32–59. Springer-Verlag, 1997.

[3] Henry G. Baker. 'Use-once' variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995.

[4] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *FSE '05*, pages 217–226. ACM Press, 2005.

[5] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320. ACM Press, 2007.

[6] Kevin Bierhoff and Jonathan Aldrich. PLURAL: checking protocol compliance under aliasing. In *ICSE '08*, pages 971–972. ACM Press, 2008.

[7] Robert DeLine and Manuel Fähndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2003.

[8] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. In *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[9] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopolou. A distributed object-oriented language with session types. In *TGC*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[10] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*. ACM Press, 2006.

[11] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[12] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP*, pages 91–108. Springer-Verlag, 1999.

[13] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. AMS, 1967.

[14] Etienne Gagnon. Sablecc, an object-oriented compile framework. Master's thesis, McGill University, Montreal, 1998.

[15] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM Press, 2010.

[16] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[17] Simon J. Gay, António Ravara, and Vasco T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Comp. Sci., Univ. Glasgow, 2003.

[18] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 2009. To appear.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[20] John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2002.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Journal of the ACM*, 12(10):576–580 and 583, October 1969.

[22] John Hogg. Islands: aliasing protection in object-oriented languages. *OOPSLA, ACM SIGPLAN Notices*, 26(11):271–285, 1991.

[23] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715, pages 509–523. Springer-Verlag, 1993.

[24] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[25] Matthew Kehrt and Jonathan Aldrich. A theory of linear objects. In *FOOL '08*, 2008.

[26] Serge Liden. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[27] Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Master's thesis, New University of Lisbon, 2008.

[28] Dimitris Mostrous. Moose: a minimal object oriented language with session types. Master's thesis, University of London, 2005.

[29] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer-Verlag, 2004.

[30] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[31] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[32] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[33] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[34] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.

[35] Vasco T. Vasconcelos. Session types for linear multithreaded functional programming. In *PPDP*, pages 1–6. ACM Press, 2009.

[36] Vasco T. Vasconcelos. Session types for linear multithreaded functional programming. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 1–6. ACM Press, 2009.

[37] Vasco T. Vasconcelos. *SFM*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag, 2009.

[38] Vasco T. Vasconcelos, Simon Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Dynamic interfaces. In *FOOL*, 2009.

[39] Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoret. Comp. Sci.*, 368(1–2):64–87, 2006.

[40] Vasco T. Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. *CONCUR, Springer LNCS*, 3170:497–511, 2004.