Revisiting memory assignment semantics in imperative programming languages

Racordon, Dimitri

UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES

Département d'Informatique Professeur D. Buchs

Revisiting Memory Assignment Semantics in Imperative Programming Languages

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de
Docteur ès sciences, mention informatique

par

Dimitri Racordon
de
Torny (FR)

Thèse No 5409

Genève
Atelier d'impression ReproMail
2019

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

DOCTORAT ÈS SCIENCES, MENTION INFORMATIQUE

Thèse de Monsieur Dimitri RACORDON

intitulée :

«Revisiting Memory Assignment Semantics in Imperative Programming Languages»

La Faculté des sciences, sur le préavis de Monsieur D. BUCHS, professeur ordinaire et directeur de thèse (Département d'informatique), Monsieur J. NOBLE, professeur (School of Engineering and Computer, Victoria University of Wellington, New Zealand), Monsieur E. CASTEGREN, docteur (Department of Computer Sciences, KTH Royal Institute of Technology, Sweden), Monsieur S. HOSTETTLER, docteur (Département d'informatique, Université de Genève), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 15 octobre 2019

Thèse - 5409 -

Le Doyen

N.B. - La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

私の息子たちの翔満と耀満へ

Acknowledgements

This work would not have been possible without the unfailing support of so many people I had the chance to meet along the way.

First, I would like to express my sincere gratitude to my advisor Professor Didier Buchs. Thank you for believing in me, for pushing me to pursue my research, for channeling my impetuous desire to explore new lands and yet giving me the freedom to choose my own path. I could not have hoped for a better boss.

To my friends and colleagues from the SMV lab, to David Lawrence for sharing with me the first years of this adventure, to Maximilien Colange for giving me a taste for convoluted formal notations, to Alban Linard for our endless discussions about programming languages and Doctor Who, to Damien Morard for patiently letting me rant about the writing of the chapters herein, to Stefan Klikovitz for forgiving my bizarre affection for long and complicate sentences, thank you so much for making this journey possible.

To my friends and colleagues at Socialease, thank you for coping with a CTO who spent more time writing inference rules than planning development sprints.

To my wife Shiori, thank you for your unconditional love and support. You never failed to cheer me up when I was feeling lost. You inspired me every day with your kindness and strength and I cannot wait to live the many experiences that await us by your side.

To my family, thank you for your love and support. My mother, my father and my sister made me who I am today. You are my biggest fans, and a formidable source of inspiration.

To my friends, Aloys, Nadine, Robin, Elliott, Alexandre, Adrien, Noémi, Florianne, Jean-Pierre, Anouck, Christian, Mireille, Sam, Philippe and whoever else I forgot, thank you for being there for me every time I was feeling overwhelmed.

Abstract

Programming languages have become an unavoidable tool, not only for computer experts, but also for scientists and engineers from all horizons. For the sake of usability, modern programming languages typically sit at a very high abstraction level and hide the intricacies of data representation and memory management. The continuing growth in computational power has enabled this evolution, allowing compilers and interpreters to support features once thought unrealistically expensive, such as automatic garbage collection algorithms and powerful static type inference. While this has undeniably contributed to make code simpler to write and clearer to read, relics of the underlying model still transpire in most languages' semantics. In imperative programming languages, where computation is expressed in terms of successive mutations of a program's state, leaks at any level of abstraction may lead to unintuitive and/or misunderstood memory assignment behaviors. In particular, the interplay between values and variables can prove to be a prolific source of confusion. While both are usually perceived as interchangeable notions, values are semantic objects that live in memory while variables are syntactic tools to interact with them. As both concepts are not necessarily thetered in a one-to-one relationship, foreseeing the reach of a modifying operation requires a clear understanding of the memory abstraction.

This thesis proposes a model to better reason about memory management. Our first objective is to provide a more accurate description of memory assignment semantics, in a universal and unambiguous way. The resulting model marks a clear distinction between variables and values, that highlights situations where aliasing occurs and situations where assignments may have side effects beyond the mutation of a single variable. Such a model is presented formally by the means of a complete semantics, and informally with examples of its application to some non-trivial examples. It is then used to describe memory errors related to assignment. Two methods are proposed. The first is based on an instrumentation of the dynamic semantics to detect accesses to uninitialized or freed memory. The second relies on a capability-based type system to guarantee memory safety statically. Finally, Anzen, a general purpose language based on the aforementioned model, is introduced as an attempt to empirically validate its practicality.

Résumé

Les langages de programmation sont devenus un outil indispensable, non seulement pour les professionnels de l'informatique, mais aussi pour les scientifiques et ingénieurs d'autres disciplines. Ainsi, dans le but de faciliter leur utilisation, les langages de programmation offrent désormais de nombreuses abstractions visant à masquer les subtilités liées à la représentation des données et à la gestion de la mémoire. La continuelle croissance de la puissance de calcul a rendu cette évolution possible, permettant aux compilateurs et interprètes de supporter des fonctionnalité autrefois jugées irréalistes, telles que la récupération automatique de mémoire ou encore l'inférence de type. Si ces améliorations ont indubitablement contribué à rendre le code plus facile à écrire et plus clair à lire, des reliques du modèle sous-jacent transparaissent toujours dans la sémantique de la plupart des languages, lesquelles peuvent conduirent à des comportements mal compris. En particulier, la relation entre valeurs et variables se révèle être une prolifique source de confusion. Alors que ces deux notions sont fréquemment perçues comme interchangeables, une valeur est en fait un object sémantique vivant dans la mémoire tandis qu'une variable est un outil syntaxique pour interagir avec. De plus, ces concepts ne sont pas nécessairement liés par une relation bijective. Par conséquent, prédire la portée d'une opération de modification requiert une compréhension claire de l'abstraction faite sur la mémoire.

Cette thèse propose un modèle pour mieux raisonner à propos de la gestion de mémoire. Notre premier objectif est de fournir une description plus précise des sémantiques d'assignation, de manière universelle et sans ambiguïté. Le modèle qui en résulte marque une distinction nette entre variables et valeurs, et met en avant les situations dans lesquelles sont créés des alias, ainsi que les situations dans lesquelles une modification peut avoir des effets de bords. Ce modèle est présenté formellement par le biais d'une sémantique complète, et informellement par le biais d'exemples de son application sur des exemples non triviaux. Il est ensuite utilisé pour décrire les erreurs mémoires liées à l'assignation. Deux méthodes sont proposées. La première consiste en une instrumentation de la sémantique dynamique pour détecter les accès à de la mémoire non initialisée ou libérée. La deuxième se repose sur un système de type basé sur des capacités pour garantir

des propriétés de sureté statiquement. Finalement, nous présentons Anzen, un langage de programmation général basé sur le modèle formel susmentionné, dont le but est de valider de manière empirique son utilisabilité.

Contents

Foreword **iii**

1 Introduction **1**
1.1 Memory in the Imperative Paradigm 2
1.2 Motivations 8
1.3 Contributions 10
1.4 Outline 12

2 Background and Related Work **15**
2.1 Memory Management 15
2.2 Formal Methods 23
2.3 Static Program Analysis 27
2.4 Synthesis 36

3 Mathematical Preliminaries **39**
3.1 General Notations 39
3.2 Data Types 40
3.3 Inference Rules 45

4 The $\mathcal{A}$-Calculus **49**
4.1 Problematic 50
4.2 Informal Introduction 54
4.3 Syntax and Semantics 57
4.4 Examples 79
4.5 Summary 83

5 Dynamic Safety **85**
5.1 Observing Memory Errors 86
5.2 Signaling Memory Errors 92
5.3 Summary 107

6 Static Safety **109**
6.1 Rust's Type System in a Nutshell . . . 110
6.2 Static Garbage Collection . . . 115
6.3 Type Checking . . . 126
6.4 Records and Static Garbage Collection . . . 151
6.5 Capabilities for Immutability . . . 154
6.6 Summary . . . 160

7 Anzen **161**
7.1 Compilers and Interpreters . . . 162
7.2 Anzen in a Nutshell . . . 164
7.3 Examples . . . 170
7.4 Anzen's Intermediate Representation . . . 172
7.5 Anzen's Compiler . . . 178
7.6 Summary . . . 185

8 Conclusion **187**
8.1 Summary of our Contributions . . . 187
8.2 Critique . . . 188
8.3 Future Directions . . . 189
8.4 The Holy War of Programming Languages . . . 191

Bibliography **193**

A Formal Companion **209**
A.1 List of Symbols and Notations . . . 209
A.2 $\mathcal{A}$-Calculus' Semantics . . . 213
A.3 $\mathcal{A}$-Calculus' Type System . . . 217

B Anzen **223**
B.1 Anzen's Concrete Syntax . . . 223
B.2 AIR's Concrete Syntax . . . 226

C Full Proof of Property 6.3.2 **229**

Chapter 1

Introduction

The last half century has seen the unbridled escalation of software complexity. It has been matched with decades of work in industry and academia, which materialized, in outstanding disciplines such as software engineering [126], software testing [8] or software verification [61]. While these methods and tools have proved to be invaluable in assisting developers write correct code, programming languages play an equally if not even more critical role. Good language design is paramount to the production and maintenance of software, as a language is *the* first weapon in a developer's arsenal. Due to the ever increasing computational power at our disposal, modern compilers have allowed us to move from cryptic sequences of machine instructions to elaborate programs. As a result, developers can now better convey their intent, at an abstraction level that is closer to the spoken language than ever before. However, this evolution can also be seen as double-edged sword. Great expressiveness brings the potential for more inadvertent behaviors, as powerful notations are generally paired with heavy cognitive loads [86]. The difficulty lies in properly mapping abstract concepts, such as object-orientation and higher-order functions, onto the rather simple low-level systems which still dominate the vast majority of contemporary computer architectures.

A good representative of this predicament is the notion of *aliasing*. A typical computer makes use of *memory* to save and retrieve information. It is organized as a collection of individual storage units, each given a unique address. Aliasing occurs when a program uses different names, a.k.a. *references*, to designate the same address. This allows for self-referential abstract data structures such as graphs to be represented in memory. Some form of aliasing is indispensable in any realistic programming language. In fact, aliasing is so ubiquitous that the use of references has become the de facto standard in imperative programming languages, commonly used to hide the specifics of memory management. Unfortunately, this comes at the cost of a remarkable complexity as it hinders one's ability to predict the effect of memory modifications. The issue is particularly pervasive in impera-

tive systems which, despite the renewed interest for the functional paradigm, still account for nine of the ten most used languages according to a 2019 StackOverlfow survey[1]. Most mainstream languages lack the ability to prevent or detect erroneous memory situations. For instance, although publicly released in 2014, Swift is prone to memory leaks, while unintended sharing plagues nearly all imperative programming languages. Consequently, this moves the burden of guaranteeing the correctness of each and every assignment onto the developer. The problem is exacerbated by the poor control modern imperative languages usually support, only offering to modify objects by reassigning their fields to new aliases. As a result, reasoning about the effects of mutation and assignment can turn into an inexhaustible source of frustration. Despite this predicament, aliasing remains a corner stone of most abstraction strategies. Hence, appropriate techniques to describe and control its effects are essential.

This thesis proposes two constructs to better reason about memory and aliasing. The first is a model to unambiguously describe assignment semantics as found in imperative languages. The second is a capability-based type system to prevent access to uninitialized memory and dangling references, while guaranteeing reference uniqueness [28] and object immutability [116]. Both are introduced in the context of a formal calculus, for which an implementation is also presented in the form of Anzen[2], a general purpose programming language.

1.1 Memory in the Imperative Paradigm

Imperative programming is a paradigm in which a program is defined as a sequence of instructions. In other words, it focuses on describing *how* a program operates at a very operational level, similarly to a cooking recipe. The evolution of the program usually depends on a state, whose changes are expressed in the form of commands that can modify it. On a computer, this state is typically represented by memory and manipulated through value *assignments* that write data to specific memory locations. Programming languages use *variables* to refer to these locations, and instructions of the form "$l := v$" to modify the memory. Here, l is a variable pointing to a specific memory location, and v is a value (e.g. the number 42).

The imperative paradigm is very close to the way a typical computer actually operates, which explains its historical success. However, to cater for the complexity involved in modern software applications, language designers have been compelled to provide techniques that map more abstract concepts onto memory

[1] `https://insights.stackoverflow.com/survey/2019`
[2] `https://www.anzen-lang.org`

assignments. Object-orientation and the more recent renewed interest in higher-order functional programming are a testament of this quest. Both approaches rely on elaborate interactions with the memory to hide the intricacies of data representations. At their core resides the notion of *pointers*, which allow a finite one-dimensional array of memory locations to be turned into a graph. Aliasing occurs when two pointers refer to the same memory location, that is when two edges of the graph point to the same node. Enabling such situations allows to save space and time, by avoiding unnecessary data duplication, and offers an excellent solution to represent self-referential data structures. In fact, representing such structures without aliasing is challenging and inefficient [96].

However, aliases are one of the most prolific sources of errors in software development, in particular when used in an imperative setting. The problem stems from the difficulty to foresee the impact of a single instruction. Aliasing breaks the symmetry between the lexical shape of a program and its execution, as operations on one variable can adversely impact other seemingly unrelated ones. As an example, consider the following program:

$$\texttt{let}\ x, y\ \texttt{in}\ x \coloneqq 0\ ;\ y \coloneqq x\ ;\ y \coloneqq 2\ ;\ x \coloneqq 4\ ;\ y$$

Assuming $y \coloneqq x$ sets y to be an alias on the value of x, this program eventually evaluates to 4[3]. But this cannot be determined by only looking at the subsequent assignments on y. Instead, one has to carefully identify to which memory location y refers; an information that does not appear explicitly, and track any write access to that location in order to understand where its value was overridden. In other words, one has to construct a mental model of the underlying memory, and integrate this inherently dynamic construct into her reasoning while deciphering a static sequence of instructions. An illustration is given in Figure 1.1.1, representing the evolution of the memory for the last two assignments. Variables are drawn with squares, while the memory to which they refer is drawn with circles.

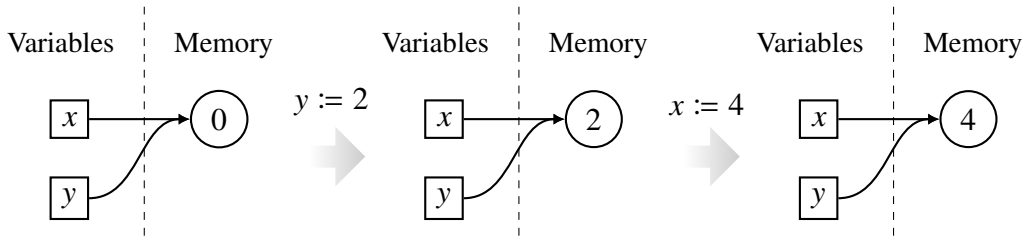


Figure 1.1.1: Effects of memory assignments.

The difficulty of foreseeing the possible side effects of an assignment obviously increases with the size of the program, as well as with the additional features

[3]The reader will notice that most mainstream languages do not adopt such assignment semantics for simple values like numbers.

the language may have to offer, such as functions and object-orientation. The remainder of this section discusses less trivial examples of software bugs related to memory management in the presence of aliases.

1.1.1 Unintended Sharing of State

Let us first have a look at the "Signing Flaw" security breach of JDK 1.1, which untrusted applets could use to gain all access rights into the virtual machine. In Java, all class instances (i.e. instances of `java.lang.Class`) hold an array of signers, which act as digitally signed identities that Java's security architecture uses to determine the class' access rights at runtime. The value of this array can be obtained by calling `java.lang.Class.getSigners`. Before the breach was discovered, the method used to simply return an alias on its array of signers. Because the returned array was an alias, an untrusted applet could freely mutate it and add trusted signers to its class, therefore fooling the security system into giving it access to restricted code signed by a trusted authority. The following listing is an excerpt of the flawed `java.lang.Class` implementation:

Java

```
public class Class {
  public Identity[] getSigners() {
    return this.signers;
  }
  private Identity[] signers;
}
```

Note that none of the standard protection mechanisms of Java (a.k.a. access control modifiers) can help solving this issue [139]. Although the property is declared private, its reference is exposed through a method, which breaks the encapsulation principle. Using Java's `final` keyword would not be of any more assistance. While it would prevent the property from being reassigned, it would not impose any restriction on the array object itself which, consequently, could still be freely mutated. In fact, most languages that use references as the primary way to manipulate memory suffer the same limitation, since they offer pointer immutability but no support for restricting object mutability. The only available solution would be to return a copy of the property.

Unfortunately, copying in the presence of aliasing can also prove misleading. In Java, copying an object only duplicates the references that constitute it, resulting in the creation of an alias for each property of the object being copied. In our above example, this means that copying `signers` would indeed create a new array, but that the elements of this array would still refer to the same objects as the

original one. Although such a strategy is usually efficient in practice, it can also lead to unintended sharing of state. Other copying approaches may exist (or even coexist) in other languages, and must be carefully taking into account.

1.1.2 Use After Free

This second aliasing issue is related to memory deallocation. *Use after free* errors denote situations where a pointer that refers to already freed memory is dereferenced, as illustrated the following C++ program:

C++

```
1  const char * mk_hello(const char * name) {
2    ostringstream oss;
3    oss << "Hello, " << name << "!";
4    return oss.str().c_str();
5  }
6
7  int main() {
8    const char * text = mk_hello("Amy");
9    cout << text << endl;
10   return 0;
11 }
```

In C++, the memory assigned to variables declared in a function is deallocated when the function exits. Those variables are called *local*. In the above listing, the variable `oss` acts as a local string buffer that is used to build a character string. It holds an object that abstracts over a C-style character string, internally represented as a pointer to the base address of a contiguous array of characters. The call to the methods `str()` followed by `c_str()` at line 4 allows one to retrieve the address of said array. However, because `oss` is a local variable, its address corresponds to memory allocated within the scope of the function `mk_hello`, which will get deallocated as soon as the function returns, at line 5. As a result, the main function now holds a pointer `text` to freed memory and the behavior at line 9 becomes undefined.

Just as data races, use after free errors can prove particularly challenging to detect and/or reproduce, as they induce undefined behaviors. They are also a source of potential security exploit [98].

1.1.3 Memory Leak

Memory leaks are directly related to memory management as well. Those occur when the system responsible to drive the program execution considers a particu-

lar memory location as allocated, whereas there are no pointers to it. While such errors are more common in programming languages that require explicit memory management, memory leaks can still occur within systems that manage memory automatically, in particular those based on reference counting [87]. In a nutshell, reference counting consists in associating a counter with each resource, whose value denotes the number of pointers thereupon. The counter is incremented every time a new alias is made, and decremented every time an alias is removed. If it reaches zero, then the resource is deallocated and the memory locations it occupied are reclaimed by the system. While simple and relatively cheap to implement, reference counting fails to deallocate memory for cyclically referred resources, as their respective counters will obviously never reach zero. The following example illustrates this situation in Swift, which uses reference counting for automatic garbage collection:

Swift

```
1  do {
2    class Person {
3      var name: String
4      var loveInterest: Person?
5    }
6
7    let jim = Person(name: "Jim", loveInterest: nil)
8    jim.loveInterest = Person(
9      name: "Sarah", loveInterest: jim)
10
11   print(jim.loveInterest.loveInterest.name)
12   // Prints "Jim"
13 }
```

The class `Person` comprises a field `loveInterest` that can hold a reference to another `Person` instance. At line 8, the love interest of Jim is set to Sarah, whose love interest is set to Jim. As a result, the reference `jim` is now an instance of `Person` that holds a reference to another instance that cycles back to `jim`. Therefore, both instances will never be deallocated, even if the reference to `jim` is no longer accessible once the scope ends at line 11. This cyclic situation is depicted in Figure 1.1.2.

Solving this problem requires to either manually break the cycle, for instance by assigning `nil` to `jim.loveInterest`, or to annotate the `loveInterest`'s field definition to prevent Swift's runtime from increasing the reference counter of the object to which the field is assigned (i.e. using the `weak` type modifier). Unfortunately both approaches require a continuous awareness of the language's garbage collection strategy, which arguably defeats the purpose of a supposedly automated

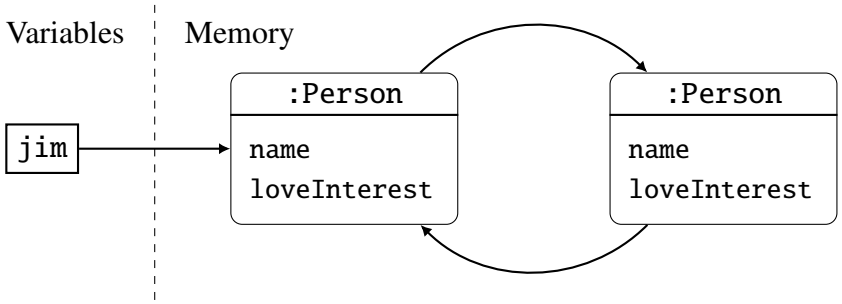


Figure 1.1.2: Example of reference cycle as an UML Object Diagram [63].

system to abstract over memory management.

1.1.4 Overview of Memory Management Issues

The examples presented above illustrate some of the most common pitfalls of memory management. We summarize them with the following observations:

- For all languages, the consequences of aliasing can be far reaching, and require a deep understanding of the semantics to be fully foreseen. Moreover, while a programming language may abstract over the notion of object representation, predicting state mutations requires some level of awareness about the runtime system's memory management strategy, as well as the optimizations it may apply.

- Access control is insufficient to prescribe alias safety, as hidden variables may be aliased by publicly available references. Besides, access control and other safety mechanisms are usually offered as an opt-ins, which makes them more good practices than enforced language constraints. This means learning programmers might not be aware of them, while more seasoned developers might ignore them and rely on their own conventions [99].

- While automatic memory management frees the developer from the responsibility to handle memory allocation and deallocation, its interplay with aliasing can introduce problematic situations that should be accounted for.

- Although the examples above feature object-oriented languages, the problems we highlighted are not necessarily related to object-orientation itself, and could be illustrated in other imperative languages. Rather, the use of encapsulation adds a layer of complexity while reasoning about aliasing.

1.2 Motivations

The starting point of our investigation is an observation on the current vast majority of contemporary programming languages. Influenced by the overwhelming popularity of object-orientation, most languages abstract over the notion of memory locations and pointers thereupon. Variables are merely containers for values, without any concern for *where* such values are actually stored. Combined with the appropriate garbage collection algorithms [142], this theoretically allows memory management to be completely disregarded, as contemporary programming languages now take care of the entire memory lifecycle.

Unfortunately, implementations generally leak details from the object representation they manipulate. In particular, programming languages often distinguish between *primitive values* and *reference values*. The former represent data that fit into a processor register, and therefore do not need any abstraction to be assigned to a variable. On the contrary, the latter denote more complex data structures that require more elaborate representations. Consequently variables assigned to primitive values do actually hold *values*, while variables assigned to reference values hold *pointers* thereupon, but abstract over the intricacies of dereferencing. Such design choices have important consequences for memory assignment. As primitive values are not represented by a pointer, they are copied upon assignment. However, assigning a reference value only copies its pointer, effectively resulting in the creation of an alias. While sound, this strategy implies that the semantics of an assignment depends on the type of its operand, an information that is typically not conveyed syntactically. Ironically, purposely manipulating aliases for a finer control is also challenging, as directly mutating the content of a variable without reassigning it to a different memory location is generally impossible. Put differently, one usually cannot modify the bits representing a particular value, and should instead reassign the references to said value to an entire different memory location. Consequently, the most straightforward solution to explicitly deal with pointers as first-class components is to wrap references into other objects, essentially reimplementing the concept of pointers at a higher abstraction level.

The recent interest into linear type systems [140], as observed in Rust or C++, only adds to the confusion by bringing yet another variant of assignment semantics. Linear type systems treat variables as linear resources [68], which cannot be "cloned", therefore enforcing uniqueness statically. Actual implementations offer to relax on this restriction at function boundaries by the means of borrowing mechanisms [104], so that unique values can be temporarily aliased. While borrowing might be paired with additional concepts such as immutability [89], its underlying semantics is in fact identical to aliasing at an operational level. Nonetheless, it is sometimes introduced and understood as a completely separate concept, thus further steepening learning curves. Despite these numerous drawbacks, support

for different assignment semantics is useful with respect to performance [128] and safety [96].

Existing formal models describing imperative languages (e.g. [111, 81, 124]) tend to assume a single assignment mechanism, often mimicking the aforementioned pointer abstraction and relying solely on reassignment to perform mutation. More ambitious approaches have proposed a formal semantics for specific languages that support various kinds of assignments, notably including Rust [141]. However, real programming languages come equipped with countless additional features whose inclusion may interfere with a reasoning focused at the effect of assignment. While legitimate criticism has been raised toward the validity of proofs established on minimal calculi [7], a dedicated formal language may prove beneficial either as a mean to model existing programs, or as an educational tool to better understand the essence of memory assignment. As a result, our first research question aims to provide such a formal model.

Research Question 1:

How can we create a formal model to express the essence of different memory assignment semantics? Can such a model improve the legibility of assignments at a syntactic level, while supporting various kinds of semantics?

As mentioned in the previous section, managing an object lifecycle automatically is not enough to address all memory related issues. The problem lies in the interplay between aliasing and memory assignment. Common safeguard mechanisms such as access control are insufficient, while comprehensive pointer analysis approaches do not scale and produce reports difficult to apprehend [104]. In response, a great body of work has been dedicated to the topic of aliasing control, resulting in a plethora of proposals to address a broad spectrum of issues (see [39] for an extended account). However, as mentioned in the "Geneva Convention" on aliasing [78], identifying the right model to control aliases is a difficult dilemma. One has to balance strong restrictions to guarantee safety with enough freedom to preserve the language's expressivity. The authors proceed to categorize four approaches:

- **detection**, consisting of the identification of potential aliasing,
- **advertisement**, consisting of code annotations aimed at helping an automated system perform detection,
- **prevention**, consisting of techniques that can guarantee the absence of aliasing statically, and
- **control**, consisting of mechanisms to tame the effects of aliasing.

While the objective of our research is not to reconsider the benefits of aliasing control, a second research question aims to investigate whether such approaches can fit into a formal model where aliasing occupies a central role. In particular,

we are interested in mechanisms that can help asserting properties on memory's treatment. Our main objectives are to manage objects' lifecycles automatically and to prevent improper memory use.

Research Question 2:

What mechanisms for aliasing control can be beneficial in the context of a formal model focusing on memory assignment semantics?

While a formal model may prove useful in reasoning about the theoretical aspects of memory assignments, it may not be suitable to assess the practicality of an actual language based on it principles. Contemporary programming languages usually offer constructs to promote code reusability, such as modules, higher-order functions and generic types. Supporting such features may have non-trivial implications on both dynamic and static semantics. Unlike formal models, implementations cannot afford to discard computational complexity and thus scalability can be another legitimate concern. Therefore our last research question relates to the design and implementation of a compiler for an actual programming language based on a formal model for assignment semantics.

Research Question 3:

How to create a general purpose language that unambiguously implements various notions of memory assignment? Would such a programming language be practical to write actual, non-trivial programs?

1.3 Contributions

This thesis is set in the context of imperative programming language design, advocating for a precise notion of assignment to work in concert with sound memory safety mechanisms. This section elaborates on our main contributions, with respect to the research questions that were introduced in Section 1.2.

1.3.1 The $\mathcal{A}$-Calculus

The main contribution of this thesis is the $\mathcal{A}$-calculus (pronounced *assignment calculus*), a formal system for reasoning about memory management that aims at unifying imperative assignment semantics. It addresses our first research question by featuring three distinct assignment semantics:

Aliasing Assignment An aliasing assignment explicitly assigns aliases. While this matches the semantics of Java for reference values, primitive values are

not treated differently so that the behavior of an aliasing assignment remains agnostic of its operands' types.

Mutation Assignment A mutation assignment modifies the content of its left operand with a *deep* copy of its right one. This effectively results in a copy assignment, with the copied object being free of any alias.

Move Assignment A move assignment treats resources linearly [68], and moves values from one reference to another.

Those assignments are defined solely in terms of variables and values, so that their semantics is not linked to a specific memory model. In other words, the $\mathcal{A}$-calculus abstracts over the concept of stack and heap (c.f. Section 2.1.1). Our proposal contains a complete formalization of the $\mathcal{A}$-calculus' operational semantics, which we use to define memory errors formally. We then show how to instrument its semantics to detect and prevent undefined behaviors linked to an imporper use of the memory (i.e. access to uninitialized memory). Lastly, we present various examples of translation from programs written in actual languages to the $\mathcal{A}$-calculus.

Beyond its formal aspect, another contribution of the $\mathcal{A}$-calculus is a precise definition of the relationship between variables and memory. Variables and the values they represent are often perceived as interchangeable concepts. However, while values are semantic objects that live in the memory, variables are syntactic tools to interact with them. Although this subtlety may not be extremely consequential in a purely functional paradigm, as variables are more tightly paired with the values to which they are bound, an unmistakable distinction between variables and values is crucial in an imperative setting to understand a program's behavior. By relying on distinct assignment operators with an unambiguous semantics, the $\mathcal{A}$-calculus places emphasis on values' journeys into memory, thus preserving the distinction.

1.3.2 A Static Type System for Aliasing Control

Our contribution toward our second research question is a static type system, which brings the following properties to the $\mathcal{A}$-calculus:

Static Garbage Collection The type system uses Ownership Types [39] to support static garbage collection. Each value is associated with a single owning reference, whose lifetime is bound by the lexical scope in which it is declared. Deallocation occurs when an owner goes out of scope, in a fashion reminiscent of region based memory management [134].

Uniqueness We treat uniqueness as a type capability, whose main purpose is to control ownership transfer. Unlike most other uniqueness implementations (e.g. [40, 73]), our type system only enforces uniqueness shallowly. In other words, properties of a "unique" object may be aliased freely.

The type system is sound with respect to memory safety. Further, capabilities inference is decidable, at the cost of annotations on function signatures. Due to the minimalistic nature of the $\mathcal{A}$-calculus, its type system can also easily be adapted to existing imperative languages [117].

1.3.3 Anzen

Our last contributions are a compiler and interpreter for the Anzen programming language. Anzen is an attempt to empirically validate the practicality of the $\mathcal{A}$-calculus. The language borrows concepts from the $\mathcal{A}$-calculus and combines them with more elaborate constructs. Its type system supports generic types and comes with a powerful inference engine. While the language features a comprehensive annotation system, most of them can be eluded. Like Rust, Anzen advocates for safe defaults so that its syntax puts emphasis on variables and values that may be subject to unintended mutations.

The compiler translates programs into an intermediate representation reminiscent to Java bytecode, but whose semantics is extremely close to the $\mathcal{A}$-calculus. As a result, the interpreter is implemented as a quasi direct translation of its formal operational semantics. The interpreter allows step-by-step program executions, and leverages Xcode's debugger [10] to support memory inspection at runtime.

1.4 Outline

This thesis is divided in eight chapters, including this introduction.

- Chapter 2 gives a detailed background on memory management and attaches a clear definition to the key concepts that are discussed in the remainder of the thesis. The chapter continues with a survey of approaches aimed at formally verifying program correctness with respect to memory management. It concludes with a synthesis of related research and positions our work with respect to the literature.

- Chapter 3 briefly introduces the mathematical concepts and notations that are used throughout the formal parts of this thesis.

- Chapter 4 presents the $\mathcal{A}$-calculus, the main contribution of this thesis. The chapter starts with a description of the problem with assignment semantics

in existing programming languages. Then, an informal description of the assignment operators' semantics precedes a complete formalization of the model. A few examples of translation from existing imperative languages to the $\mathcal{A}$-calculus are proposed.

- Chapter 5 uses the assignment-calculus to formalize common memory errors, and proposes to instrument its semantics to detect undefined behavior at runtime with a mechanism reminiscent to exceptions [30].

- Chapter 6 presents a static type system for the $\mathcal{A}$-calculus to guarantee memory safety. The system is first introduced informally by the means of examples, and then formally by the means of inference rules. A soundness proof is discussed before the chapter concludes with some remarks about the issues related to the support of other forms of aliasing control.

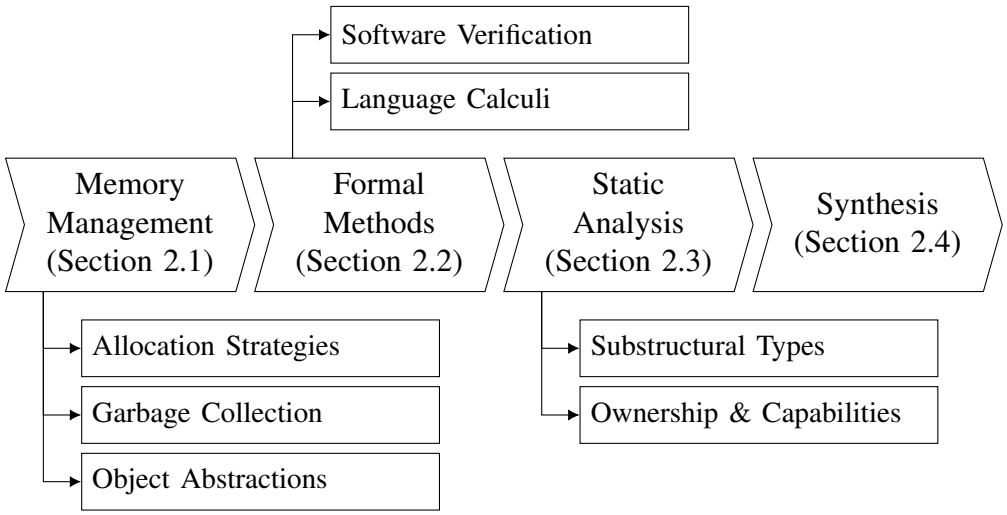
- Chapter 7 introduces the Anzen programming language, together with its compiler and interpreter. The chapter starts with some background on compiler design, before delving into Anzen's features and implementation thereof. Anzen's intermediate representation (AIR) is described, along with a brief comparison of its semantics with that of the $\mathcal{A}$-calculus.

- Chapter 8 closes this thesis with a summary and a critique of its contributions, and provides directions for future works.

Chapter 2

Background and Related Work

This chapter presents some background and related work on topics closely related to this thesis' contributions. Its organization is depicted by the following picture. Chevrons represent sections, and rectangular boxes indicate the main themes that are discussed.



2.1 Memory Management

Although a Turing machine theoretically operates on an infinite tape, actual programming languages, despite being labeled *Turing complete*, obviously can only ever handle a finite amount of memory. "Memory management" is the collective term to designate methods related to this activity. Their essential requirement is to provide a program with the memory it needs to store the data it manipulates. Computing the total amount of memory a program requires is undecidable – in

the same way as the halting problem is undecidable. Consequently, memory management techniques require some *dynamic* memory allocation mechanism that is able to allocate specific portions of a finite memory at runtime. This mechanism shall inevitably be matched with a deallocation strategy to indicate what portions are no longer in use, and might consequently be reclaimed by the system.

This section briefly covers the core concepts related to memory management. In particular, it illustrates the implications of the most common allocation strategies, with respect to memory safety.

2.1.1 Stack vs. Heap Allocation

The phrase "*memory management of a language X*" is somewhat an abuse of terminology. Memory consumption is inherently linked to the execution of a program, that is the interpretation of the language in which it is written. Hence, memory management is in fact a concept linked to the *execution model* of the language (a.k.a. its runtime system), which is responsible for allocating and freeing memory in accordance with its memory management strategy. For example, a typical C runtime system will allocate memory for all local variables of a function when it is called, maintain the address of the memory block reserved by this allocation, and deallocate it once the function returns.

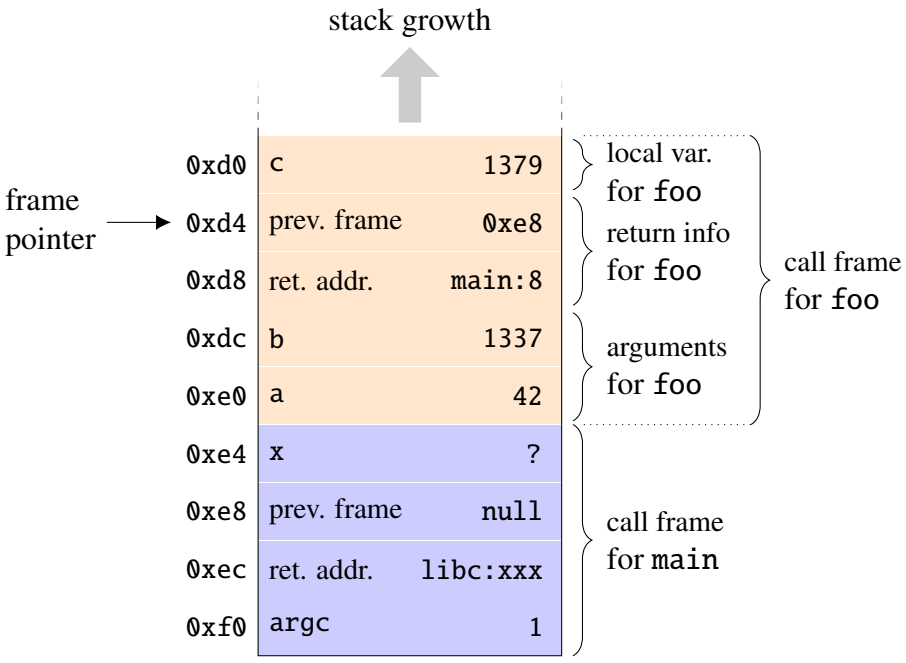


Figure 2.1.1: Anatomy of a call stack.

Allocating memory at the beginning of a function for its local variables and deallocating this memory once the function returns is called *stack allocation*. The term refers to the call stack (a.k.a. the execution stack) of a runtime system. On most computers and virtual machines, this is the structure responsible for storing the execution context of a thread. The details of its implementation are highly platform-specific. But at a more abstract level, it is merely a collection of memory blocks, also called *frames*. One is added to the collection at every function call, and the last appended block is removed every time a function returns. A frame typically contains the values of the variables and arguments of the function being called, as well as some additional data required to restore the execution context of its caller. As a result, memory for local variables is automatically allocated when a new frame is pushed onto the stack[1], and similarly automatically deallocated when the frame is popped out. An example follows:

Example 2.1.1: Call stack of a C program

Consider the following C program, which illustrates the application of a function onto two integer arguments:

C

```c
int foo(int a, int b) {
  int c = a + b;
  return c;
}

int main(int argc) {
  int x = foo(42, 1337);
  printf("%i\n", x);
  return 0;
}
```

Figure 2.1.1 depicts the state of an Intel x86 call stack as the program is about to return from `foo`*, at line 3. The blue blocks at the bottom of the figure correspond to the call frame for the* `main` *function, while the orange ones represent the frame for the* `foo` *function. Notice how the arguments are laid out first, followed by some information used by the runtime system to restore the context of the caller, and then by the local variables. To execute the return statement, the runtime system looks at the current value of its frame pointer, updates it to* `0xe8` *(i.e. the address of the previous frame),*

[1] Some runtime systems may push variables individually as their declaration is executed, rather than as a single block when the function starts. While this strategy may have implications with respect to time and space performances, it is irrelevant for the purpose of our discussion.

and sets the program counter to `main:8`*, so the next statement to be executed is a call to* `printf`*. In other words, the current frame is "forgotten" and so is its associated memory. Now considered deallocated, it can be overlaid by the next frame that is pushed onto the stack.*

The term *lexical scope* (a.k.a. static scope or simply scope) is often used in concert with stack-allocation. Strictly speaking however, a lexical scope does not relate to memory management, but rather to section of a program in which a given identifier (e.g. a variable name) is visible. In the code in Example 2.1.1 for instance, the variable `c`'s visibility is delimited by the function `foo`'s body (i.e. its lexical scope). The notion is relevant in the context of stack-allocation because the *frame* ensuing a function call contains space for all variables in its scope. Consequently, the removal of such a frame from the call stack actually coincides with the point in the program where the variables defined directly within the function's scope are no longer visible. For example, the moment at which the orange frame is removed from the stack in Figure 2.1.1 coincides with the moment at which the function `foo` returns, and therefore with the end of its lexical scope in the source code. For other classes of lexical scopes, such as branches of a conditional (a.k.a. `if`) statement, saying that a variable goes "out of scope" usually does not correspond to any actual operation on the memory at runtime. Nonetheless, some programming languages (e.g. C++) allow custom behavior to be defined at these points, typically by calling a particular function, called a *destructor*, which might be used to perform memory management operations when variables leave their scope.

While conveniently handling allocation and deallocation mechanisms, stack-allocated memory comes with important limitations. First, the size of the variables allocated on the stack must be decided statically (i.e. at compile-time), and cannot change during the execution of a program. Secondly, as memory is automatically linked to the lifetime of a frame, variables allocated on the stack cannot outlive the function in which they were declared. This also means stack-allocated variables cannot be referred to across threads. Finally, call stacks are commonly rather small in size, especially when implemented at the machine level, leaving them ill-suited for storing large data. *Heap allocation* solves these shortcomings by using memory from a region that is not managed automatically. Instead, the runtime system typically provides primitives (e.g. `malloc` and `free` in C) to manually allocate and deallocate arbitrary sections of that region.

Although heap allocation memory lifts some limitations of stack allocation, the latter is usually preferred to its counterpart for several reasons. Pointers to stack-allocated memory are usually represented as offsets relative to the current frame, which can be computed at compile-time and thus hardcoded into the byte-

code or machine code of a program. For instance in Figure 2.1.1, the pointer to the variable `c` is the value of the frame pointer decreased by four, and is an invariant in any of the function `foo`'s invocations. On the other hand, determining the value of a pointer to the heap involves a call to the runtime system which, in addition to a slight performance cost [143], eliminates opportunities for compile time optimizations. Other performance considerations can be made with respect to memory fragmentation [23], which may hinder predictability in real time systems. More importantly, the most significant drawback of heap-allocation is that it requires explicit deallocation. As heap memory is not linked to any another particular aspect of a program's execution, it must be freed manually.

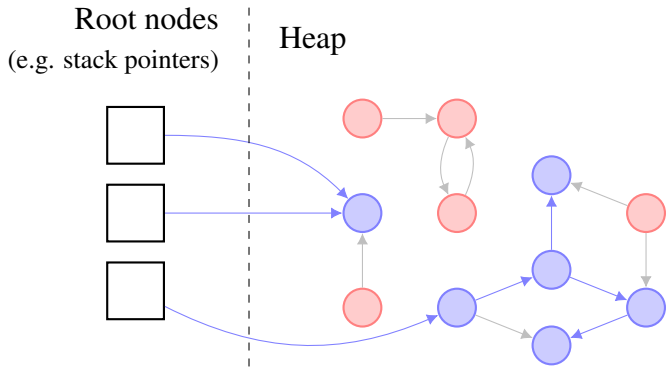
2.1.2 Dynamic Garbage Collection

Managing heap memory manually (e.g. by explicitly calling `free` in a C program) is tedious and error-prone. Forgetting to free memory creates leaks, which may exhaust available memory, whereas premature deallocations may give rise to other memory errors, potentially leading to unpredictable results or even crashing the program altogether. To alleviate this issue, some runtime systems (or third party libraries) come equipped with garbage collection strategies that operate automatically behind the scenes. There exist several techniques related to automatic garbage collection, but they ordinarily fall under one of two categories.

Tracing garbage collection is arguably the most popular method. In a nutshell, tracing garbage collection consists in periodically identifying and deallocating the memory that is no longer in use, by checking whether there is no chain of pointers (or references) from a local or global variable to said memory. In other words, if allocated memory is understood as nodes of a directed graph connected by pointers, a *mark* stage creates for each variable a spanning tree of its reachable subgraph, and a *sweep* stage frees all nodes not present in such trees. An example is given below. While all follow the same two-stage principle, there exists numerous implementations of tracing garbage collection, with varying performance and consequences on memory fragmentation. However, a complete discussion is beyond the scope of this work. A comprehensive account is given in [142].

Example 2.1.2: Tracing garbage collection

Consider the following picture, which illustrates the operation of a tracing garbage collection algorithm at the end of the mark stage:



On the left are represented the root nodes, which usually correspond to the local and global variables of the program. On the right is represented the heap, in the form of a non-connected graph of data structures linked by references. Blue edges represent the branches of the spanning trees constructed from the root while other links are left in gray. Similarly, nodes accessible from a root are drawn with blue circles while others are drawn in red. The sweep phase simply consists of deallocating all red nodes.

The cost of the mark stage of a tracing algorithm is linear to the size of the heap. To mitigate the impact on performances, garbage collection is typically ran only under certain conditions (e.g. the total amount of heap allocated memory surpasses an acceptable threshold). The unfortunate consequence of this strategy is that the precise moment memory is deallocated becomes unpredictable.

Reference counting solves this latter issue. Representing once again memory blocks as nodes of a directed graph, reference counting consists in keeping for each node a counter of the number of pointers thereupon. The counter is incremented every time a new pointer is created and decremented every time one is removed (i.e. when a pointer is reassigned or destroyed). If the counter reaches zero, then the associated node is immediately freed. Because the deallocation happens synchronously with the decrement, reference counting can be used to implement predictable destructors (i.e. hooks that are triggered when an node is destroyed). These allow for safer custom allocation and deallocation models, such as the "Resource Acquisition is Initialization" idiom [128]. Besides, counter increments and decrements are relatively cheap operations, and more importantly predictable with respect to performance. Hence, reference counting is usually a better fit for real time systems than its tracing counterpart. Dealing with reference cycles is however challenging, often requiring the addition of special attributes to avoid them (e.g. weak references [55]). Those are error-prone, as they require continuous awareness of the runtime system's memory management scheme.

2.1.3 Static Garbage Collection

Some systems may not afford to run dynamic garbage collection algorithms concurrently with the program they execute, due to their impact on performance (e.g. in the context of real time systems). Another approach is to statically identify the point from where the value assigned to a particular variable is no longer needed, in order to automatically insert deallocation instructions at those points [21]. This technique solves the main issues of both tracing and reference counting, as it does not put any computational pressure at runtime, and is impervious to reference cycles. Instead, its drawback resides in a weaker flexibility.

Determining the lifetime of a pointer at compile-time is undecidable, which theoretically restricts static garbage collection to the smaller subset of cases where it can *prove* a bound on lifetimes. In practice there are multiple workarounds. A first one is to simply ignore cases where lifetimes cannot be statically established and rely on a runtime deallocation strategy. Static garbage collection becomes a sort of optimization technique that can hopefully reduce the runtime overhead of automatic garbage collection. A popular concretization of this approach is to convert heap allocations to stack allocations whenever possible [22]. Another workaround is to conservatively approximate a maximum bound on lifetimes. While this method may not be able to free memory as soon as possible, it guarantees that memory is never deallocated early, but always eventually.

Even in decidable cases, computing lifetimes can be a costly task that does not scale well with the size of a program. Indeed, a pointer might be passed in and out of numerous function calls, which all have to be considered to determine from where the pointer eventually remains unused. Annotations can help alleviate this issue [13], but they also clutter function signatures. Another approach is to augment the language's type system with inferable information related to variable lifetimes. While this usually results in much more constrained programing languages [96], conservative scenarios are simpler to establish, hence addressing the scalability issue. Literature on this technique is discussed in Section 2.3.

2.1.4 Object Abstraction

One essential pattern of object-oriented programming is the concept of *object abstraction*. Object abstraction is used to group data into logical objects one can reason about at a higher level. An object is essentially an aggregate of *fields* (a.k.a. attributes or properties), which typically are themselves represented as other objects. It may further be associated with functions, usually called *methods*, that can read and modify the object's fields through a contextual reference (most often `this` or `self`) that refers to the object itself. This allows the specifics of its representation to be hidden from the outside, a principle referred to as *encapsulation*. A

good example is a tree structure. Represented as a collection of individual nodes linked from parent to children, a tree object should only be referenced by its root, and all operations thereupon (e.g. inserting a new value) applied from the root and propagated down as necessary. Figure 2.1.2 exemplifies the such a representation.

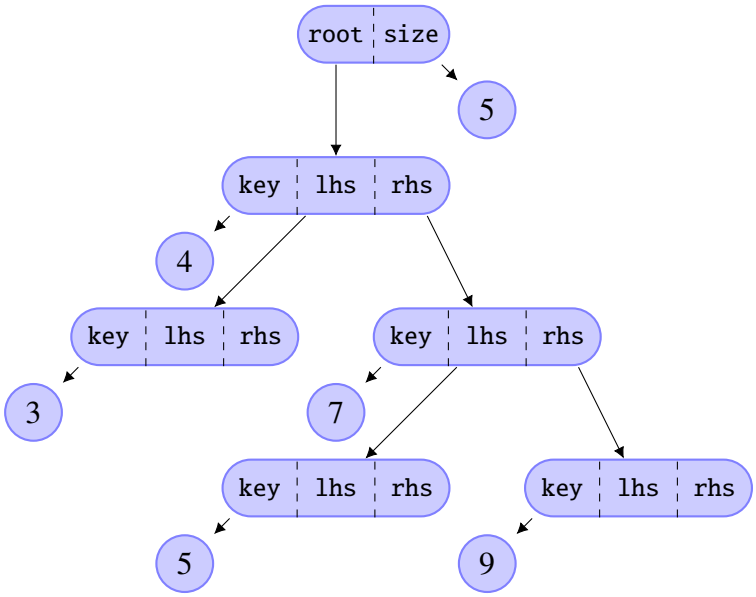


Figure 2.1.2: Object representation of a binary tree.

The advantages of object abstraction and encapsulation are twofold. First, it allows for a better separation of concerns, drawing a clear boundary between the internal representation of the object and its externally visible interface. Note that a representation is (often) not necessarily a one-to-one correspondance with the abstraction the object offers. For instance, a binary tree such as that of Figure 2.1.2 could serve as the representation of a list or a set. Fortunately, a clear distinction between interface and representation frees developers from worrying about implementation specifics in most situations. Secondly, object abstraction and encapsulation let one reason about objects invariants in complete isolation, precisely because of that separation of concerns. For instance, one may write an internal method on a binary tree that works under the assumption it may never encounter any circular reference. Thanks to abstraction, this property can be expressed on the nodes directly rather than on the pointers that constitute them, while encapsulation can be used to make strong assumptions about the way an internal representation can be altered.

Nothing in the above definition is said about *where* the object representation is supposed to be stored, leaving the door open to both stack and heap allocation strategies. Indeed, as an object is essentially an abstraction, its representation

could be stored in the form of contiguous memory on the stack or in the form of a collection of pointers to the heap. Nevertheless, object-oriented languages tend to favor the latter, as it is better suited for other common object-oriented patterns, such as inheritance and polymorphism [108]. Furthermore, while static analysis does not necessarily assume a particular allocation strategy, most of the approaches that will be discussed were primarily devised to reason about heap-allocated memory, and so we will adopt the same position in the remainder of this chapter.

2.2 Formal Methods

Formal methods have for objective to prove (or disprove) the correctness of a system, with respect to given a specification of this system. Unlike approaches based on testing and/or simulation, formal methods typically do not derive results from the observation of a system's behavior, but on a mathematical description (i.e. a model) from which the exhaustive set of all possible behaviors can be inferred. This mathematical description must agree to a meta-model, that formally defines how a model can be constructed (i.e. its syntax) and what it means (i.e. its semantics). In other words, a meta-model is a model of a model.

There exist innumerable meta-models, specialized to various applications, and a complete review is beyond the scope of this thesis. Instead, this section settles for a far humbler goal and focuses on language calculi.

2.2.1 Language Calculi

A language calculus is a formal system that can express a model of computation, usually in a fashion resembling a minimalistic programming language. A good way to introduce this concept is through the lens of the λ-calculus [83]. The λ-calculus is a model that expresses computation solely in terms of function application, therefore representing the core semantics of functional programming. Syntactically, the meta-model is described by the following grammar.

$$e \coloneqq x \mid \lambda x.e \mid e\ e$$
$$x \coloneqq a, b, c, \ldots$$

In plain English, let x denote variable names, an expression is either a variable x, an abstraction $\lambda x.e$ or an application $e\ e$. Semantics is expressed by two transformation rules, which describe how to *reduce* an expression. The first is called α-conversion (a.k.a. α-renaming), and consists in substituting every *free* occurence

of a particular variable name with another. For example, let $\rightsquigarrow$ denote the transformation of a term:

$$(\lambda a.(b\ \lambda a.b)\ b)[^{b}/_{c}] \quad \rightsquigarrow \quad \lambda a.(c\ \lambda a.c)\ c$$

α-conversion is more complex than a naive term substitution, because is does not replace variables that are bound by an abstraction. In fact, the body of an abstraction defines the lexical scope in which its argument is visible. Consequently, it is not affected by the renaming of a variable defined in an enclosing scope, even if it has the same name. For example:

$$(\lambda a.(b\ \lambda a.b)\ b)[^{a}/_{c}] \quad \rightsquigarrow \quad \lambda c.(b\ \lambda a.b)\ b$$

The second transformation rule is called β-reduction, and describes function application. Let $\lambda a.e_1$ be an abstraction and $\lambda a.e_1\ e_2$ its application on an expression e_2, β-reduction consists in applying α-conversion on e_1 to substitute every free occurence of a with e_2. For example:

$$\lambda a.(b\ \lambda a.b)\ b \quad \rightsquigarrow \quad (b\ \lambda a.b)[^{a}/_{b}] \quad \rightsquigarrow \quad b\ \lambda a.b$$

Despite its minimalistic nature, the λ-calculus is equivalent to a Turing machine [137], and therefore can be used to model any kind of programming language. However, this simplicity also presents drawbacks when describing elaborate programming concepts. For instance, expressing boolean algebra in the λ-calculus is tedious and unnatural, whereas representing advanced notions such as classes is simply unrealistic. As a result, a plethora of variants have been proposed over the years. We discuss a few of them below.

Calculi for Assignment

Early proposals to bring assignments to the λ-calculus include λ_{var} [111], an extension that features assignable variables. Assignable variables are introduced into an expression by an additional construct `let` x `in` e, which delimits their lexical scope. They only differ from regular variables in that they can appear on the left operand of an assignment. Sequences of instructions are handled with monads. A subsequent line of research focus on equipping λ_{var} with type systems (e.g .[147, 37]) to guarantee freedom from side effects for pure applicative terms. Another approach from the same era formalizes the notion of memory location, in order to support a call-by-value approach [59]. Similar to λ_{var} the authors propose assignable variables, but those are interpreted as memory locations in terms' reductions. This semantics is revisited in ClassicJava [62] to formalize a subset of the Java language.

In his thesis [45], Colby also advocates for a semantics expressed in terms of a transition system that maps program states onto program states, unlike the fixed-point computation commonly used to describe the λ-calculus' semantics. The advantage resides in that program analysis can defined over evaluation traces, rather than phrased in terms of a type system's inference rules. A similar idea is explored in [15] to analyze imperative programs.

Calculi for Objects

The resounding success of object-orientation in the last couple of decades has pushed research to direct a lot of effort toward the formalization of object-oriented calculi. The λ-calculus and close variants are not ideal to reason about object abstractions, as they rely on function application to encode data structures. This is unnatural in the context of object-orientation and yields a significant abstraction gap. Concurrently, higher-order functions are not nearly as pivotal, first because objects propose an alternative to data encapsulation, and secondly because object-oriented languages programming language are traditionally more tilted toward an imperative paradigm.

Among the most popular object calculi is Featherweight Java [81], that aims to provide a sound base for studying the consequences of extensions to the Java language. As a result, it actually discards most of Java's features, including statefulness, and only keeps a functional subset. Evidently, support for assignments is a natural extension for Featherweight, and is proposed in a variety of models (e.g. [100, 20]). Other noteworthy extensions include aspects of concurrency [113] and functional programming [17].

More recently, [34] introduces a calculus whose semantics is not defined with an auxiliary memory structure to represent assignable variables' bindings. Instead, the authors proposes to stay at a higher abstraction level and reduce assignable variables to the value they represent. Such values can be either irreducible data (e.g. a number) or another variable. Their approach is set in the context of object-oriented languages, and thus manipulates class instances. Consider for instance the following term, assuming the existence of a class C with a single integer field i:

$$a = \texttt{new}\ C(42)\ ;\ b = a\ ;\ b.i$$

The term is composed of three instructions. The first declares a new instances of the class C, initialized with the value 42 for its field i, and assigned to a variable a. The second instruction assigns an alias on a to the variable b. The third dereferences the field i of the instance referred to by the variable b. By substituting expressions to their respective values, the term can be reduced as follows:

$$a = \texttt{new}\ C(42)\ ;\ b = a\ ;\ b.i \quad \rightsquigarrow \quad a = \texttt{new}\ C(42)\ ;\ a.i \quad \rightsquigarrow \quad 42$$

The advantage is that aliasing is conveyed at a syntactic level, whereas approaches based on an external store requires aliasing information to be derived from variable mappings. The language also supports a notion of block to represent hierarchical topologies of references, enabling a reasoning about aliasing constraints purely on the syntax. The model is further extended in [67] to integrate aliasing control mechanisms.

2.2.2 Software Verification

Mathematically describing a system allows for said system to be formally *verified*. Furthermore, as formal models are described with a formal semantics, such verification can be automated.

Automated Theorem Proving

Automated theorem proving (a.k.a. automated deduction) is arguably the most mathematically-oriented approach to software verification, with applications dating back the 50s [112]. Despite this seniority, modern tools such as Coq [16] or Isabelle/HOL [109] consistently demonstrate the relevance of (semi)-automated theorem proving, with remarkable results such as CompCert [95], a fully fomally verified compiler for the C language. However, a significant drawback of this technique is that it most often demands a heavy manual effort to help the deduction process, with a very high level of expertise.

Model Checking

Model checking [43] is a term that proxies a vast array of methods that aim at detecting implementation problems by checking whether the system's model follows a given specification. Model checking corresponds to the exhaustive exploration of all possible configurations, so that verification boils down to checking that the specification is satisfied in each of the enumerated configurations. Furthermore, the result of such check is extremely accurate, as one can determine exactly which configurations violate a given invariant, hence effectively providing concrete counter-examples.

Though early papers often refer to the model as a Kripke structure, model checking is in fact agnostic of the formalism used to model a system, as long as its evolutions can be formally defined. It follows that language calculi obviously fulfill these requirements. Programming languages themselves can even be seen as suitable formalisms, though verification tools that can consume real programming languages as an input are scarce. Another more common approach is to automatically transform a source code into a formalism better-suited for model checking.

So as to mitigate the complexity of the actual input language, models are often extracted from the intermediate representation (IR) of the compiler (e.g. Java bytecode). Noteworthy efforts toward that direction include Java PathFinder [75], that translates Java code into Promela [79], MoonWalker [48], its counterpart for the Microsoft .NET Framework. Less established tools have been proposed to extract model specifications from LLVM [12, 90].

Unfortunately, the appeal of model checking is often met with the daunting problem of *state space explosion* [44], in particular when modeling software. The number of states reachable by a typical program is most often too large to be enumerated, because of the combinatorial explosion of the values it can manipulate. Consequently, important research efforts have been aimed at containing this state space explosion [91].

2.3 Static Program Analysis

Unlike the more general formal software verification approach (c.f. Section 2.2), static program analysis attempts to prove invariants sufficient to prescribe correctness on the *shape* of a program's evaluation, rather than its actual execution. Put differently, whereas the former aims at proving the full functional correctness of a given system, the latter settles for far humbler goals and focuses on a statically identifiable error patterns [94]. For instance, a compiler for a statically typed language will usually produce an error if one attempts to assign a variable to a value with mismatching type, but will be unable to guarantee that a division by zero cannot occur.

Static program analysis has been successfully applied to tackle numerous challenges, ranging from code optimizations to code defect detection. This section discusses the techniques most related to this thesis' contributions, with a particular focus on Ownership Types and type capabilities.

2.3.1 Typestate Analysis

Typestate analysis [66] is a refinement of the concept of type analysis. The traditional purpose of a type, in a typed language, is to restrict the operations supported on a particular value. For instance, a variable might be declared holding numerical values, therefore allowing operations working on numbers (e.g. the addition) to be applied to it, but preventing its use in operations working on character strings (e.g. the concatenation). A compiler (or more precisely its type checker) may use this information to accept or reject inputs, hence eliminating programs that would otherwise attempt to apply an operation improperly. A type typically remains constant throughout the entire execution of the program, though many operations

may actually be nonsensical at some specific points. For example, while a file descriptor might be used as argument for a write command, such an operation will most likely fail if the file descriptor has not yet been properly initialized. Following this observation, a typestate adds further restrictions on the set of applicable operations, which do indeed depend on the particular context in which a data is used. One could define an associated typestate for the file descriptor that describes whether or not it is opened (i.e. initialized), and hence disallow the application of write commands if necessary.

Though in their work [127] Strom and Yemini frame typestate in the context of static verification of variable initialization, subsequent research have aimed at verifying more complex objects invariants [50, 18]. However, one recurring criticism of typestate systems is that they are difficult to apply in practice. Adoption has remained scarce in actual programming languages, with the notable exception of Plaid [3], and tools supporting typestate on top of existing programming languages suffer from the burden of maintaining both an implementation and its specification alongside each other.

Another common weakness of typestate systems is a poor support of aliasing, as propagating typestate changes on all variables that may hold a reference to the same object is often intractable. Most attempts to tackle this issue either rely on expensive whole-program verification approaches [60], focus on strong aliasing restrictions at the expense of expressiveness, or use sophisticated type annotation systems. We elaborate further on these different approaches in the remainder of this section. [144] proposes mixing static and dynamic typing, a practice usually referred to as *gradual typing*, in order to alleviate the need for complex typestate specifications.

2.3.2 Ownership Types

The core idea behind Ownership Types is to limit the visibility of changes [110, 42] of an object, based on the observation that aliasing and mutation are not intrinsically problematic, but rather that changes in one object may unintendedly break invariants in another. In other words, Ownership Types aim at enforcing the encapsulation principle. The essence of the approach is to partition the memory into topologically nested regions, called *ownership contexts*, whose ownership is attributed to a particular object. Intuitively, a region often corresponds to the representation of an object, while the owner of such a region is the root of that representation. For example, on the binary tree from Figure 2.1.2, one could delimit an ownership context around each node of the tree, naturally nested within that of its parent. Not all properties of an object must necessarily be part of its ownership context. Instead, some references might be explicitly placed in another context, so as to denote, for instance, shareable objects.

An object is said to be *owned* by another one if it is part of the latter's ownership context. The term "Ownership Types" stems from the fact that information about an object's owner is part of its type, and is usually defined statically. To that end, most Ownership Types systems (or ownership systems for short) propose at least two annotations, conventionally `rep` and `world` (a.k.a. `norep` in older literature), which respectively indicate whether a property is part of an object's representation, or if on the contrary it is to be considered shareable among all components of the program. This in turn can be used by an owner to restrict access to its representation, under different policies. Advocating reusability, ownership systems also generally offer some level of genericity on ownership annotations, so that the same type might be instantiated in different contexts. For instance, one could define a container with a generic parameter that controls in which ownership context its elements should be placed.

The advantage of ownership systems over the typical access control most object-oriented languages offer is that aliasing restrictions are baked into references' types. In Java for instance, although one can protect a property from being directly accessed by marking it private, this does not prevent a method from returning a reference on that property, which then might get freely aliased and create a backdoor into the object's internal representation. In an ownership system, the return type of such a method must contain information about the returned value's owner, which can be used to forbid the creation of an alias, or limit its capabilities.

The remainder of this sub-section is a discussion about the most influential ownership systems, in particular on those related to memory and alias safety. We refer the reader to [39] for a more detailed survey on Ownership Types and their applications to other domains.

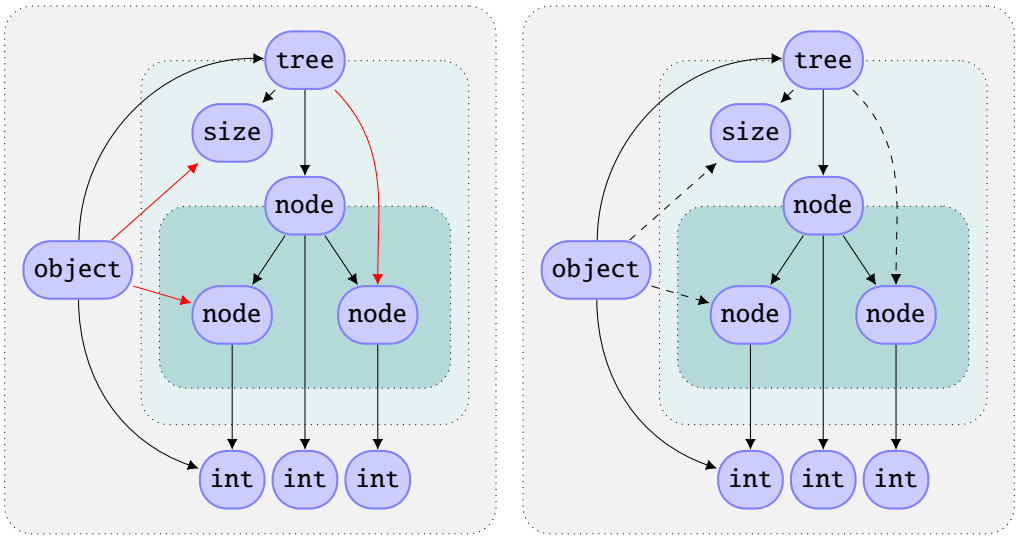
Ownership For Topological Restrictions

The *owners-as-dominators* [42] policy (a.k.a. *deep ownership*) is the first and most restrictive approach. It prescribes all accesses to an object owned by some owner to go through the methods of that same owner, effectively forbidding any outside alias to the internal representation of an object. On the other hand, the owner can freely manipulate aliases within its own representation, and be given permission to alias the direct representation of an enclosing object. In other words, an object a may hold an alias on b if either a is the owner of b, or b is owned by an object higher in the hierarchy than a. This structures the memory as a tree, as siblings may not alias each others' representation, and leads to an interesting theorem stating that an object's owner necessarily appears on any path to that object [41] (hence the term "dominator").

While the owners-as-dominators scheme perfectly captures the essence of encapsulation, it often appears too restrictive in practice. In particular, it excludes

patterns that do require access to an internal representation, such as iterators and command objects, which therefore can only be heavily worked around [40]. The *owners-as-modifiers* [103, 51] policy relaxes the access restrictions by supporting unconstrained read-only aliases, while updates must pass through the owner. This enables a programming style where the properties of an object can be directly referenced, which is more in line with current software practice that discourages unnecessary getters [76].

An example is depicted in Figure 2.3.3. The dotted rounded rectangles delimit the contexts, whose hierarchy is represented topographically, and whose respective owner is drawn on their top border. The outer rectangle, drawn in gray, represents the `world` context. Figure 2.3.3a illustrates the owners-as-dominators strategy, which forbids the arrows drawn in red. Figure 2.3.3b illustrates the owners-as-modifiers strategy, which only allows read accesses, denoted with dashed lines, from outside an ownership context.



(a) Owners-as-dominators. (b) Owners-as-modifiers.

Figure 2.3.3: Ownership access policies.

The owners-as-modifiers is among the ownership systems to have received the most attention in particular with respect to program verification [52]. Nonetheless, numerous other proposals exist, with varying levels of expressiveness. In his thesis [24], Boyapati gives nested class objects[2] a privileged access to the internal representation of its enclosing class instances. Furthermore, nested class instances are allowed to be owned externally, hence enabling the implementa-

[2]A nested class (a.k.a. inner class) is a class declared within the declaration of another, often used to describe a dependent type.

tion of constructs that require external access to an object's representation. For instance, an iterator can be implemented as an inner class of that representing the associated collection, so as to enjoy an unconstrained access to the collection's representation. Tribal Ownership [32] adopt as similar view, but proposes to leverage the natural hierarchy described by virtual class families to infer ownership relations. Ownership Domains [4] generalize the concept of privileged relationships between ownership contexts, allowing an object to define multiple ownership domains, and let the developer specify fine-grained aliasing rules between these domains and are consequently more flexible. Comparable results can be obtained with systems that allow multiple owners [31, 114].

Arguing approaches relaxing the constraints on internal representations accesses treat encapsulation too loosely, the owners-as-accessors [115] discipline does not proscribe any alias (read or write), but ensures they cannot be used (for neither read or write) unless their owner appears on the call stack. This guarantees the latter the opportunity to maintain its invariants, for exemple allowing a tree object to properly updates its size whenever a node is added or removed from its internal representation.

Ownership For Memory Safety

Other ownership systems also trade safety assumptions on encapsulation for more relaxed aliasing policies, but preserve important guarantees on memory safety. In Clarke's thesis [41], *owner-polymorphic methods* may be granted a temporary unrestricted access to the internal representation of the object that calls them. In essence, this corresponds to the notion of reference borrowing (c.f. Section 2.3.3), as the model can be understood as an object that temporarily lends its ownership. The rationale is that separation of concerns is preserved, as an owner ultimately retains control over its context, as long as the receiver of an alias is forbidden to store it outside of its method-local variables.

An even more permissive system is suggested in Wrigstad's thesis [145], lifting aliasing restrictions to objects whose owners live in either the same or an older stack frame. An example is portrayed in Figure 2.3.4. Black arrows denote valid references, while red ones show illegal ones. In both systems, the resulting ownership relations form a directed acyclic graph rather than a tree, but with the guarantee that dangling pointers may not occur, even with a stack-allocation strategy. [26] formalizes this intuition, combining ownership with region-based memory management.

Ownership systems have also been employed to ensure data race freedom. In [25], ownership is associated with the notion of locks (thus the occasional use of the term owners-as-locks in related literature). The ownership system prescribes that acquiring a lock on some owner is the sole and sufficient condition

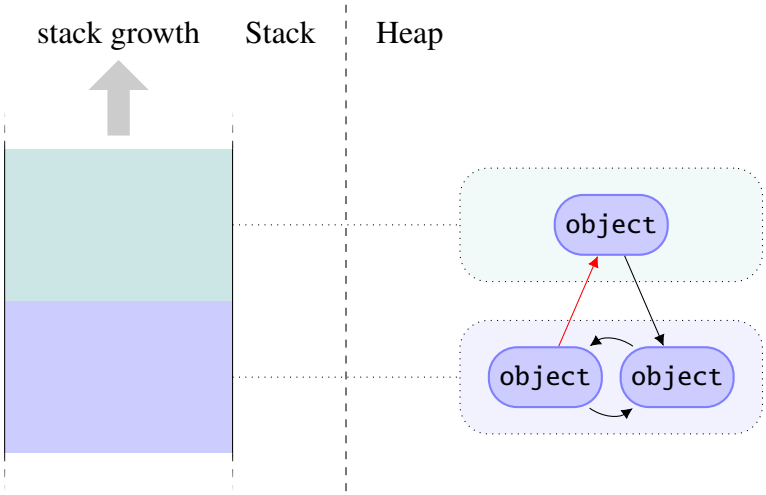


Figure 2.3.4: Stack-based (a.k.a. generational) ownership.

to gain exclusive access on the members of its context. A different approach is taken in [146], using ownership to distinguish between thread-local and shareable portions of the heap. Threads define ownership contexts, within which memory accesses are unconstrained. On the contrary, accessing an object in a shareable portion of the heap requires synchronization.

2.3.3 Substructural Type Systems

Type systems are usually formalized in the form of a deductive system based on inference rules. For instance, the classical Hindley-Milner type system for the λ-calculus defines the following rule to type applications:

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}$$

This rule states that if it can be deduced that e_1 has the function type $\tau \to \sigma$ and that e_2 has the type τ, then $e_1\ e_2$ has the type σ. Let Γ be defined as a sequence of assumptions of the form $[x_1 : \tau_1, x_2 : \tau_2, \dots]$, denoting that x_i has type τ_i, formal deduction offers three structural properties to establish proofs with such an inference rule:

- **Exchange** states that the order in which the assumptions appear in Γ is irrelevant (e.g. $[x_1 : \tau_1, x_2 : \tau_2] \vdash e : \sigma \implies [x_2 : \tau_2, x_1 : \tau_1] \vdash e : \sigma$).
- **Contraction** states that identical assumptions in Γ may be replaced by a single occurence (e.g. $[x_1 : \tau_1, x_1 : \tau_1] \vdash e : \sigma \implies [x_1 : \tau_1] \vdash e : \sigma$).

- **Weakening** states that adding unnecessary assumptions to Γ has no influence on inference (e.g. $[x_1 : \tau_1] \vdash e : \sigma \implies [x_1 : \tau_1, x_2 : \tau_2] \vdash e : \sigma$).

Substructural type systems are type systems in which one or several of these properties do not hold. The choice of the properties that are discarded leads to different type systems:

- **Linear type systems** abandon contraction and weakening. Variables cannot be used more than once to type an expressions without contraction, and cannot be ignored without weakening. Thus a linear type system requires all variables to be used *exactly* once.

- **Affine type systems** abandon contraction. Like in linear type systems, variables cannot be used more than once to type an expression without contraction. However, weakening still allows them to be discarded. Thus an affine type system requires all variables to be used *at most* once.

- **Relevant type systems** abandon weakening. Like in linear type systems, variables cannot be ignored without weakening. However, contraction still alows them to be used multiple times. Thus a relevant type system requires all variables to be used *at least* once.

- **Ordered type systems** abandon all structural properties. Thus all variables must be used *exactly* once, like in linear type systems, and the order in which they are used cannot be arbitrary.

All four type systems above have practical applications for program analysis [2], however linear and affine type systems have received more attention.

Linear types [140] are well-suited to reason about memory. A linear resource must be used exactly once, and therefore infering its lifetime is trivial: once it has been read, its memory can be deallocated. Another evident application for linear types is to assert assumptions on resource usages. Consider for example a file descriptor. Using linearity a type system can determine statically that an opened descriptor is eventually closed, and that a closed descriptor cannot read. The alert reader will notice the similarity with the objectives of typestate analysis (c.f. Section 2.3.1). The abandon of weakening leaves linear type systems difficult to use in practice, because they require that all variables be necessarily read. Affine type systems [135] relax on this constraint, and only preserve the uniqueness requirement. This corresponds to the approach adopted by Rust [89] for instance.

Linearity also implies uniqueness, because it proscribes cloning. This means that although a linear resource may be successively assigned to different variables, all assignments must necessarily incapacitate their right operand, for instance by

setting a reference to the null pointer, or by guaranteeing that it can no longer appear on the right side of an assignment [27], or passed as a function argument. Hence linear type systems present themselves as an elegant solution to prevent unintended sharing of state.

2.3.4 Type Capabilities

Originally proposed as a generalization of reference (or pointer) annotations [29], type capabilities (a.k.a. permissions) encompass techniques aimed at extending types with permissions. Just as Ownership Types, they present themselves as properties of references that a compiler may use to reason about the correctness of an instruction, given its use in a particular context. However, whereas most ownership systems typically apply a single policy on all variables (with the notable exception of Ownership Domains [4]), type capabilities can offer a finer level of control over aliasing restrictions. In a sense, they can arguably be seen as a more generalized yet more powerful extension of Ownership Types.

Type capability systems usually aim at formalizing one or both of two properties, namely *uniqueness* and/or *mutability*. The former is sufficient to prescribe safety from data races. Indeed, if an object is known to be uniquely referred, then one can safely assume any mutation of said object not to break invariants elsewhere in the program.

In his seminal work on *fractional permissions* [28], Boyland sees uniqueness as a fractionable capability, whose each piece is held by an alias on a given object. If the object is uniquely referred, then its referent holds all the pieces, which assembled together form the uniqueness capability, necessary to cause mutation. However, whenever an alias is created, one these pieces is lent and therefore the uniqueness capability is temporarily lost, until said alias is destroyed.

Despite the appeal of representing uniqueness as a fractionable capability, accounts of use attempts in real settings have proved Boyland's system to be hard for software developers [104]. Automated approaches have to rely on pointer analysis, which can seem fragile when applied to large code bases, while using annotations can impede maintainability. As a result, a handful of related approaches have been proposed to simplify uniqueness management, while retaining sufficient expressiveness to prescribe reasonable guarantees on data safety. Clarke and Wrigstad advocate for *external* uniqueness as an adequate relaxation [40]. External uniqueness is an extension of the owners-as-dominators scheme, which prohibits multiple references on owners from outside their ownership context, hence allowing the enforcement of object locality at function boundaries. Unlike Boyland's proposal, it naturally aligns itself with encapsulation and therefore can be supported by a fairly simple annotation system. Two operations are supported: movement (which roughly corresponds to the treatment of references as linear re-

sources [68]) and borrowing. The former allows one externally unique object to be reassigned to another variable, provided that the one to which it was originally assigned is nullified, or known to remain unused for the remainder of its lifetime [27]. The latter consists of temporarily lifting the uniqueness of a reference, for the duration of a lexical scope. Since ownership ensures that no reference escapes the borrowed construct, uniqueness can be expected to be recovered at the end of said scope. For instance, a method call would have to borrow an externally unique reference so that `this` could be used for the duration of the call. This mechanism is certainly analogous to Boyland's fractional permissions, but leverages lexical scoping to be easier to predict. Unfortunately, reliance on the rather constrained deep ownership imposes significant restrictions in terms of expressivity (c.f. Section 2.3.2). The issue can be mitigated by relaxing the ownership [5], though at the cost of a weakened notion of encapsulation. The approach also requires additional constraints to guarantee safety in multi-threaded contexts, in which ownership alone no longer suffices to prevent data races, as aliases may be accessed in another thread, effectively breaking uniqueness.

In [73], Haller and Odersky recede from explicit ownership systems in favor of a dedicated annotation system, pointing out that ad-hoc inference of uniqueness often struggles to treat borrowing elegantly. In their system, a reference is guarded by a certain capability representing a region of the heap, and can only be accessed if such a capability is available. As capabilities denote regions, a unique reference represents in fact an entry point to an externally unique aggregate. Variables can be annotated `@unique`, so that their capability is treated affinely. In other words, use of a unique reference in a method call consumes its capability and leaves it unusable in all contexts, which contrasts with Clarke and Wrigstad's external uniqueness. Borrowing is indicated by an additional annotation `@transient` on method parameters, denoting that the method does not consume the capability. Marking a parameter by `@peer(y)` indicates that it is guarded by the same capability as `y`, which is necessary to handle cases requiring access to two elements of the same aggregate (e.g. to assign a field of `y`). Practical experience with such a type system have been conducted with an extension of Scala's annotations [73], but have shown to be unsound when combined with all the intricacies of Scala's type system [121]. A more recent attempt, named LaCasa [72], uses Scala's implicits instead, and focuses on the actor model. Another effort at bringing uniqueness as a built-in capability in the form of an annotation system can be found in [104], with the objective to provide a more flexible support of borrowing. The most distinguishing feature of this approach is that rather than relying on a purely flow-sensitive inference, the authors propose to annotate parameters with the change of permissions they will experience upon calling the method. Their type system also supports other capabilities to convey additional properties such as immutability and thread locality, hence allowing for a large array of situation to be described.

In situations where uniqueness cannot be guaranteed or would result in a too constrained system, mutability (or immutability) can be used instead to derive data race freedom. There, the idea is to limit the contexts in which a particular object may be modified, so as to establish sound assumptions on invariants [3]. In [104], a reference can be declared immutable, in which case it can only be used for reading. The type system further imposes that other references on the same object are also immutable, so a referred object by an immutable reference is guaranteed to remain constant. Nonetheless such restrictions are not permanent, as a reference may get its right access restored once immutable aliases go out of scope. Similar ideas are presented in [69], which additionally propose a readable capability to denote references through which an object cannot be mutated, but that do not necessarily proscribe concurrent mutable references.

2.4 Synthesis

Across Chapter 1 and the present chapter, we have introduced various problems related to memory treatment in imperative programming languages, and discussed different techniques to mitigate the difficulty of writing correct programs. Furthermore, we have showed that aliasing poses formidable challenges, as it breaks the symmetry between the lexical shape of a program and that of its execution. We close this journey with a brief summary of these approaches.

Garbage Collection Although automatic garbage collection has for primary objective to automate deallocation, its adoption generally allows other error-prone tasks to be avoided [142]. Dynamic garbage collection performs this task at runtime, and thus imposes a non-negligible overhead on space and/or memory. Static garbage collection solves the latter, but requires either costly pointer analysis that do not scale well with the size of the program, or conservative assumptions that limit expressiveness.

Language Calculi Language calculi are mathematical tools to reason about computation, and hence the core semantics of the concepts a language manipulates. Most models for imperative assignments augment the λ-calculus' semantics with a notion of store in order to associate values to assignable variables [111, 59].

The success of object-orientation has motivated a shift from approaches purely based on function application to a native support for object abstrac-

[3] This observation is also the foundation for Ownership Types.

tion, at the expense of higher-order constructs[4] [81].

Although side effects have been studied extensively through the lenses of type systems [2], models formalizing different flavors of assignment semantics are lacking.

Software Verification While software verification promises grandiose benefits, its use in practice is faced with important limitations. Approaches based on automated theorem proving require a very high level of expertise, unaffordable to most software development. Model checking addresses this issue with a more automated approach to satisfy proofs, but does not scale well to large software bases [44].

Type Systems Type systems can be leveraged to establish safety invariants on both memory and aliasing. Ownership Types [39] focus on the latter, and propose to partition the memory into topologically nested regions to restrict aliasing. Most related approaches aim to enforce object encapsulation, while offering enough flexibility to express common programming patterns.

Type systems can also check memory safety properties by considering flow-sensitive capabilities, which dictate what operations are allowed to be performed on a particular reference. Typestate analysis formalize these capabilities in the form of state automata [66]. Substructural [2] and capability-based [29] type systems impose restrictions on resource usage.

Functional programming languages usually do not suffer memory issues as much their imperative counterparts. While memory leaks may still constitute a significant concern in implementations based on lazy evaluation [71], memory safety is generally guaranteed, and data races are eluded by the reliance on immutable values. Conversely, mutation is at the core of the imperative paradigm. Therefore, imperative languages cannot afford to fully abstract over memory, despite all the clever ruses that have been discussed above to simplify memory treatment and catch improper use. This thesis argues that such a quest is a non-goal, and that a sane formal model to describe assignment should embrace the concept of memory, in order to mark an unequivocal distinction between variables, memory and the relation therebetween. The remainder of this thesis discusses such a formal model, its properties and implementation. The core of our approach is a language calculus that focuses on assignment semantics, so as to describe the interactions with precision, at the operational level. Although we also abstract over the notion of pointers, the creation of aliases is syntactically explicit.

[4] A first-order object-oriented system can nonetheless encode partial application through defunctionalization [120].

Chapter 3

Mathematical Preliminaries

This chapter introduces the mathematical concepts and notations that are used throughout this thesis. In particular, we describe how we formally represent abstract data structures, such as composite (a.k.a. compound or aggregate) data types and symbol tables (a.k.a. associative arrays). A summary of our notations is provided in Appendix A, and is intended to be a companion for the remainder of this document.

3.1 General Notations

Sets Let A and B be two sets, $A \cup B$ denotes the union of A with B, $A \cap B$ denotes the intersection of A with B, $A - B$ denotes the subtraction of B to A, and $A \times B$ denotes the cartesian product of A with B. $A \subset B$ holds if A is strictly included in B, $A \subseteq B$ holds if A either $A \subset B$ or $A = B$, and $a \in A$ holds if a is an element of A. The set of all subsets of A is denoted by $\mathcal{P}(A) = \{S' \mid S' \subseteq S\}$. If A is a finite set, $\| A \|$ denotes its cardinal. Let $a, b \in \mathbb{N}$, $\{a \dots b\}$ denotes the set $\{i \mid i \in \mathbb{N} \wedge a \leq i \leq b\}$. Similarly, $\{s_a \dots s_b\}$ denotes the set $\{s_i \mid a \leq i \leq b\}$.

Words Let Σ be a set of letters. The set of all words over Σ is written Σ^* and is the minimal set such that:

$$\begin{aligned} \epsilon &\in \Sigma^*, \text{where } \epsilon \text{ is the empty word} \\ s &\in \Sigma^*, \text{for } s \in \Sigma \\ \overline{w}_1\overline{w}_2 &\in \Sigma^*, \text{for } \overline{w_1}, \overline{w_2} \in \Sigma^* \end{aligned}$$

Let $\overline{w} \in \Sigma^*$ be a word over Σ. We write $s \in \overline{w}$ if $s \in \Sigma$ appears at least once in $\overline{w}$. For example, assuming a set of symbols $\Sigma = \{a, b, c\}$ and a word $\overline{w} = aab$, then $a \in \overline{w}$ but $c \notin \overline{w}$. $\| \overline{w} \|$ denotes the cardinal of $\overline{w}$ (e.g. $\| aab \| = 3$), and $\| \overline{w} \|_s$

denotes the number of occurrences of s in $\overline{w}$ (e.g. $\| aab \|_a = 2$). The set of words over Σ without letter repetition is written $\Sigma^{\#}$, and is defined as follows:

$$\overline{w} \in \Sigma^{\#} \Leftrightarrow \overline{w} \in \Sigma^{*} \wedge \forall s \in \overline{w}, \| \overline{w} \|_s = 1$$

Let $\overline{w} \in \Sigma^{*}$ be a word over Σ, $\overline{w}_i$ represents the i-th letter of $\overline{w}$, for any $i \in 1 \ldots \| \overline{w} \|$ (e.g. $(aab)_3 = b$). Let $i, j \in \mathbb{N}$, $\overline{w}_{i,j}$ represents the substring from the i-th to the j-th letter (e.g. $(aab)_{2,3} = ab$). Let $\overline{w} \in \Sigma^{\#}$ be a word without repetition and $s \in \overline{w}$ a letter in $\overline{w}$, $\overline{w}[s]$ denotes the position of the letter s in $\overline{w}$, that is $\overline{w}[s] = i \Leftrightarrow \overline{w}_i = s$.

Functions Let f be a function. $\mathrm{dom}(f)$ and $\mathrm{codom}(f)$ denote its domain and codomain, respectively. The notation $f : A \to B$ denotes a total function from a domain A to B, that is $\mathrm{dom}(f) = A$ and $\mathrm{codom}(f) = B$. A *partial* function $g : A \nrightarrow B$ is characterized by $\mathrm{dom}(g) \subseteq A$.

3.2 Data Types

There is a gap between the classic set-theoretic mathematical notations and that commonly used in programming languages. As this work is essentially a study of the latter through the lens of the formal mathematical instrument, it is only natural to aim at bridging this gap.

Composite Types

One of the most basic kinds of data structure in programming is the *composite type*. A composite type denotes any type that is a composition of other types, such as, for example, a pair of integers. The closest set-theoretic equivalent of a composite type is the cartesian product. However, one cannot access a particular element of the pair without "binding" each component to some variable with this approach. For example, let p be a pair, then expressing a constraint on its second element can only be achieved by matching p with a pair of variables. More formally, one is compelled to write $p = \langle x, y \rangle \wedge x > 8$, whereas most programming languages allow for a far a concise notation, e.g. `p.second > 8`. One could argue for the use of indices, and write p_i for the i-th element of p. Unfortunately, this approach does not scale well with large tuples. Another alternative is to use subscripts, which resembles the typical syntax of object-oriented programming languages. Unfortunately, such notation is in fact impractical when dealing with chains of aggregate objects. For instance, let t denote the root of a binary tree, then referring to a grandchild of t using subscripts becomes barely legible. We choose a third alternative, and represent composite types as families of functions, whose domain represent labels and whose codomain represent the values.

Definition 3.2.1: Composite Type

Let L be a finite set of labels, and $d : L \to D$ be a function that returns the domain of a label $l \in L$. A composite type is a family of functions $F_{L,d} : L \to \bigcup_{l \in L} d(l)$.

Notation: Composite Type

Let $L = \{l_1 \dots l_n\}$ be a set of labels and $D = \{D_1 \dots D_n\}$ be a set of domains. We use the notation $\langle l_1 : D_1 \dots l_n : D_n \rangle$ to denote the composite type $F_{L,d}$ where d is defined such that $\forall i \in \{1 \dots n\}, d(l_i) = D_i$.

Example: *The set of pairs of integers is defined by a composite type:*

$$\langle \text{first} : \mathbb{Z}, \text{second} : \mathbb{Z} \rangle$$

Put differently, the set of pairs of integers is a family of functions $F_{L,d}$ where $L = \{\text{first}, \text{second}\}$, and d is defined such that $d(\text{first}) = d(\text{second}) = \mathbb{Z}$.

Just as a cartesian product denotes a set in which one may pick a particular tuple, a composite type denotes a set of functions in which one may pick a particular function, that we call a *record*.

Definition 3.2.2: Record

Let L be a finite set of labels, and $d : L \to D$ be a function that returns the domain of a label $l \in L$. A record is an instance of a composite type, that is a member of the family of functions $F_{L,d}$.

Notation: Record

When the domain of each label is either known, obvious or irrelevant, we write $\langle l_1 \mapsto v_1 \dots l_n \mapsto v_n \rangle$ the record of a composite type $F_{L,d}$ where $L = \{l_1 \dots l_n\}$ and $v_1 \in d(l_1) \dots v_n \in d(l_n)$.

Example: *Let p be a record of $\langle \text{first} : \mathbb{Z}, \text{second} : \mathbb{Z} \rangle$, defined as follows:*

$$p = \langle \text{first} \mapsto 42, \text{second} \mapsto 1337 \rangle$$

In other words, p is a pair whose elements are 42 and 1337.

Notation: Dot-notation

We borrow the so-called "dot-notation" from programming languages and write $a.b$ rather than $a(b)$. The dot is left associative, i.e. $a.b.c = (a.b).c = (a(b))(c)$. Furthermore, we call $a.b$ a *field* of a.

Example: *Let p be a pair defined by the following record:*

$$p = \langle \text{first} \mapsto 42, \text{second} \mapsto 1337 \rangle$$

*Then $p.$*second *denotes the second field of p, that is $p.$*second $= 1337$.

Notation: Update by extension

Let $r : L \to D$ be a record. Let $l_1 \dots l_n \subseteq L$ and $v_1 \dots v_n \subseteq D$. We write $r' = r[l_1 \mapsto v_1 \dots l_n \mapsto v_n]$ the *update* of r, that is the record that maps l_i to v_i but m to $r.m$ for any other $m \in L$. More formally:

$$\forall m \in L, r'.m = \begin{cases} v_i & \text{if } \exists i, m = l_i \\ r.m & \text{otherwise} \end{cases}$$

Example: *Let $p = \langle x \mapsto 42, y \mapsto 1337 \rangle$ be a pair of integers. $p' = p[x \mapsto p.x + 2]$ is the pair p whose first field is incremented by two, that is $p'.x = p.x + 2 = 44$ and $p'.y = p.y = 1337$.*

Notation: Update by intension

Let $r : L \to D$ be a record. We sometimes define updates by intension rather than extension and write $r[l \mapsto v \mid p(l)]$, which reads as r updated for each label l satisfying a predicate p.

Example: *Let $v = \langle x \mapsto 2, y \mapsto 4, z \mapsto 8 \rangle$ be a 3-dimensional vector, represented by the coordinates of its tip and assuming its tail is at the origin. The update $v' = v[a \mapsto a * 2 \mid a \in \text{dom}(v)]$ is the vector v scaled by 2:*

$$v'.x = v.x * 2 = 4 \text{ and } v'.y = v.y * 2 = 8 \text{ and } v'.z = v.z * 2 = 16$$

Notation: Update syntactic sugar

Let $r : L \to D$ be a record. Let $l_1 \dots l_n \subseteq L$ and $v_1 \dots v_n \subseteq D$. When updating multiple fields with the same value, we usually write $r[l_1 \dots l_n \mapsto v]$ short for $r[l_1 \mapsto v \dots l_n \mapsto v]$.

Example: *Let $v = \langle x \mapsto 2, y \mapsto 4, z \mapsto 8 \rangle$ be a 3-dimensional vector, represented by the coordinates of its tip and assuming its tail is at the origin. The update $v' = v[x, y \mapsto 0]$ is the vector projection of v onto the z-axis:*

$$v'.x = v'.y = 0 \text{ and } v'.z = v.z = 8$$

Notation: Deep update

Let a be a record that contains a field $a.b$ that is a record $L \to D$. Let $l_1 \dots l_n \subseteq L$ and $v_1 \dots v_n \subseteq D$. We write $a[b.l_1 \mapsto v_1 \dots b.l_n \mapsto v_n]$ short for $a[b \mapsto a.b[l_1 \mapsto v_1 \dots l_n \mapsto v_n]]$.

Example: *Let T denote the set of linked lists of integers, inductively defined as the minimal set such that:*

> *$\epsilon \in T$ is an empty list.*
>
> *$\{h : \mathbb{Z}, t : T\} \subseteq T$ is the set of linked lists headed by an integer and tailed by another linked list.*

Let $l = \langle h \mapsto 1, t \mapsto \langle h \mapsto 2, t \mapsto \langle h \mapsto 4, t \mapsto \epsilon \rangle \rangle \rangle$ be a record representing the sequence $1, 2, 4$ as a linked list. $l' = l[t.h \mapsto 3]$ is the update that only modifies the second element of l, so that l' represents the sequence $1, 3, 4$.

Table Types

Some data structure may behave similarly to a record, but not feature a fixed domain. For instance, a bank ledger may be represented as a record mapping names and dates onto transaction details. However, unlike a record, its domain shall grow as more transactions are registered. We call such structure a *table*[1] in the remainder of this thesis. Just as composite types, we use functions to represent instances of table types. Their labels do not have to belong to a finite set, but should share a single domain. An instance of a table type is simply called a table.

Definition 3.2.3: Table Type

Let L be a countable set of labels and V a set of values. A table type is a family of partial functions $F_L : L \nrightarrow V$.

[1] The terms *dictionary*, *map* and *associative array* are sometimes used in programming languages and literature to represent the same object.

Notation: Table

When the set of labels and their domain are either known, obvious or irrelevant, we write $t = \{l_1 \mapsto v_1 \dots l_n \mapsto v_n\}$ the table t: $L \twoheadrightarrow V$ where $\{l_1 \dots l_n\} \subseteq L$ and $\{v_1 \dots v_n\} \subseteq V$. We write $\text{dom}(t)$ the set $L' \subseteq L$ for which t is defined, and write $\varnothing$ to denote an empty table t such that $\text{dom}(t) = \varnothing$.

Example: *The following table is a log that maps dates, represented as character strings, onto temperatures:*

$$g = \{\texttt{"19/08"} \mapsto 24, \texttt{"08/11"} \mapsto 19\}$$

Notation: Table updates and insertions

We extend the notation for updates and write $t' = t[l_1 \mapsto v_1 \dots l_n \mapsto v_n]$ an update of t. However, since the domain of a table is not fixed, $\text{dom}(t')$ may differ from $\text{dom}(t)$. More formally:

$$\text{dom}(t') = \text{dom}(t) \cup \{l_1 \dots l_n\} \land \forall m \in \text{dom}(t'), t'.m = \begin{cases} v_i & \text{if } \exists i, m = l_i \\ t.m & \text{otherwise} \end{cases}$$

Notation: Table removals

We use the notation $t|_{l_1 \dots l_n}$ to *remove* $\{l_1 \dots l_n\}$ from the domain of t. In other words, it reduces the size of the domain of t. More formally, let $t' = t|_{l_1 \dots l_n}$, then:

$$\text{dom}(t') = \text{dom}(t) - \{l_1 \dots l_n\} \land \forall m \in \text{dom}(t'), t'.m = t.m$$

Example: *Let* $g = \{\texttt{"19/08"} \mapsto 24, \texttt{"08/11"} \mapsto 19\}$ *be a temperature log.*

$$g' = g|_{\texttt{"19/08"}} = \{\texttt{"08/11"} \mapsto 19\}$$

is an update of g that removes the entry for the 19th of August.

Notation: Table merges

Let t_1 and t_2 be two tables $L \twoheadrightarrow D$. We write $t_1 \uplus t_2$ the merging of the table t_2 into t_1, formally given by:

$$t_1 \uplus t_2 = t_1[x \mapsto t_2.x \mid x \in \text{dom}(t_2)]$$

3.3 Inference Rules

We define operational semantics and type systems with inference rules. An inference rule is a logical statement of the form:

$$\frac{p_1 \; p_2 \; \cdots \; p_n}{q}$$

$p_1, p_2, \ldots, p_n$ constitute the rule's *premise*, and is composed of *n hypotheses*. q denotes the rule's *conclusion*. Together, they read as "We can conclude q if all $p_1, p_n, \ldots, p_n$ hold". In other words, it states that $(p_1 \wedge \cdots \wedge p_n) \implies q$, where $p_1 \ldots p_n, q$ are predicate terms (e.g. $a > b$). A rule that has no hypotheses is an *axiom*.

We use inference rules to build formal proofs with a process called resolution. Given a set of inference rules, resolution is a recursive process that consists in attempting to satisfy a proof obligation by first finding an inference rule that concludes it, and then proving that this inference rule's hypotheses hold. The recursion terminates when an axiom is found, or when the hypotheses are not satisfiable. We represent this process with a graphical notation that depicts recursive applications on top of one another.

Example 3.3.1: Sums

Consider the following inference rules, describing the operational semantics of a simple language to compute sums:

$$\frac{e \in \mathbb{Z}}{e \Downarrow e}\ \textsc{Lit} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{\mathsf{add}(e_1, e_2) \Downarrow n_1 + n_2}\ \textsc{Add}$$

Computing the evaluation of an expression $\mathsf{add}(\mathsf{add}(2,3),4)$ *under this semantics consists in proving a conclusion of the form* $\mathsf{add}(\mathsf{add}(2,3),4) \Downarrow n$*, where n is the result to deduce, as done in the following proof derivation:*

$$\dfrac{\dfrac{\dfrac{\dfrac{2 \in \mathbb{Z}}{2 \Downarrow 2}\ \textsc{Lit} \qquad \dfrac{3 \in \mathbb{Z}}{3 \Downarrow 3}\ \textsc{Lit}}{\mathsf{add}(2,3) \Downarrow 5}\ \textsc{Add} \qquad \dfrac{4 \in \mathbb{Z}}{4 \Downarrow 4}\ \textsc{Lit}}{\mathsf{add}(\mathsf{add}(2,3),4) \Downarrow 9}\ \textsc{Add}$$

The derivation eventually shows that $n = 9$*, which correspond to the evaluated value.*

We now formally describe this resolution process. Let X denote a set of variables. Let H_X denote the set of Horn clauses [80] over the variables in X. Let $t \in H_X$ be a term, $t[^x/_u]$ denotes the term t in which occurrences of x are substituted with u (e.g. $(a \wedge b)[^a/_c] \equiv c \wedge b$). Let $\sigma : X \twoheadrightarrow H_X$ be a substitution table that maps variables onto terms, we write $t[/\sigma]$ short for $t[^{x_1}/_{\sigma.t_1}] \dots [^{x_n}/_{\sigma.t_n}]$ for all $x_i \in \mathrm{dom}(\sigma)$ (e.g. $(a \wedge b)[/\{a \mapsto c, b \mapsto d\}] \equiv c \wedge d$). Let $T \subseteq H_X$ be a set of terms, we write $T[^x/_u]$ (resp. $T[/\sigma]$) short for $\{t[^x/_u] \mid t \in T\}$ (resp. $\{t[/\sigma] \mid t \in T\}$). Let $t \in H_X$ be a term, $\mathrm{vars}(t)$ denote the variables in t (e.g. $\mathrm{vars}(a \wedge b) = \{a, b\}$).

Let I be a set of inference rules. Let $Q \subseteq H_X$ be a set of proof obligations. A single resolution step consists in eliminating a proof obligation $q \in Q$ by finding an inference rule $(P \implies q') \in I$ such that q' is unifiable (i.e. equivalent up to substitutions) with q, to produce a new set of proof obligations Q' extended with the hypotheses in P.

Definition 3.3.1: Resolution rule

Let X be a set of variables and H_X a set of Horn clauses over X. Let $I \subseteq \mathcal{P}(H_X) \times H_X$ be a set of inference rules of the form $\{p_1 \dots p_n\} \implies q$. Let $Q \subseteq H_X$ be a set of proof obligations and $\sigma : X \twoheadrightarrow H_X$ a substitution table. The resolution rule is given by the following statement:

$$\frac{(C \implies q') \in I \qquad \exists \sigma', q[\sigma'] \equiv q'[\sigma']}{\{q\} \cup Q, \sigma \longrightarrow Q[\sigma'] \cup C[\sigma'], \sigma \uplus \sigma'}$$

where $\mathrm{vars}(Q) \cap \mathrm{vars}(C \implies q') = \varnothing$.

Example 3.3.2: Resolution rule's application

Consider the inference rules presented in Example 3.3.1. Let us apply the resolution's transitive closure $\longrightarrow^*$ on a term $\mathsf{add}(1,2) \Downarrow n$. *We start with the set of proof obligations $Q = \{\mathsf{add}(1,2) \Downarrow n\}$ and an empty substitution table $\sigma = \varnothing$. The first step is computed using* Add*:*

$$\{\mathsf{add}(1,2) \Downarrow n\}, \varnothing \longrightarrow \{1 \Downarrow n_1, 2 \Downarrow n_2\}, \{e_1 \mapsto 1, e_2 \mapsto 2, n \mapsto n_1 + n_2\}$$

Next, $1 \Downarrow n_1$ is proven with Lit*:*

$$\longrightarrow \{1 \in \mathbb{Z}, 2 \Downarrow n_2\}, \{e_1 \mapsto 1, e_2 \mapsto 2, n \mapsto n_1 + n_2, e \mapsto 1, n_1 \mapsto 1\}$$

We proceed with a second application of Lit *to prove $2 \Downarrow n_2$:*

$$\longrightarrow \{1 \in \mathbb{Z}, 2 \in \mathbb{Z}\}, \{e_1 \mapsto 1, \dots, e' \mapsto 2, n_2 \mapsto 2\}$$

The hypotheses $1 \in \mathbb{Z}$ *and* $2 \in \mathbb{Z}$ *are trivially proven without hypotheses, so the set of proof obligations is emptied:*

$$\longrightarrow^* \varnothing, \{e_1 \mapsto 1, e_2 \mapsto 2, n \mapsto n_1 + n_2, e \mapsto 1, n_1 \mapsto 1, e' \mapsto 2, n_2 \mapsto 2\}$$

The process terminates because there are no more proof obligations, and the substitution table indicates that $n = n_1 + n_2 = 1 + 2 = 3$.

A set of inference rules can have different properties that may impact resolution. A system is *deterministic* if for each hypothesis there is only one whose conclusion is unifiable. For instance, the inferences rules presented in Example 3.3.1 form a deterministic system, because only one rule can be applied to evaluate a particular term. A non-deterministic system does not necessarily lead to different results, if all choices eventually produce the same conclusion. In this case, it is said to be *confluent*. Finally, note that resolution does not necessarily *terminate*. For instance, a system could contain a rule whose hypotheses recursively require to prove the rule's conclusion, and thereby resolution could fail ever emptying the set of proof obligations.

Example 3.3.3: Confluent non-deterministic terminating system

Consider the following inference rules, describing a system that evaluates words:

$$\textsc{Prefix}\ \frac{\overline{w} \Downarrow \overline{w}'}{s\overline{w} \Downarrow \overline{w}'} \qquad \textsc{Postfix}\ \frac{\overline{w} \Downarrow \overline{w}'}{\overline{w}s \Downarrow \overline{w}'} \qquad \textsc{Empty}\ \frac{}{\epsilon \Downarrow \epsilon}$$

Both rules have different conclusions. Prefix *matches a letter prefixing an arbitrary word, whereas* Postfix *matches a arbitrary word suffixed by a letter. However, both rules are unifiable. Accordingly, although both rules can be used to prove* $\overline{w} \Downarrow \overline{w}'$*, the result of* $\overline{w}'$ *is necessarily* ϵ *for any word* $\overline{w}$*. The system is non-deterministic, but confluent and terminating.*

Chapter 4

The $\mathcal{A}$-Calculus

Inside every large language is a small language struggling to get out...

Atsushi Igarashi, Benjamin Pierce, Philip Wadler [81]

Imperative programming languages describe computation as sequences of statements whose execution modifies the *state* of a program. This state is most commonly defined by the content of the machine or interpreter's memory running the program, and its evolution is is driven by *assignments*, which are particular statements that write (i.e. assign) some value to a specific location in the memory. Hence, all imperative languages feature at least one way to perform these commands, usually through instructions of the form "$l := e$", where l represents some memory location, and e the expression of some value. As mentioned in the introduction, the precise semantics of the ":=" operator may vary from one language to the other. While this can already represent a challenge for learning developers, and hinders one's knowledge transfer, the fact that semantics may also vary within a single language is even more worrisome. Indeed, in an effort to abstract over the complex intricacies of memory management, contemporary programming languages tend to overload a single assignment operator with different meanings, depending on the type of its left and/or right operands. While adequately conveying the developer's intent in most situations, this abstraction usually ends up leaking implementation details in several edge cases, leading to sound yet confusing behaviors for both novices and experts alike. In particular, whether or not an assignment creates an alias can prove difficult to evaluate.

In this chapter, we propose to take a deep look into assignment semantics, by the means of a language calculus that accurately models the different approaches

to assignment. Its most distinguishing feature lies in the use of multiple unequivocal operators, which dispel the ambiguities found in most contemporary programming languages. Hopefully, this results in programs whose semantics is easier to reason about. We first motivate the need for such a theoretical model, before presenting it informally and formally.

4.1 Problematic

Most contemporary programming languages abstract over the notion of pointers. Thus, variables can be understood as the value they ultimately represent, without concerns for *how* they are represented, nor *where* they are stored. The advantage of this approach is immediate and unequivocal: it allows reasoning in terms of abstract data types. The use of abstract data structures is a natural evolution. For instance, manipulating a single object representing a matrix of 2D points is certainly preferable to dealing explicitly with a pointer to a contiguous range of memory locations, each holding a pointer to 128 bits long structures. However this introduces a non-trivial challenge as to how to deal with storage. Indeed, whereas data structures may represent very abstract concepts such as lists, trees and tables, computer memory is generally always organized in terms of stack and heap, inheriting all the intricacies discussed Section 2.1. A simple way to overcome this issue is to primarily allocate objects on the heap and rely on some form of automatic garbage collection (see Section 2.1.2). We call the languages that adopt this strategy *reference-based languages* throughout this thesis, since almost all variables actually represent references (i.e. pointers) to heap addresses. Examples of such languages include Java, Python and Smalltalk. Conversely, we say that a language is *stack-based*[1] if its notion of variable is more tightly associated with the call stack, and pointers (if supported) are manipulated explicitly. Examples include C, Rust and Ada.

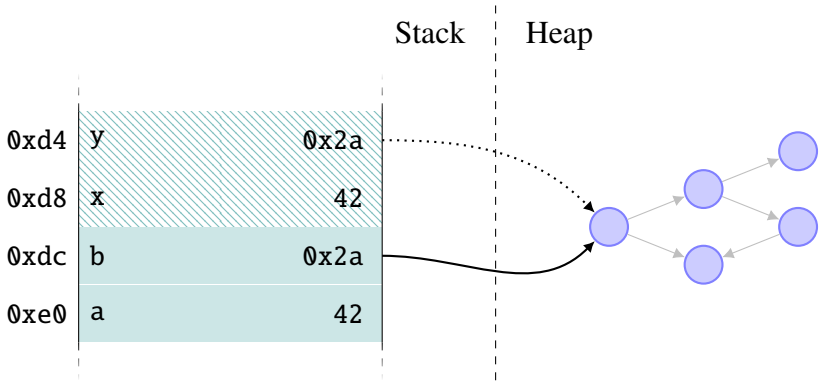
Reference-based languages alleviate the burden of memory management. However, the abstraction often leaks implementation details, leading to confusing assignment semantics. For instance, many runtime systems store values of primitive types like numbers and booleans directly on the call stack, mainly for performance reasons. Furthermore, they usually implement assignments as a straightforward “bitwise copy" of the right operand, which is typically available as a single CPU instruction and thus rather fast. Unfortunately, this strategy can introduce subtle differences between values stored on the stack and those stored on the heap. More precisely, the former gets actually copied, whereas the latter only see the value of their pointer copied. Consequently, assignments whose right operands are in fact

[1] Not to be confused with *stack-oriented programming languages*, in which a program is expressed directly in terms of stack operations.

pointers to heap-allocated memory effectively result in the creation of an alias, which can easily go unnoticed and cause unintended sharing of state issues, as demonstrated in Section 1.1. An example follows:

Example 4.1.1: Effect of bitwise copy assignment

Consider the following figure, depicting a situation where two local variables `x` *and* `y` *(represented by hashed cells) are being assigned to the value of two other variables* `a` *and* `b` *(represented by plain cells), respectively.*



Variable `a` *holds a machine size integer, i.e. a 32-bits sequence in this particular example, stored at the stack address* `0xdc`*. The variable being used as the right operand of a bitwise copy assignment, that 32-bits sequence is copied, as is, and stored at* `0xd8`*, which represents variable* `x`*. As a result,* `a` *and* `x` *are now two distinct variables holding two distinct values. Variable* `b` *on the other hand holds a pointer to a heap address, represented here as a 32-bits sequence stored at the stack address* `0xdc`*. Just like the assignment of* `x`*, this 32-bits sequence is copied, as is, and stored at* `0xd4`*, which represents variable* `y`*. However, the copy is applied on the pointer's value, while the object on the heap is completely ignored. As a result,* `b` *and* `y` *are now two distinct variables, but whose values actually point to the same object in the heap. In other words,* `y` *is effectively an alias on* `b`*.*

At the language level, seemingly identical operations may have varying semantics. For instance in Python, using an augmented assignment statement (e.g. `a += b`) or expanding the statement (e.g. `a = a + b`) is semantically equivalent for the reference being assigned, but has different side effects, as illustrated in the this example:

Example 4.1.2: Inconsistent assignment semantics

`foo` is aliased at line 2 by a variable `bar`. Its value is mutated at line 3, using the augmented assignment `+=`, `bar` can also observe this change.

```
1 foo = [1, 2]
2 bar = foo
3 foo += [3]
4 print(bar)
5 # Prints "[1, 2, 3]"
```

Here, `foo` is reassigned at line 3, and therefore `bar` can no longer observe this change.

```
1 foo = [1, 2]
2 bar = foo
3 foo = foo + [3]
4 print(bar)
5 # Prints "[1, 2]"
```

Purposely manipulating references can also prove problematic. Indeed, if assigning to a variable holding a pointer necessarily rebinds said variable to another object, we run the risk of unintentionally breaking a chain of references. The workaround is then to rely only on self-modifying methods, or to wrap objects within containers. The latter essentially boils down to emulating pointers, the very concept being abstracted away. The problem is exacerbated by the fact that programming languages commonly overload a single assignment operator, whose semantics consequently depends on the type of its operands. For instance, Swift has distinct assignment semantics for what it calls *reference* (e.g. classes) and *value* (e.g. structures) types. The former follows Java's semantics, while assignments of value types create copies, as illustrated in the example below. Move semantics can add yet another source of confusion in languages that, like (modern[2]) C++ or Rust, support move semantics along with other assignment policies.

Example 4.1.3: Implicit assignment semantics in Swift.

Applied on reference types, assignments result in the creation of aliases.

```
1 class C { var m: Int }
2 var x = C(m: 15)
3 var y = x
4 x.m = 20
5 print(y.m)
6 // Prints "20"
```

[2]Move semantics were introduced in C++11.

On value types, assignments result in an actual copy.

```
struct S { var m: Int }
var x = S(m: 15)
var y = x
x.m = 20
print(y.m)
// Prints "15"
```

Overloading a single assignment operator can also introduce additional issues due to interactions with other language features. Swift for example features a powerful system of interfaces that lets a developer design specifications that can later be implemented by concrete types. Furthermore, the language allows function signatures to be defined over interfaces and concrete types alike, unlocking a lot of potential for genericity. However, as an interface can be implemented by both value and reference types, there is no way to determine the semantics of an assignment involving an argument typed with one.

Example 4.1.4: Obfuscated assignment semantics

Consider the following program:

Swift

```
protocol Fooable {
  var bar: Int { get set }
}
func f(a: Fooable) -> Fooable {
  var b = a
  b.bar = 20
  return b
}
```

`Fooable` *is a protocol (the equivalent of e.g. a Java Interface) that specifies a member* `bar`*. The function* `f` *expects a value of any type conforming to this protocol as parameter. If it is called with an argument of a value type, then the assignment at line 5 is a copy and* `f` *is pure. On the other hand, if it is called with an argument of a reference type, then the assignment at line 5 creates an alias and* `f` *has a side effect on its argument.*

These subtle semantic differences have far reaching consequences that are hard to understand for beginners, and can significantly steepen the learning curve, as pointers and references consistently appear as a difficult aspect of programming [132]. Furthermore, while seasoned developers may hold a more solid grasp on these notions, errors are likely to occur in refactoring. Note that the root of

the problem is not the abstraction of pointers itself. It lifts the burden of explicit dereferencing (i.e. the act of getting the value that is stored at the memory referenced by a pointer), and removes the need to distinguish between stack and heap allocations, which is arguably beneficial to the developer. The issue is that the abstraction leaks implementation details when dealing with assignments. Choosing one type of assignment over the others is not a good solution either, as different situations call for different semantics. Indeed, it makes sense in some cases to prefer mutation over reassignment, and vice versa. A good abstraction should handle both these scenarios consistently, for all kinds of values a program may manipulate, independently of where and how they are stored at the implementation level.

4.2 Informal Introduction

This section gives an overview of the $\mathcal{A}$-calculus. We start by introducing its multiple unequivocal assignment operators, and describe their semantics informally. We then move on to describe the relationship between these operators with function parameter passing. We illustrate our framework in the context of a hypothetical programming language, whose formal description is provided by the means of a minimal calculus in Section 4.3. An implementation of the $\mathcal{A}$-calculus is discussed in Chapter 7, in the form of the Anzen programming language.

4.2.1 Assignment Operators

Our framework features three assignment operators, with a clear and unambiguous semantics, independent of the type of their operands:

- An aliasing operator `&-` creates an alias to the object on its right for the variable on its left. Its semantics is the closest to what is generally understood as an assignment in reference-based languages.

- A mutation operator := mutates the object assigned to the variable on its left with a copy of the object assigned to the the variable on its right. The copying process is applied transitively to each of the source object's properties, so that no alias remains. Unlike the aliasing operators, this one mutates the value of its left operand – provided there was any – rather than (re)assigning the variable to a different memory location.

- A move operator ← moves the object from the variable on its right to the variable on its left, effectively treating the object affinely [135]. This operator is deeply intertwined with a notion of (external) uniqueness and ownership, and therefore assumes the moved object is not aliased.

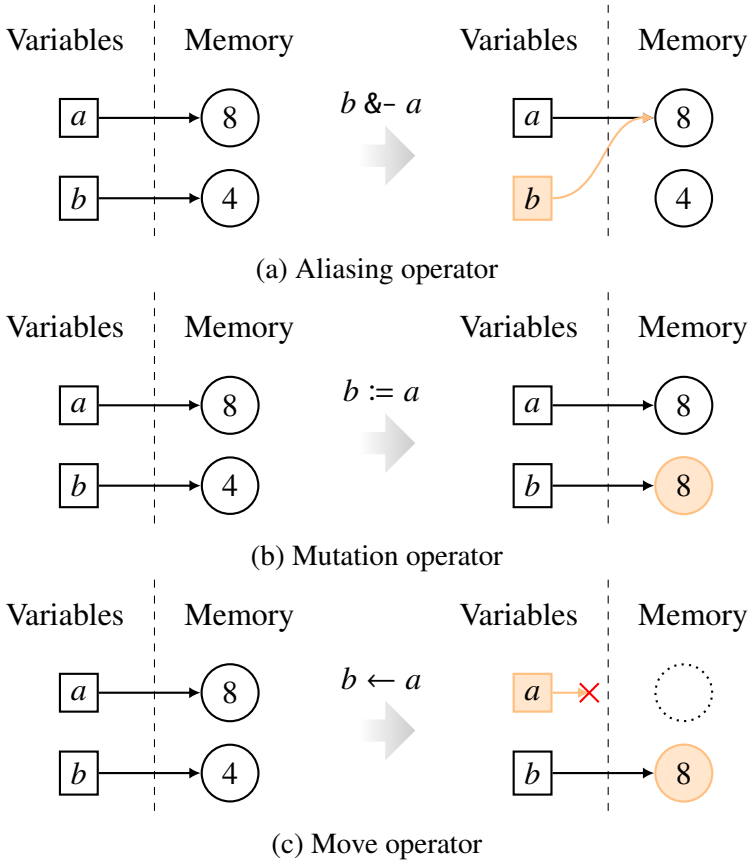


Figure 4.2.1: Effect of assignment operators. Each illustration depicts the situations before and after a particular assignment, starting from a state where two variables a and b are bound to unrelated memory locations, holding the values 8 and 4 respectively. Changes are highlighted in color.

Notice that we use the term "variable" as opposed to "pointer". Similarly, we use the term "memory" rather than "stack" or "heap". The reason is that we aim at making a clear distinction between the semantics of our operators and the actual memory model that is used to implement them. Whether a variable is a pointer to heap-allocated memory, or the address of some primitive type allocated on the stack should be irrelevant for the developer. That way she may focus solely on the semantics of her program, at an abstraction level that does not bother about the specifics of compilation, optimization and/or interpretation. Consequently, fields of a record can be understood as variables as well.

4.2.2 Parameter Passing

There is a compelling connection between variable assignment and the passing of parameters in a function call. In fact, just as most runtime systems implement assignments as a straightforward "bitwise copy", they often do the same for function arguments. In other words, those get a bitwise copy of the argument's stack value. This makes sense from an implementation perspective, as it preserves the symmetry with assignment semantics. Unfortunately, it may also cause the same confusing situations as those we described in Section 4.1, with an extra layer of complexity introduced by the transfer of control flow. We tackle the issue by reusing the three aforementioned assignment operators for function arguments. A noteworthy advantage of this approach is that it lets us explicitly define the intended passing semantics. The use of an aliasing operator corresponds to a *pass-by-alias*, while the use of a mutation operator corresponds to a *pass-by-value* policy [46]. Unsurprisingly, the move operator also treats objects as linear resources, and therefore mimics the affine semantics of Rust and C++.

In order to limit the syntactic overload of our example, we use parameters' names as the left operand of an assignment statement to represent an argument together with its passing semantics. For instance, we write $f(x \texttt{ \&- } a, y := b)$ the application of a function f with two parameters x and y, on two arguments a and b passed by alias and value, respectively. Similarly, we also reuse the assignment operators to describe how return values should be handled. The aliasing operator corresponds to a *return-by-alias*, the mutation operator corresponds to a *return-by-value*, and the move operator retains its linear semantics. An example follows:

Example 4.2.1: Explicit parameter passing semantics

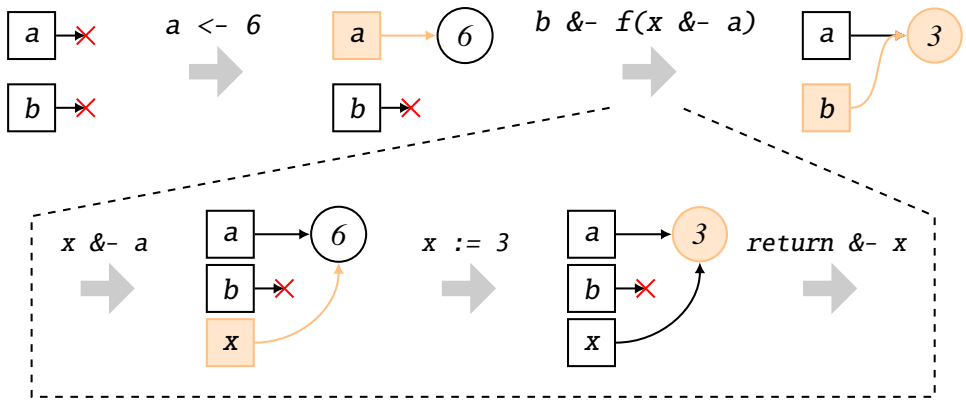
Consider the following program:

$\mathcal{A}$-calculus

```
1 fun f(x) {
2   x := 3
3   return &- x
4 }
5 let a <- 6
6 let b &- f(x &- a)
```

A first variable `a` *is declared and initialized at line 5. A second variable* `b` *is declared at the next line, and bound to an alias of the function* `f`*'s call result.* `f` *is called with an alias on* `a`*, assigned to its* `x` *parameter. It mutates its value at line 2, before returning it by reference at line 3. The figure below illustrates the links between variables and memory after*

each line of the program. Changes are highlighted in orange. The dashed container "zooms" into `f`*'s evaluation.*



The advantages of this approach are two-fold. First, instead of having to opt for one particular passing policy, we offer the developer to choose for herself the most appropriate one, and provide her with the means to do so explicitly. We should stress that languages that support a first-class notion of reference share a similar ability. However, our system is more powerful as we do not constrain the choice at the function declaration, so that a different passing policy can be applied at each function call without having to resort to overloading. Second, reusing the assignment operators makes the relationship between assignment and parameter passing explicit, which we believe to be beneficial for educational purposes. Even the induced syntactic overhead may in fact be seen as an improvement. Indeed, whereas it is traditional to identify parameters by their position, one can leverage named parameters to improve the legibility of the code. For instance, the expression `iter(from <- 0, by_steps_of <- 2)` is arguably more self-explanatory than `iter(0, 2)`. Such a peculiarity is often referred to as "named parameters".

4.3 Syntax and Semantics

This section formalizes the operational semantics of the $\mathcal{A}$-calculus. We borrow from the λ-calculus with assignment [111] to represent the underlying computational model, which we extend with specific terms related to the semantics we have presented informally in the previous section. As a result, a program is expressed by a single term $t \in \mathcal{AC}$, where $\mathcal{AC}$ is a set that describes the terms of a variant of the classic λ-calculus with assignment, extended with our assignment operators. We use the words "term" and "program" interchangeably in the remainder of this thesis. We make two assumptions on the memory model. First,

we consider memory as an infinite resource, and do not formalize allocation failures. Second and more importantly, we do not distinguish between heap and stack, and do not assume memory to ever be automatically garbage collected.

4.3.1 Abstract Syntax

We start by formally defining the abstract syntax of the $\mathcal{A}$-calculus.

Definition 4.3.1: Abstract syntax

Assume a set of atomic literals, written $\mathbb{A}$, typically denoting instances of primitive data types (e.g. numbers and booleans). Assume a countable set of identifiers, written $\mathbb{X}$, denoting valid variable identifiers. Let $\mathbb{O} = \{ \&\text{-} , \leftarrow , := \}$ denote the set of assignment operators. The set of terms $\mathcal{AC}$ is defined recursively as the minimal set such that:

$$
\begin{array}{rcll}
a \in \mathbb{A} & \Longrightarrow & a \in \mathcal{AC} & \textit{an atom} \\
x \in \mathbb{X} & \Longrightarrow & x \in \mathcal{AC} & \textit{a variable} \\
x \in \mathbb{X} \wedge t \in \mathcal{AC} & \Longrightarrow & \texttt{let}\ x\ \texttt{in}\ t \in \mathcal{AC} & \textit{a declaration} \\
x \in \mathbb{X} & \Longrightarrow & \texttt{alloc}\ x \in \mathcal{AC} & \textit{an allocation} \\
t \in \mathcal{AC} & \Longrightarrow & \texttt{del}\ t \in \mathcal{AC} & \textit{a deallocation} \\
\overline{x} \in \mathbb{X}^{\#} \wedge t \in \mathcal{AC} & \Longrightarrow & \lambda \overline{x} \triangleright t \in \mathcal{AC} & \textit{a function} \\
\diamond \in \mathbb{O} \wedge x \in \mathbb{X} \wedge t \in \mathcal{AC} & \Longrightarrow & x \diamond t \in \mathcal{AC} & \textit{an assignment} \\
t \in \mathcal{AC} \wedge \overline{u} \in (\mathbb{X} \times \mathbb{O} \times \mathcal{AC})^{\#} & \Longrightarrow & t(\overline{u}) \in \mathcal{AC} & \textit{an application} \\
\diamond \in \mathbb{O} \wedge t \in \mathcal{AC} & \Longrightarrow & \texttt{ret}\ \diamond t \in \mathcal{AC} & \textit{a return} \\
t_1, t_2 \in \mathcal{AC} & \Longrightarrow & t_1\ ;\ t_2 \in \mathcal{AC} & \textit{a sequence}
\end{array}
$$

Although parameter and argument lists are represented by words, we sometimes use commas to better distinguish between each individual subterm. For instance, we write $\lambda x, y \triangleright t$ rather than $\lambda xy \triangleright t$. All functions are anonymous but can be assigned to variables. For instance we write $\texttt{let}\ f\ \texttt{in}\ \texttt{alloc}\ f\ ;\ f \leftarrow \lambda x, z \triangleright \ldots$ to bind the variable f to a function. Consequently, recursion must be expressed by the means of the Y combinator. Scopes are delimited implicitly. We sometimes use indentation to better visualize logical containment in the abstract syntax, although this is a purely stylistic choice that does not otherwise impact neither the syntax nor the semantics of a term. Variables cannot be declared and bound to a value at once. Instead, variable declarations merely introduce a name into a new scope, in which allocation and initialization must be carried out. For instance, $\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \leftarrow 2$ denotes the declaration, allocation and initialization

of a variable. Splitting declaration, allocation and initialization allows to accurately model languages that do not necessarily couple them (e.g. C). In addition, it also makes the delimitation of lexical scopes extremely precise, as the order in which variables go in and out of scope is conveyed unambiguously. Figure 4.3.2 illustrates this idea. As the scope in which y is defined is shorter than that of x, the function call at the end refers to an undefined variable. Functions also delimit scopes, as the lifetime of an argument name is bound to that of their body. In the interest of preserving a symmetry between function parameters and variable declarations, the order in which arguments go out of scope is inverse to the order in which parameters are declared. For instance, argument x goes out of scope before y in the function $\lambda x, y \triangleright t$. For the sake of legibility and when the context allows it, we sometimes write `let` $x_1 \dots x_n$ `in` t short for `let` x_1 `in` $\dots$ `let` x_n `in` t. Similarly, we sometimes write `alloc` $x_1 \dots x_n$ as short for `alloc` $x_1 \dots$ `alloc` x_1. These are purely syntactic sugars and as such do not have any impact on the semantics. We use parenthesis liberally to delimit scopes and clarify precedence. Otherwise, sequences of terms are always assumed to be declared in the innermost scope, for instance both assignments are enclosed in the function's body in $\lambda x \triangleright x \leftarrow 2; x \leftarrow 3$, and function application is left associative, for instance $f(x \leftarrow 1)(y \leftarrow 2)(z \leftarrow 3) = (f(x \leftarrow 1)(y \leftarrow 2))(z \leftarrow 3)$. In contrast, the sequence operator ; is right associative, that is $t_1 \ ; \ t_2 \ ; \ t_3 = t_1 \ ; \ (t_2 \ ; \ t_3)$.

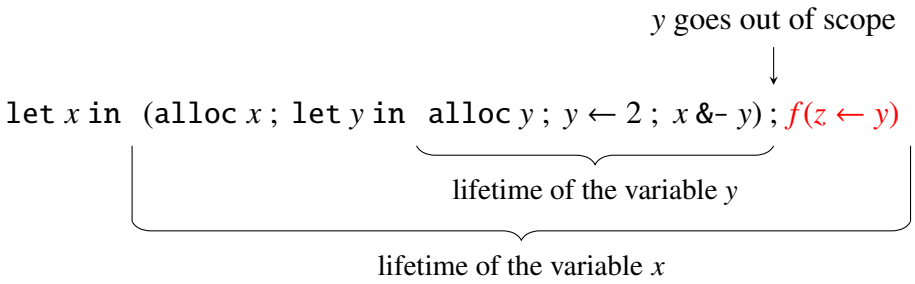


Figure 4.3.2: Example of lexical scoping.

We bring the reader's attention on the fact that return expressions are made explicit. This is a departure from most variants of λ-calculi (and functional languages in general), in which return values typically correspond the evaluation of a function's body. We cannot take the same direction, because our syntax needs to make the return policy explicit, by the means of our assignment operators. Furthermore, although a sequences of return statements are not syntactically ill-formed, the actual return value of a function is defined by the last return statement it executes. For instance, the function $\lambda \triangleright$ `ret` $\leftarrow 1$; `ret` $\leftarrow 2$ always returns 2 by move. In other words, a function never "returns early" and instead ignores non-final return statements.

4.3.2 Operational Semantics

Unlike most variants of λ-calculi, the $\mathcal{A}$-calculus has to factor the presence of memory into the evaluation of a term, because it has to distinguish between values and references thereupon. As a result, expressions are in effect evaluated as references (i.e. memory locations) rather than values, which is purposely reminiscent to the way variables are typically handled in the runtime systems of reference-based languages. We use the set of natural numbers $\mathbb{N}$ to denote memory locations. So as to avoid formalizing value representations beyond the strict necessary, with respect to assignment semantics, we neglect data sizes and we assume that a location $l \in \mathbb{N}$ can hold an arbitrarily large value. Recall also that we assume an unbounded memory and therefore do not cap the number of memory locations a program may handle at once. We reserve $0 \in \mathbb{N}$ to represent the null pointer.

We define the set of semantic values $\mathbb{V}$ that describes the data a program may manipulate. This set includes atomic values and function objects. We ignore higher-order functions for the moment, as their formalization introduces complex challenges in terms of value and reference captures. Instead, while we do treat functions as first-class citizens, but do not let them refer to their instantiation context. Simply put, we do not support closures, and force all identifiers referred to in function bodies to be either parameters or variables declared in the same function. An important consequence of this restriction is that our language does not support partial application. A workaround is proposed at the end of this chapter. We also add $\perp$ and $\square$ to represent *undefined* and *moved* values, respectively.

Definition 4.3.2: Semantic values

The set of semantic values $\mathbb{V}$ is defined as follows:

$$\begin{aligned} \perp &\in \mathbb{V} && \textit{undefined value} \\ \square &\in \mathbb{V} && \textit{moved value} \\ a \in \mathbb{A} &\implies a \in \mathbb{V} && \textit{an atom} \\ \overline{x} \in \mathbb{X}^{\#} \wedge t \in \mathcal{AC} &\implies \lambda \overline{x} \triangleright t \in \mathbb{V} && \textit{a function} \end{aligned}$$

Given $\mathbb{V}$ the set of semantic values, we define a table $\mu : \mathbb{N} \twoheadrightarrow \mathbb{V}$ to represent the memory of the executing machine. In other words, μ describes what value is stored at a given location. We abstract over data sizes and assume instead that any arbitrary large value can be stored at any given location. Besides, we also abstract over *where* the memory is allocated, that is we do not discriminate between stack and heap. We define another table $\pi : \mathbb{X} \twoheadrightarrow \mathbb{N}$ to represent variables. π describes what memory location is assigned to a given variable. Consequently, all variables are actually pointers, and do not refer directly to a particular value. Both tables are used both to form an evaluation context:

Definition 4.3.3: Evaluation context

Given $\mathbb{V}$ the set of semantic values, an evaluation context C is record of a composite type $\langle \pi, \mu \rangle$, where:

$\pi : \mathbb{X} \rightharpoonup \mathbb{N}$ *is a table mapping identifiers to memory addresses, and*

$\mu : \mathbb{N} \rightharpoonup \mathbb{V}$ *is a table mapping memory addresses to values.*

Let C be a given context, we refer to $C.\pi$ as its *pointer table* and to $C.\mu$ as its *memory table*. We say that a context C is empty if both its pointer table and memory table are empty, that is $C = \langle \pi \mapsto \varnothing, \mu \mapsto \varnothing \rangle$. We often use a tabular representation to describe the content of an evaluation context. For example, the following matrix represents an evaluation context in which three variables x, y and z are defined in the pointer table. x and y are assigned to the memory locations l_1 and l_2 respectively, while z is an alias on l_1. The memory table is defined for l_1 and l_2, respectively holding the values v_a and v_b.

π	μ
$x \mapsto l_1$	$l_1 \mapsto v_a$
$y \mapsto l_2$	$l_2 \mapsto v_b$
$z \mapsto l_1$	

We can now give the operational semantics of the $\mathcal{A}$-calculus, by the means of big-step inference rules. All conclusions are of the form $C \vdash t \Downarrow l, C'$, where C and C' are the evaluation contexts before and after evaluating the term t, respectively, and $l \in \mathbb{N}$ is a memory address at which the value represented by the term t is stored. Note that this semantics does not consider memory or data safety. Approaches to guarantee these properties dynamically and statically will be discussed in Chapter 5 and Chapter 6. Evaluation of programs that use memory improperly (e.g. dereferencing the null pointer) is therefore not defined. Undefined identifiers and type errors (e.g. evaluating an application with a non-function callee) are not defined either.

Variable Declarations

As mentioned earlier, variable declarations do not assign anything to the identifier being declared. Instead, declarations only consist in defining a new identifier in the pointer table before evaluating the body of the declaration, where subsequent allocations and assignments are carried out. More formally, given a declaration of the form `let` x `in` t, the goal is to evaluate t in a context where x is defined as a fresh identifier. The following is a first attempt:

Variable declarations (attempt)

$$\textsc{E-Let}\ \frac{C[\pi.x \mapsto 0] \vdash t \Downarrow l, C'}{C \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow l, C'}$$

Recall that variable declarations are supposed to delimit scopes. The above rule is therefore incorrect, because it keeps x in the pointer table beyond the evaluation of t. Instead, C' must be updated to remove the freshly defined identifier. We refine our attempt accordingly.

Variable declarations (attempt)

$$\textsc{E-Let}\ \frac{C[\pi.x \mapsto 0] \vdash t \Downarrow l, C'}{C \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow l, C'[\pi \mapsto C'.\pi|_x]}$$

The update $C'[\pi \mapsto C'.\pi|_x]$ now removes x from $C'.\pi$'s domain, thereby ending the identifier's lifetime. Unfortunately, this solution is also inaccurate, due to *name shadowing*. Indeed, if x is already defined in $C.\pi$, then the update $C[\pi.x \mapsto 0]$ shadows its mapping. The shadowing must be lifted once t's evaluation is complete. This is achieved by the means of a function unset, that removes a given identifier from a pointer table, or restores its mapping if it has been shadowed.

$$\text{unset}(x, \pi', \pi) = \begin{cases} \pi'[x \mapsto \pi.x] & \text{if } x \in \text{dom}(\pi) \\ \pi'|_x & \text{otherwise} \end{cases}$$

With that, the semantics of variable declarations can finally be defined properly:

Variable declarations

$$\textsc{E-Let}\ \frac{C[\pi.x \mapsto 0] \vdash t \Downarrow l, C'}{C \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow l, C'[\pi \mapsto \text{unset}(x, C'.\pi, C.\pi)]}$$

Example 4.3.1: Name shadowing

Consider the following term:

$$\texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ x, y\ ;\ x \leftarrow 2\ ;\ y \leftarrow 4\ ;\ \texttt{let}\ x\ \texttt{in}\ x\ \texttt{\&-}\ y$$

Let C be an evaluation context defined so that $C.\pi.x = l_0$ and $C.\mu.l_0 = 2$. We skip the grayed out parts and focus on the second declaration, whose

evaluation is described by the following derivation:

$$\frac{C[\pi.x \mapsto 0] \vdash x \mathbin{\&-} y \Downarrow l, C'}{C \vdash \texttt{let}\ x\ \texttt{in}\ x \mathbin{\&-} y \Downarrow l, C'[\pi \mapsto \text{unset}(x, C'.\pi, C.\pi)]}$$

The second x is no longer bound to l_0 before x &- y is evaluated, due to the premise $C[\pi.x \mapsto 0]$. However, the alias to y is removed from the pointer table in the conclusion, as $\text{unset}(x, C'.\pi, C.\pi)$ *computes $C'[\pi.x \mapsto l_0]$, thus restoring the binding of the first x.*

Memory Management

Given an evaluation context C, an allocation equates to picking a memory location l that is not in the domain of C's memory table, that is finding l such that $l \notin \text{dom}(C.\mu)$. Recall that we assume an unbounded memory. Therefore finding such a location is always possible, or more formally:

$$\frac{}{\exists l \notin \text{dom}(C.\mu)}$$

In the $\mathcal{A}$-calculus, atomic and function literals are always evaluated as newly allocated objects. Assignment operators can then appropriately consume the memory locations at which they are stored, just like for any other expression, and thereby remain agnostic of their right operand.

Literal values

E-Atom
$$\frac{a \in \mathbb{A} \qquad \exists l \notin \text{dom}(C.\mu)}{C \vdash a \Downarrow l, C[\mu.l \mapsto x]}$$

E-Fun
$$\frac{\exists l \notin \text{dom}(C.\mu)}{C \vdash \lambda\overline{x} \triangleright t \Downarrow l, C[\mu.l \mapsto \lambda\overline{x} \triangleright t]}$$

A literal term always represents the same value. Therefore an alternative implementation could be to use the *flyweight pattern* [65]. In a nutshell, the approach consists of pre-allocating memory for all literals found in the program, and evaluating literal terms as the corresponding memory location at runtime. This method can save memory and avoid many unnecessary allocations, as immutable objects can easily be reused. This is for instance how Swift operates to allocate constant strings. However, we take another direction because this approach is nothing more than an optimization, whereas the purpose of the $\mathcal{A}$-calculus is to accurately represent the semantics of any existing implementation.

Dereferencing an identifier consists in consulting the pointer table in order to retrieve the memory to which it is associated. Notice that the rule does not apply if the identifier is undefined.

Variable dereferencing

$$\textsc{E-Var}\ \frac{x \in \mathrm{dom}(C.\pi)}{C \vdash x \Downarrow C.\pi.x, C}$$

Variable allocations borrow from the same principle and associate a given identifier to a new location, holding an uninitialized value. Deallocations simply consist in the update of the memory table to remove a memory location.

Allocations and deallocations

$$\textsc{E-New}\ \frac{x \in \mathrm{dom}(C.\pi) \qquad \exists l \notin \mathrm{dom}(C.\mu)}{C \vdash \texttt{alloc}\ x \Downarrow l, C[\pi.x \mapsto l, \mu.l \mapsto \bot]}$$

$$\textsc{E-Del}\ \frac{C \vdash t \Downarrow l, C' \qquad l \in \mathrm{dom}(C'.\mu) \qquad l \neq 0}{C \vdash \texttt{del}\ t \Downarrow l, C'[\mu \mapsto C'.\mu|_l]}$$

While a deallocation evaluates to the memory location it has freed, using it in an assignment will obviously lead to a memory error. Hence, although using a deallocation as a right operand to an assignment operator is syntactically and semantically defined, doing so will inevitably result in an erroneous execution if the assigned variable is eventually used.

Assignment Semantics

The advantage of evaluating all expressions as memory locations emerges when formalizing the assignment operators, whose semantics can be expressed purely in terms of pointer and memory map updates. While many modern languages no longer consider assignments as assignable expressions, this feature is still used prominently in languages like C and Java. As a result, assignments are first-class terms in the $\mathcal{A}$-calculus, and produce the memory location assigned to their left operand.

An aliasing assignment is the simplest to evaluate. It should evaluate its right operands to determine the memory location it represents, before updating the pointer table to map its left operand to that location, thus resulting in the creation of an alias.

Aliasing assignment

E-Alias

$$\frac{C \vdash t \Downarrow l, C'}{C \vdash x \,\&\text{-}\, t \Downarrow l, C'[\pi.x \mapsto l]}$$

Remark that if x is the only remaining reference to a memory location l' before the assignment, then access to that particular location is lost after. In the absence of any automatic garbage collection strategy, this corresponds of course to a memory leak (c.f. Section 5.2.4).

Example 4.3.2: Aliasing assignment

Let x and y be two initialized variables. Let $C_1 \vdash y \,\&\text{-}\, x \Downarrow l, C_2$ be the conclusion of an aliasing assignment, then:

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto v_1 \\ y \mapsto m & m \mapsto v_2 \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto v_1 \\ y \mapsto l & m \mapsto v_2 \end{array}$$

Mutation semantics consists in taking the value represented by its right operand, and store a copy at the location represented by its left operand. Since $\mathbb{V}$ only contains atomic values and thin first-class functions (i.e. functions without closure), copying is a trivial operation. Note that the left operand must refer to a valid memory location, as checked by $l \neq 0$, and that the right operand must be initialized, as checked by $v \notin \{\bot, \square\}$.

Mutation assignment

E-Mut

$$\frac{\begin{array}{c} C \vdash t \Downarrow m, C' \\ m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad l = C'.\pi.x \quad l \neq 0 \end{array}}{C \vdash x \coloneqq t \Downarrow l, C'[\mu.l \mapsto v]}$$

Example 4.3.3: Mutation assignment

Let x and y be two initialized variables. Let $C_1 \vdash y \coloneqq x \Downarrow m, C_2$ be the conclusion of a mutation assignment, then:

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto v_1 \\ y \mapsto m & m \mapsto v_2 \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto v_1 \\ y \mapsto m & m \mapsto v_1 \end{array}$$

Notice that the right operand is evaluated *before* its left counterpart. This order

is relevant because any term evaluation may have side effects on the context. As such, t has the potential to change the pointer table for x.

Example 4.3.4: Evaluation order

Consider the following term:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \coloneqq \texttt{del}\ x$$

Let C be an empty evaluation context, the term's evaluation is illustrated by the following failed derivation:

$$\dfrac{\dfrac{\dfrac{x \in \mathrm{dom}(C.\pi)}{C \vdash x \Downarrow C.\pi.x, C}}{C \vdash \texttt{del}\ x \Downarrow C.\pi.x, C[\mu \mapsto C.\mu|_l]} \quad v = C[\mu \mapsto C.\mu|_{C.\pi.x}].\mu.(C.\pi.x)}{C \vdash \texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \coloneqq \texttt{del}\ x \Downarrow l, C'}$$

The derivation fails because x's memory is deallocated before the mutation assignment takes place, removing $C.\pi.x$ (i.e. the memory location to which x was bound) from the memory table's domain. Therefore v is undefined.

The semantics of the move operator is very similar to that of the mutation operator. It only differs in that the value of the right operand should be *moved*, in order to satisfy the the operator's linear semantics. This is carried out by the update $C''[\mu.m \mapsto \square]$ in the rule's conclusion.

Move assignment

E-Move

$$\dfrac{\begin{array}{c} C \vdash t \Downarrow m, C' \\ m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad l = C'.\pi.x \quad l \neq 0 \end{array}}{C \vdash x \leftarrow t \Downarrow l, C'[\mu.m \mapsto \square][\mu.l \mapsto v]}$$

Example 4.3.5: Move assignment

Let x and y be two initialized variables. Let $C_1 \vdash y \leftarrow x \Downarrow m, C_2$ be the conclusion of a move assignment, then:

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto v_1 \\ y \mapsto m & m \mapsto v_2 \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l & l \mapsto \square \\ y \mapsto m & m \mapsto v_1 \end{array}$$

Functions

The next set of rules describes the semantics of function calls. Roughly speaking, a function is simply a piece of reusable code that is inlined at each of its call sites, and whose passing of parameters and return value are merely assignments. For example:

$$x \leftarrow (\lambda y \triangleright t \,;\, \texttt{ret} \leftarrow y)(y \leftarrow x) \equiv \texttt{let}\ y\ \texttt{in}\ y \leftarrow x \,;\, t \,;\, x \leftarrow y$$

We subdivide function calls' evaluations into three steps:

Evaluate the callee This step is described by a dedicated evaluation operator $\Downarrow^{\text{callee}}$ that retrieves the function value (e.g. $\lambda x \triangleright x$) represented by the callee. Unlike $\Downarrow$, this operator produces function values (i.e. elements of $\mathbb{V}$) rather than memory locations.

Evaluate the arguments This step is described by another dedicated evaluation operator $\Downarrow^{\text{args}}$ that processes argument lists recursively.

Evaluate the body A function's body is merely a term that should be evaluated in the proper evaluation context. Hence this last step is described by $\Downarrow$.

The first step consists in getting the memory location represented by the callee, in order to determine the function's parameters and body.

Callee evaluation

E-Callee
$$\frac{C \vdash t_c \Downarrow l, C' \qquad l \in \text{dom}(C'.\mu) \qquad C'.\mu.l \notin \{\bot, \square\} \qquad C'.\mu.l = \lambda\overline{x} \triangleright t}{C \vdash t_c \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C'}$$

Recall that we reuse our assignment operators to specify the parameter passing policy. The alias operator corresponds to the *pass-by-alias* semantics, the mutation operator corresponds to the *pass-by-value* semantics, and the move operator considers arguments as linear resources. Consequently, a parameter can be understood as a variable whose declaration and initialization precede the function's body, while an argument describes how its corresponding parameter is initialized. Therefore, it makes sense to treat each argument as a regular assignment, leading to the following first attempt at the definition of $\Downarrow^{\text{args}}$'s semantics.

Argument lists (attempt)

E-Args-0
$$\frac{}{C \vdash \epsilon \Downarrow^{\text{args}} 0, C}$$

E-Args-N
$$\frac{C \vdash \texttt{let}\ x\ \texttt{in}\ x \diamond t \Downarrow l_x, C_x \qquad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

Despite the similarities between parameters' handling and variable declarations, E-Let cannot be reused directly. The reason is that parameter names must be preserved in the domain of the pointer table until the function's body is evaluated, while E-Let's name shadowing mechanism removes them in the conclusion of the rule, applying unset function. In other words, given an evaluation context C_1 and a list of parameter assignments $\overline{u}$, the pointer table of context C_2 that would result from the evaluation of $C_1 \vdash \overline{u} \Downarrow 0, C_2$ would not contain the name of the parameters in $\overline{u}$ with this first attempt. Instead, E-Let's name shadowing mechanism shall be discarded.

Argument lists (attempt)

$$\text{E-Args-0}\ \frac{}{C \vdash \epsilon \Downarrow^{\text{args}} 0, C} \qquad \text{E-Args-N}\ \frac{C[\pi.x \mapsto 0] \vdash x \diamond t \Downarrow l_x, C_x \qquad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

This second attempt is unfortunately incorrect as well, because it does not allow passing parameters by value or move. The problem is due to the fact that all parameter identifiers refer to the null pointer before the argument's evaluation. Therefore a mutation or move assignment will fail to properly dereference the parameter name. The solution is to perform an allocation right before evaluating parameters passed by value or move.

Argument lists

$$\text{E-Args-0}\ \frac{}{C \vdash \epsilon \Downarrow^{\text{args}} 0, C} \qquad \text{E-Args-Alias}\ \frac{C[\pi.x \mapsto 0] \vdash x\ \&\text{-}\ t \Downarrow l_x, C_x \qquad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x\ \&\text{-}\ t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

$$\text{E-Args-MM}\ \frac{\diamond \in \{:=, \leftarrow\} \qquad \exists l \notin \text{dom}(C.\mu) \qquad C[\pi.x \mapsto l, \mu.l \mapsto \bot] \vdash x \diamond t \Downarrow l_x, C_x \qquad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

Example 4.3.6: Argument List

Consider the following function call:

$$f(x\ \&\text{-}\ z, y \leftarrow 2)$$

The term contains a list of parameter assignments $\overline{u} = x\ \&\text{-}\ z, y \leftarrow 2$, whose

evaluation by $\Downarrow^{\text{args}}$ is described below. Given a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline z \mapsto l_1 & l_1 \mapsto 4 \end{array}$$

E-Args-Alias *applies first and evaluates x &- z:*

$$\frac{\dfrac{\vdots}{C[\pi.x \mapsto 0] \vdash x \,\&\text{-}\, a \Downarrow l_x, C_x} \quad \dfrac{\vdots}{C_x \vdash y \leftarrow b \Downarrow^{\text{args}} 0, C_y}}{C \vdash x \,\&\text{-}\, a, y \leftarrow b \Downarrow^{\text{args}} 0, C_y}$$

As a result, C_x is now defined as follows:

$$C_x = \begin{array}{c|c} \pi & \mu \\ \hline z \mapsto l_1 & l_1 \mapsto 4 \\ x \mapsto l_1 & \end{array}$$

Next applies E-Args-MM*, to evaluate $y \leftarrow 2$, before the recursion ends.*

$$\frac{\dfrac{\vdots}{C[\pi.y \mapsto l_2, \mu.l_2 \mapsto \bot] \vdash y := 2 \Downarrow l_y, C_y} \quad \dfrac{}{C_y \vdash \epsilon \Downarrow^{\text{args}} 0, C_y}}{C_x \vdash y \leftarrow 2 \Downarrow^{\text{args}} 0, C_y}$$

As a result, C_y is now given as follows:

$$C_y = \begin{array}{c|c} \pi & \mu \\ \hline z \mapsto l_1 & l_1 \mapsto 4 \\ x \mapsto l_1 & l_2 \mapsto 2 \\ y \mapsto l_2 & l_3 \mapsto \square \end{array}$$

Note that l_3 denotes the memory location allocated for the literal 4, which has been moved by the passing of the parameter y.

As any other expression, a function call should be evaluated as a memory location. Unfortunately, this location cannot be known at the call site, because it depends on the return statement that will be executed. The support for multiple return semantics further complicates the task as to whether an allocation is required. One solution is to take a similar path as that we took for the arguments, and interpret return statements as regular assignments. Let $\mathfrak{R} \in \mathbb{X}$ be a reserved identifier denoting a virtual *return identifier*. Then a return statement is simply

an assignment to the return identifier. In other words, `ret` $\diamond\, t$ can be considered equivalent to $\mathfrak{R} \diamond t$. Just as for arguments, an allocation is needed unless the return value is passed by alias.

Function returns

$$\text{E-Ret-Alias}\ \frac{C[\pi.\mathfrak{R} \mapsto 0] \vdash \mathfrak{R}\ \&\text{-}\ t \Downarrow l, C'}{C \vdash \texttt{ret}\ \diamond\, t \Downarrow l, C'} \qquad \text{E-Ret-MM}\ \frac{\diamond \in \{:=, \leftarrow\} \qquad \exists l \notin \mathrm{dom}(C.\mu) \qquad C[\pi.\mathfrak{R} \mapsto l, \mu.l \mapsto \bot] \vdash \mathfrak{R} \diamond t \Downarrow l, C'}{C \vdash \texttt{ret}\ \diamond\, t \Downarrow l, C'}$$

The rule E-Call assembles all pieces together, but several considerations must be made regarding pointer tables.

Function calls (attempt)

$$\text{E-Call}\ \frac{C \vdash f \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C_1 \qquad C_1 \vdash \overline{u} \Downarrow^{\text{args}} 0, C_2 \qquad C_2 \vdash t' \Downarrow l, C'}{C \vdash f(\overline{u}) \Downarrow l, C'}$$

This first attempt is incorrect, because name shadowing must also be taken into account. Indeed parameter names may shadow identifiers from the call site. Furthermore, if the function call occurs in a return expression (e.g. `ret` $\leftarrow f()$), the return identifier may also be shadowed. The solution consists of applying unset for $\mathfrak{R}$ and each of the parameter identifiers $\overline{x}$. This requires a minor extension of unset to words of identifiers.

Function calls (attempt)

$$\text{E-Call}\ \frac{C \vdash f \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C_1 \qquad C_1 \vdash \overline{u} \Downarrow^{\text{args}} 0, C_2 \qquad C_2 \vdash t \Downarrow l, C'}{C \vdash f(\overline{u}) \Downarrow l, C'[\pi \mapsto \mathrm{unset}(\mathfrak{R}\overline{x}, C_1.\pi, C.\pi)]}$$

Notice that C''s pointer map is updated by applying unset on $C_1.\pi$ rather than $C'.\pi$. The reason is that $C'.\pi$ is defined for the parameters of the function defined for the function parameters' names, which have to be removed from the pointer table. In fact, C''s pointer table can be completely discarded.

The alert reader will notice that this solution is still incorrect if the name of the parameter appears in arguments' terms (e.g. in a function call of the form $f(x\ \&\text{-}\ x)$). Recall that E-Args-Alias and E-Args-MM emulate a variable declaration for each parameter in order to deal with name shadowing. Consequently, a name may be shadowed before its argument is evaluated. The solution is to leverage the λ-calculus's α-renaming transformation, and rename all identifiers in the callee so that none of them can clash with the identifiers in the current scope. Note

that this does not entirely get rid of shadowing, as the return identifier cannot be renamed.

Function calls

$$\textsc{E-Call}\quad \frac{C \vdash f \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C_1 \qquad \lambda\overline{x}' \triangleright t' = \text{rename}_{C_1}(\lambda\overline{x} \triangleright t) \qquad C_1 \vdash \overline{u}[/\{\forall_{i=1}^{\|\overline{x}\|}, \overline{x}_i \mapsto \overline{x}'_i\}] \Downarrow^{\text{args}} 0, C_2 \qquad C_2 \vdash t \Downarrow l, C'}{C \vdash f(\overline{u}) \Downarrow l, C'[\pi \mapsto \text{unset}(\Re, C_2.\pi, C.\pi)]}$$

Example 4.3.7: Function call

Consider the following function call:

$$f(y \leftarrow 8)$$

Assume it is evaluated in a context C defined as such:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline f \mapsto l_1 & l_1 \mapsto \lambda y \triangleright \texttt{ret} \leftarrow y \end{array}$$

The first step consists in dereferencing f to a function value. In this particular example, this boils down to the application of E-Var*:*

$$\frac{\dfrac{f \in C.\pi}{C \vdash f \Downarrow l, C}\ \textsc{E-Var} \qquad C.\mu.l = \lambda y \triangleright \texttt{ret} \leftarrow y}{C \vdash f \Downarrow^{\text{callee}} (\lambda y \triangleright \texttt{ret} \leftarrow y), C}\ \textsc{E-Callee}$$

Since none of the variables in $y \triangleright \texttt{ret} \leftarrow y$ conflicts with the identifiers in $\text{dom}(C.\pi)$*, the callee's renaming can yield the same function. The second step evaluates the function's argument $y \leftarrow 8$, from which stems an evaluation context C_y in which a new memory location l_2 has been allocated for the parameter y, holding the value* 8*:*

$$\frac{\exists l_2 \notin \text{dom}(C.\mu) \qquad C[\pi.y \mapsto l_2, \mu.l_2 \mapsto \bot] \vdash y \leftarrow 8 \Downarrow l_y, C_y \qquad \overline{C_y \vdash \epsilon \Downarrow^{\text{args}} 0, C_y}}{C \vdash y \coloneqq 8 \Downarrow^{\text{args}} 0, C_y}\ \textsc{E-Args-MM}$$

In E-Call*, non-parameter identifiers are filtered from $C_y.\pi$, resulting in the following intermediate evaluation context:*

$$C'_y = C_y[\pi.x \mapsto C_y.\pi.x \mid x \in \overline{x}] = \begin{array}{c|c} \pi & \mu \\ \hline y \mapsto l_2 & l_1 \mapsto \lambda y \triangleright \texttt{ret} \leftarrow y \\ & l_2 \mapsto 8 \\ & l_3 \mapsto \square \end{array}$$

The third step evaluates the body of the function, requiring another allocation to satisfy the return by move semantics.

$$\frac{\exists l_4 \notin \mathrm{dom}(C_y.\mu) \qquad C'_y[\pi.\mathfrak{R} \mapsto l_4, \mu.l_4 \mapsto \bot] \vdash \mathfrak{R} \leftarrow y \Downarrow l_4, C'}{C'_y \vdash \texttt{ret} \leftarrow y \Downarrow l, C'} \text{ E-RET-MM}$$

This results in the memory location l_4 the following evaluation context:

$$C'[\pi \mapsto \mathrm{unset}(\mathfrak{R}, C_y.\pi, C.\pi)] = \begin{array}{c|l} \pi & \mu \\ \hline f \mapsto l_1 & l_1 \mapsto \lambda y \triangleright \texttt{ret} \leftarrow y \\ & l_2 \mapsto \square \\ & l_3 \mapsto \square \\ & l_4 \mapsto 8 \end{array}$$

Finally, the evaluation of a sequence is given by that of both its subterms, in order. Remember that our semantics is based on the assumption that return statements are never followed by any other expression. Indeed, as we do not implement a mechanism to "return early", all terms following a return statement would otherwise be evaluated as well. Such programs are considered ill-formed.

Sequences

$$\text{E-SEQ} \quad \frac{C \vdash t_1 \Downarrow l_1, C_1 \qquad C_1 \vdash t_2 \Downarrow l_2, C_2}{C \vdash t_1; t_2 \Downarrow l_2, C_2}$$

4.3.3 Extending the Calculus

Our calculus being very minimal, implementing elaborate constructs can prove challenging. Consider conditional expressions for example. Borrowing from Church or Mogensen–Scott encodings [83, 102], boolean literals can be represented as functions that unconditionally return either of two arguments. Therefore, a predicate can be expressed by the function below:

$$\lambda p, x, a, b \triangleright (\texttt{ret} := p(x := x)(t := a, f := b))$$

While such approach is theoretically sound, it significantly hinders the practicality of our calculus for various reasons. First, since the encoding relies extensively on function application, a lot of care must be taken to choose the appropriate parameter passing policy. Using the mutation operator is a safe choice, as we do not run the risk to accidentally create aliases, nor to unintendedly "consume" any

variable with a destructive read. However, there might be cases in which this strategy is either suboptimal or even inaccurate with respect to the modeled semantics. One possible solution is to support polymorphic assignment operators, therefore providing some level of genericity, though this would considerably elevate the complexity of the calculus. Second, since our calculus does not support higher-order functions, representing composite structures with the usual Church or Mogensen–Scott methods is actually intractable. While this limitation could be lifted, it would require a very powerful type system to accommodate any sort of static analysis. Consider for instance the following Church encoding of a pair:

$$\lambda x, y \triangleright (\texttt{ret} \leftarrow \lambda f \triangleright (\texttt{ret} \leftarrow f(x \,\&\text{-}\, x, y \,\&\text{-}\, y)))$$

This describes a function that takes two parameters x and y, representing the components of a pair. When applied, it returns another function that will have captured both arguments by closure, and accepts yet another function as parameter which it applies to the values it captured. Therefore, writing an accessor simply consists in passing a function that returns its first argument.

$$\begin{array}{l}
\texttt{let}\ \mathit{enc}, p\ \texttt{in} \\
\quad \texttt{alloc}\ \mathit{enc}, p\ ; \\
\quad \mathit{enc} \leftarrow (\lambda x, y \triangleright (\texttt{ret} \leftarrow \lambda f \triangleright (\texttt{ret} \leftarrow f(x \,\&\text{-}\, x, y \,\&\text{-}\, y))))\ ; \\
\quad p \leftarrow \mathit{enc}(x := 4, y := 8)\ ; \\
\quad p(f \leftarrow \lambda x, y \triangleright \texttt{ret} := x)
\end{array}$$

Unfortunately, typing *enc* requires a type system that can manipulate existentially typed functions as first-class citizen, without instantiating them at the moment of their declaration. The reason is that the returned function accepts as parameter a function that takes the two components of the pair, and returns some result, whose type is therefore unknown at the time the pair encoding is created. In fact, such a typing is not possible in most mainstream statically typed programming languages. In order to tackle these shortcomings, another plan of attack is to extend the calculus with more abstract constructions, so that we no longer need to rely exclusively on higher-order functions.

Records

The first extension we propose is a support for records. Syntactically, we need two additional constructs, namely one that creates new record values, and one that allows access (read and write) to its fields:

Definition 4.3.4: Record syntax

Let the set of terms $\mathcal{AC}_R$ be a superset of $\mathcal{AC}$ to support records, defined as the minimum set such that:

$$\begin{aligned}
\ldots \in \mathcal{AC}_R & && \textit{terms of } \mathcal{AC} \\
\overline{x} \in \mathbb{X}^{\#} &\implies \texttt{new}\ \overline{x} \in \mathcal{AC}_R && \textit{an instantiation} \\
t \in \mathcal{AC}_R \wedge x \in \mathbb{X} &\implies t.x \in \mathcal{AC}_R && \textit{a field} \\
t \in \mathcal{AC}_R \wedge x \in \mathbb{X} &\implies \texttt{alloc}\ t_1.x \in \mathcal{AC}_R && \textit{a field allocation} \\
\diamond \in \mathbb{O} \wedge x \in \mathbb{X} \wedge t_1, t_2 \in \mathcal{AC}_R &\implies t_1.x \diamond t_2 \in \mathcal{AC}_R && \textit{a field assignment}
\end{aligned}$$

Notice the addition of a syntactic construction for field assignments, which lets fields appear as left operands. We also extend the semantic domain $\mathbb{V}$ to include a representation for records, which we provide in the form of a table mapping identifiers to memory locations.

Definition 4.3.5: Record values

Let the set of semantic values $\mathbb{V}_R$ be a superset of $\mathbb{V}$ to support records, defined as the minium set such that:

$$\begin{aligned}
v \in \mathbb{V} &\implies v \in \mathbb{V}_R && \textit{values of } \mathcal{AC} \\
X \subseteq \mathbb{X}, x : X \to \mathbb{N} &\implies x \in \mathbb{V}_R && \textit{a record}
\end{aligned}$$

Finally, we extend the operational semantics to support the evaluation of the records related constructs. We use an operator $\Downarrow_R$ to express such semantics, defined for the terms $t \in \mathcal{AC}$ of the regular $\mathcal{A}$-calculus such that:

$$\frac{t \in \mathcal{AC} \qquad t \neq (x := u) \qquad C \vdash t \Downarrow l, C'}{C \vdash t \Downarrow_R l, C'}$$

In other words, the semantics of $\Downarrow_R$ is identical to that of $\Downarrow$ for the terms of the regular $\mathcal{A}$-calculus, except for mutation assignments, whose semantics requires minor amendments that will be described later[3].

We define $\Downarrow_R$ for the syntactic extensions introduced in Definition 4.3.4, starting with record instantiations. These necessitate an allocation for the new record instance, whose fields all refer to the null pointer.

[3]The reader will notice that this formal description is actually inaccurate, as it does not account for rules that evaluate subterms in their premises. Therefore, all rules should in fact be redefined so that occurence of $\Downarrow$ in their premises are replaced with $\Downarrow_R$.

Instantiations

ER-Inst
$$\frac{\exists l \notin \mathrm{dom}(C.\mu) \qquad r = \langle \overline{x}_i \mapsto 0 | 1 \leq i \leq \| \overline{x} \| \rangle}{C \vdash \texttt{new}\ \overline{x} \Downarrow_R l, C[\mu.l \mapsto r]}$$

Example 4.3.8: Instantiation

Consider the term `new` p, q*, expressing the instantiation of a 2D point. Its evaluation is described by the following derivation:*

$$\frac{\exists l \notin \mathrm{dom}(C.\mu) \qquad r = \langle p, q \mapsto 0 \rangle}{C \vdash \texttt{new}\ p, q \Downarrow_R l, C[\mu.l \mapsto r]}\ \text{ER-Inst}$$

Assuming an empty initial context C, the result is given by:

$$C[\mu.l \mapsto \langle p, q \mapsto 0 \rangle] = \begin{array}{c|c} \pi & \mu \\ \hline & l \mapsto \langle p \mapsto 0, q \mapsto 0 \rangle \end{array}$$

A record is essentially a pointer table, mapping each field label to a particular memory location. Hence, operations on variables are naturally extended to fields. Let rec be a predicate that holds for record values. The next rule describes allocation.

Field allocations

ER-Field-New
$$\frac{\begin{array}{c} C \vdash t \Downarrow_R l, C' \\ l \in \mathrm{dom}(C'.\mu) \quad r = C'.\mu.l \quad \mathrm{rec}(r) \quad x \in \mathrm{dom}(r) \quad m \notin \mathrm{dom}(C'.\mu) \end{array}}{C \vdash \texttt{alloc}\ t.x \Downarrow_R m, C[\mu.l.x \mapsto m, \mu.m \mapsto \bot]}$$

Dereferencing a field is also very similar to dereferencing a variable (c.f. E-Var). The difference lies in the pointer map in which the memory location is retrieved. Variables are defined in the context's pointer table, while fields are defined in the record represented by the left operand.

Field dereferencing

ER-Field
$$\frac{C \vdash t \Downarrow_R l, C' \qquad l \in \mathrm{dom}(C'.\mu) \qquad r = C'.\mu.l \qquad \mathrm{rec}(r) \qquad x \in \mathrm{dom}(r)}{C \vdash t.x \Downarrow r.x, C'}$$

Obviously, all fields assignments should evaluate their left operand as a field. Nonetheless, evaluations are still carried out from right to left, so the potential side

effects from the right operand's evaluation apply before that of the left. Aliasing assignments on fields should modify the left operand's pointer table rather than that of the evaluation context.

Field aliasing assignment

ER-Field-Alias

$$\frac{C \vdash t_2 \Downarrow_R m, C' \qquad C' \vdash t_1 \Downarrow_R l, C'' \quad l \in \mathrm{dom}(C''.\mu) \quad r = C''.\mu.l \quad \mathrm{rec}(r) \quad x \in \mathrm{dom}(r)}{C \vdash t_1.x \;\&\text{-}\; t_2 \Downarrow_R m, C''[\mu.l.x \mapsto m]}$$

Example 4.3.9: Field aliasing assignment

Consider the following term:

$$x \leftarrow 2 \;;\; y \leftarrow \texttt{new}\ p \;;\; y.p \;\&\text{-}\; x$$

We skip the grayed out parts and focus on the field aliasing assignment. Let $C_1 \vdash y.p \;\&\text{-}_R x \Downarrow_R l_0, C_2$ *be the conclusion of its evaluation, then:*

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 2 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto 0 \rangle \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 2 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto l_1 \rangle \end{array}$$

Recall that the mutation operator is defined so that it modifies the left operand with a *transitive* (a.k.a. deep) copy of its right one. As a result, the operator can no longer simply duplicate the semantic value stored in $C.\mu$, as this would result in the creation of aliases for each of the fields. Instead, we shall define a function copy to perform such transitive operation. E-Mut is then modified as follows.

Mutation assignment

ER-Mut

$$\frac{C \vdash t \Downarrow m, C' \quad m \in \mathrm{dom}(C'.\mu) \qquad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad u, C'' = \mathrm{copy}(v, C') \quad l = C''.\pi.x \quad l \neq 0}{C \vdash x := t \Downarrow l, C''[\mu.l \mapsto u]}$$

Applying the mutation assignment operator on a field as a left operand is defined similarly, with the difference that the left operand is dereferenced. Note that the field must refer to a valid memory location, as checked by $C''.\mu.l \neq 0$. This of course mimics the check that is performed on the identifier in ER-Mut.

Field mutation assignment

ER-Field-Mut

$$\frac{C \vdash t_2 \Downarrow_R m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad u, C_u = \mathrm{copy}(v, C') \quad C_u \vdash t_1.x \Downarrow_R l, C'' \quad C''.\mu.l \neq 0}{C \vdash t_1.x := t_2 \Downarrow_R l, C''[\mu.l \mapsto u]}$$

Example 4.3.10: Field mutation assignment

Consider the following term:

$$x \leftarrow 2 \; ; \; y \leftarrow \texttt{new}\; p \; ; \; y.p \; \&\text{-}\; x \; ; \; y.p := z$$

We skip the grayed out parts and focus on the field mutation assignment. Let $C_1 \vdash y.p := z \Downarrow l, C_2$ be the conclusion of its evaluation, then:

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 2 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto l_1 \rangle \\ z \mapsto l_3 & l_3 \mapsto 8 \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 8 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto l_1 \rangle \\ z \mapsto l_3 & l_3 \mapsto 8 \end{array}$$

Move assignments on field records are defined similarly.

Field move assignment

ER-Field-Move

$$\frac{C \vdash t_2 \Downarrow_R m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad C' \vdash t_1.x \Downarrow_R l, C'' \quad C''.\mu.l \neq 0}{C \vdash t_1.x \leftarrow t_2 \Downarrow_R l, C''[\mu.l \mapsto v, \mu.m \mapsto \square]}$$

Note that E-Move does not need to be modified to destructively read the fields of a moved record. Indeed, while the operator treats the record as a linear value, its fields may be aliased freely. This allows to move self referential structures (e.g. graphs) with an externally unique handle [40].

Example 4.3.11: Field move assignment

Consider the following term:

$$x \leftarrow 2 \; ; \; y \leftarrow \texttt{new}\; p \; ; \; y.p \; \&\text{-}\; x \; ; \; y.p \leftarrow z$$

We skip the grayed out parts and focus on the field move assignment. Let

$C_1 \vdash y.p \leftarrow z \Downarrow_R l, C_2$ be the conclusion of its evaluation, then:

$$C_1 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 2 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto l_1 \rangle \\ z \mapsto l_3 & l_3 \mapsto 8 \end{array} \implies C_2 = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 8 \\ y \mapsto l_2 & l_2 \mapsto \langle p \mapsto l_1 \rangle \\ z \mapsto l_3 & l_3 \mapsto \square \end{array}$$

Conditional Terms

Our second extension brings conditional terms to our calculus. Syntactically, we need only a single additional construct:

Definition 4.3.6: Conditional expression syntax

Let the set of terms $\mathcal{AC}_C$ be a superset of $\mathcal{AC}$ to support conditional terms, defined as the minimum set such that:

$$\begin{array}{ll} \ldots \in \mathcal{AC}_C & \textit{terms of } \mathcal{AC} \\ t_1, t_2, t_3, \in \mathcal{AC}_C \implies \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in \mathcal{AC}_C & \textit{a conditional term} \end{array}$$

A conditional expression `if` t_1 `then` t_2 `else` t_3 reads as "if t_1 is true, then evaluate t_2, otherwise evaluate t_3". Consequently, we also need to identify values from our semantic domain that can be qualified as "true". To this end, we assume `true`, `false` $\in \mathbb{A}$ to be atomic values denoting booleans[4]. We extend the operational semantics of the regular $\mathcal{A}$-calculus to support condition terms' evaluations with an operator $\Downarrow_C$, defined for the terms $t \in \mathcal{AC}$ of the regular $\mathcal{A}$-calculus such that:

$$\frac{t \in \mathcal{AC} \qquad C \vdash t \Downarrow l, C'}{C \vdash t \Downarrow_C l, C'}$$

Conditional term evaluation simply boils down to the evaluation of its condition, so as to choose the next expression to evaluate. In most contemporary languages, branches of a conditional term delimit lexical scopes, so that variables declared in one are not "visible" in the other. It is not necessary in the $\mathcal{A}$-calculus, because identifier scopes are explicitly delimited by variable and function declarations. Therefore conditional terms do not have to deal with shadowing.

[4]Some languages (e.g. C) may feature a concept of truthy and falsey values rather or in addition to booleans. We choose otherwise for the sake of conciseness.

Conditional terms

$$\textsc{EC-Cond-T}\quad \frac{C \vdash t_1 \Downarrow_C l_1, C' \qquad l_1 \in \mathrm{dom}(C'.\mu) \qquad C'.\mu.l_1 \notin \{\bot, \square\} \qquad C'.\mu.l_1 = \texttt{true} \qquad C' \vdash t_2 \Downarrow_C l_2, C''}{C \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_C l_2, C''}$$

$$\textsc{EC-Cond-F}\quad \frac{C \vdash t_1 \Downarrow_C l_1, C' \qquad l_1 \in \mathrm{dom}(C'.\mu) \qquad C'.\mu.l_1 \notin \{\bot, \square\} \qquad C'.\mu.l_1 = \texttt{false} \qquad C' \vdash t_3 \Downarrow_C l_3, C''}{C \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_C l_3, C''}$$

The rules produce the location of the branch that is evaluated, in order to allow the modeling of ternary operators (e.g. `?:` in C), as found in most C-family languages.

Conditional terms can of course be used in combination with the record extension presented above. Unless stated otherwise, we consider both extensions to be part of the syntax and semantics of the $\mathcal{A}$-calculus for the remainder of this thesis. By abuse of notation, we simply write $\mathcal{AC}$ for the terms of the $\mathcal{A}$-calculus, extended with records and conditional expressions, and we simply write $\Downarrow$ to denote its operational semantics.

4.4 Examples

We now present two examples of translations from real programming languages to the $\mathcal{A}$-calculus. For the sake of legibility, we use a concrete syntax rather than the abstract one we described in Section 4.3. In concrete syntax, sequences of terms are delimited by line breaks rather than semicolons, and lexical scopes are delimited by curly braces. We write `fun (x,y) { e }` in concrete syntax for $\lambda x, y \triangleright e$. We write `return` rather than `ret`, and enclose member names in angle brackets in records' allocations. For instance, we write `new <m,n>` for `new` m, n.

4.4.1 Reassignment vs Self-Mutation

In this chapter's introduction, we saw that augmented assignments (e.g. `a += b`) could have different semantics that their supposed expansion (e.g. `a = a + b`). More specifically, we presented the following example in Python:

Python

```python
foo = [1, 3]
bar = foo
foo += [3]
print(bar)
# Prints "[1, 2, 3]"
```

Python is a reference-based language. Therefore, it makes sense that the `bar` reference be assigned by alias at line 2. However, the fact that `foo` modifies its value is more surprising. The supposed expansion of the augmented assignment is `foo = foo + [3]`, and therefore one could expect that `foo` is simply reassigned to another list. However, Python implements augmented assignments as self-mutating methods.

The subtlety is explicit in the $\mathcal{A}$-calculus:

$\mathcal{A}$-calculus

```
let foo, bar in {
  // Create the list [1, 2]
  alloc foo
  foo <- new <head,tail>
  alloc foo.head, foo.tail
  foo.head <- 1
  foo.tail <- new <head,tail>
  alloc foo.tail.head
  foo.tail.head <- 2

  // Create an alias on foo
  bar &- foo

  // Mutate foo
  foo.tail.tail <- new <head,tail>
  alloc foo.tail.tail.head
  foo.tail.tail.head <- 3

  print(line &- bar)
}
```

4.4.2 Higher-Order Functions

In the C programming language, a function is an object that can only be applied. Unlike other objects, like numbers and pointers, it cannot appear on either side

of an assignment, passed as an argument to another function or be returned from one. Consequently, a C function is said to be a *second-class citizen* in the C language[5]. In contrast, a Swift function can be manipulated as any other object in the language. It can be assigned to a reference, passed as an argument to another function or returned from one. Consequently, a Swift function is said to be a *first-class citizen* in the Swift language.

The terms "first-class function", "higher-order function" and "closure" are often used interchangeably in literature. However, there is a subtle difference that differentiate this notions. A first-class function is merely a function that can be used as any other value. A higher-order function is a function that either accepts another function as argument, or produces one as its return value. Obviously, both notions are tightly linked, as one could hardly design a programming language with higher-order functions without functions being first-class citizen in the first place. A closure is a function that has access to the context in which it has been instantiated, so that it may refer to references from that context. Put differently, the body of a closure can contain free references that are bound outside of the function's body. Such references are said to be *captured* in the function's context. This enables a critical mechanism in functional programming called *partial application*. Consider for example the program below, which features partial application. The function `make_inc` accepts a parameter `i` and produces another function that increments another number by `i`. The returned function is in fact the partial application of the integer addition.

Python

```
def make_inc(i):
  return lambda x: x + i

inc_by_two = make_inc(2)
print(inc_by_two(3))
# Prints "5"
```

Although most programming languages adopting some aspects of functional programming support closures, the latter are not an essential requirement to the support of higher-order (and hence first-class) functions. Note however that a closure can be emulated with a record that encapsulates its context, together with a first-class function whose domain is extended to accept the captured references as argument. This process is often referred to as *defunctionalization* [120]. As mentioned before, the $\mathcal{A}$-calculus only supports thin first-class functions, but allows closures to be defuntionalized by utilizing the record extension.

[5]The reader proficient in C will remark that although functions are second-class, function pointers can be manipulated as first-class citizens.

The program below defunctionalizes the higher-order function presented in the Python example above. The record `fn`, allocated at line 6, contains two fields `i` and `f`. The first corresponds to the reference captured by closure in the original example. It is "captured" by alias at line 9, so as to match Python's capture semantics. At line 15, the field `fn.f` is assigned to a first-class function that reproduces the Python's lambda function behavior. It is eventually applied at line 25.

$\mathcal{A}$-calculus

```
1  let make_inc in {
2    alloc make_inc
3    make_inc <- fun (i) {
4      let fn in {
5        alloc fn
6        fn <- new <i,f>
7
8        // Save the closure's context
9        fn.i &- i
10
11       // Define a first-class function equivalent
12       // accepting the captured references as
13       // additional arguments.
14       alloc fn.f
15       fn.f  <- fun (x, i) { return := x + i }
16
17       return <- fn
18     }
19   }
20   let inc_by_two in {
21     // Create the higher-order function and apply it.
22     alloc inc_by_two
23     inc_by_two <- make_inc(i <- 2)
24     print(
25       line &- inc_by_two.f(x := 3, i &- inc_by_two.i)
     )
26   }
27 }
```

4.5 Summary

We have presented the $\mathcal{A}$-calculus, a formal system for reasoning about memory management that aims at unifying imperative assignment semantics. The model features three assignment operators with distinct yet consistent semantics. An aliasing operator `&-` creates an alias to the object on its right for the variable on its left. A mutation operator $:=$ mutates the object assigned to the variable on its left with a copy of the object assigned to the the variable on its right. A move operator $\leftarrow$ moves the object from the variable on its right to the variable on its left. The following two chapters discuss memory safety, based on the $\mathcal{A}$-calculus's formal operational semantics.

Chapter 5

Dynamic Safety

Manipulating memory through an imperative language is hard. The difficulty stems from the struggle to statically describe the dynamic evolution of memory. In other words, while variables and operations thereupon are expressed at a syntactic level, the state of memory cannot be fully conveyed lexically, requiring to keep a memory model in mind when writing code. Aliasing further worsens the situation by breaking the one-to-one mapping from variables onto memory. As a result, memory errors are consistently reported accounting for about 5% of all software bugs, all programming languages taken together [119, 14].

Memory errors denote situations where the static description diverges from its actual execution. They generally fall into one of these three broad categories:

Invalid memory errors occur when a program attempts to read a value at location that has not yet been assigned to any sensical data (with respect to its semantic type), or that has been freed beforehand.

Memory leak errors occur when a program does not deallocate memory that is no longer accessible by any variable. Unlike other errors, memory leaks generally do not introduce unexpected behaviors directly, but may eventually cause a machine to run out of available memory.

Out-of-bounds access errors occur when a program attempts to access a location that has not been allocated, typically by an improper use of a buffer (e.g. accessing the 11th position of a 10 elements array).

A clear understanding of what constitutes a memory error is paramount to identify faulty programs. This chapter illustrates the use of the $\mathcal{A}$-calculus to formalize memory errors. Two approaches are described. One is based on a post-mortem examination of failed executions to determine the cause of the failure. Another offers to instrument the semantics to model errors explicitly, with a mechanism reminiscent to exceptions [30].

We mentioned in Section 4.3 that we glossed over the specifics of value representation, and therefore assumed memory locations can hold arbitrary large values. This abstraction leaves the $\mathcal{A}$-calculus unable to model out-of-bounds errors due to assignments of oversized values (e.g. assigning a 64-bits integer to a 32-bits location). However, these errors can be detected by the means of type checking. Since our calculus does not support any sort of pointer arithmetic, errors due to out-of-bounds indices cannot be modeled either. Solutions to address this problem usually consist in associating bounds metadata to each pointer and to check them when loading and storing values [53, 58, 105].

5.1 Observing Memory Errors

The operational semantics we presented in Section 4.3 is undefined in the presence of invalid memory accesses. Therefore, a trivial description of an error could be characterized by the mere inability to evaluate a term. More formally, given a term t and an evaluation context C, an invalid memory access in t's evaluation implies there is no l, C' such that $C \vdash t \Downarrow l, C'$ can be proven. Unfortunately such an approach fails to distinguish between faulty programs and non-terminating evaluations. Indeed, if t's evaluation does not terminate (e.g. due to infinite recursion), then we cannot prove $C \vdash t \Downarrow l, C'$ either. As a result, we are left with a system that can only determine the absence of invalid accesses when a program successfully terminates.

A more useful result would be to determine the *cause* of a failed execution. Termination remains obviously undecidable, but an execution whose failure is caused by the evaluation's inability to progress can offer some insights into the error that occurred. Consider for instance the following inference rules, describing the operational semantics of a simple language to express divisions:

$$\textsc{Lit}\ \frac{t \in \mathbb{N}}{t \Downarrow t} \qquad \textsc{Div}\ \frac{t_1 \Downarrow n_1 \qquad t_2 \Downarrow n_2 \qquad n_2 \neq 0}{t_1 \div t_2 \Downarrow n_1 \div n_2}$$

Attempting to evaluate a term $(16 \div 8) \div 0$ will fail because Div cannot satisfy the hypothesis $n_2 \neq 0$. This will result in a term $2 \div 0$ that cannot be evaluated further by any rule, as illustrated below:

$$\dfrac{\dfrac{\dfrac{16 \in \mathbb{N}}{16 \Downarrow 16} \qquad \dfrac{8 \in \mathbb{N}}{8 \Downarrow 8} \qquad 8 \neq 0}{(16 \div 8) \Downarrow 2} \qquad \dfrac{0 \in \mathbb{N}}{0 \Downarrow 0} \qquad 0 \neq 0}{(16 \div 8) \div 0 \Downarrow n}$$

The idea is to "inspect" the evaluation to observe the cause of non-progression. We achieve this goal by using the resolution rule $\longrightarrow$ (c.f. Section 3.3). Let $q = C \vdash t \Downarrow l, C'$ denote the evaluation of a term t in an evaluation context C. We compute the resolution's transitive closure $\longrightarrow^*$, starting from the singleton $Q = \{q\}$ as the set of proof obligations, and an empty substitution table $\sigma = \varnothing$. If the evaluation does not terminate, neither does $\longrightarrow^*$ and we are left with an undecidable result. However, $\{q\}, \sigma \longrightarrow^* Q', \sigma'$ either denotes a successful execution, characterized by $Q' = \varnothing$, or a failed one if Q' is not empty. In this case, the cause of the error can be inferred by looking at the clauses in Q' that could not be satisfied. We illustrate this principle below:

Example 5.1.1: Division by zero

Let us apply resolution on the evaluation of $(16 \div 8) \div 0$. *We start with* $q = \{(16 \div 8) \div 0 \Downarrow n\}$ *as the set of proof obligations, and compute the transitive closure:*

$$\{q\}, \varnothing \longrightarrow^* Q', \sigma'$$

The only rule that applies on q is Div, *therefore we first compute:*

$$\longrightarrow \{16 \div 8 \Downarrow n_1, 0 \Downarrow n_2, n_2 \neq 0\}, \{t_1 \mapsto 16 \div 8, t_2 \mapsto 2\}$$

The second step is to prove Div*'s hypotheses. Proving* $0 \Downarrow n_2$ *requires an application of* Lit*:*

$$\longrightarrow \{16 \div 8 \Downarrow n_1, 0 \in \mathbb{N}, 0 \neq 0\}, \{t_1 \mapsto 16 \div 8, t_2 \mapsto 2, n_2 \mapsto 0\}$$

$0 \in \mathbb{N}$ *is proven trivially. However, we now also know that* $n_2 = 0$, *which therefore leaves* $0 \neq 0$ *unsatisfiable, and indicates the proof is incorrect. Nonetheless the transitive closure is not complete yet, as we still have to prove* $(16 \div 8 \Downarrow n_1)$. *This will eventually determine that* $n_1 = 2$*:*

$$\longrightarrow^* \{0 \neq 0\}, \{t_1 \mapsto 16 \div 8, t_2 \mapsto 2, n_2 \mapsto 0, n_1 \mapsto 2, \ldots\}$$

The process hence results in a non-empty set Q', *symbolizing a failed evaluation. Furthermore, the content of this set reveals that the error is due to* $0 \neq 0$, *which in the above semantics is indicative of a division by zero.*

Note that the approach is sound only because the semantics is defined so that errors prevent progression. Otherwise, evaluation would successfully compute an incorrect value, hiding issues within the execution. This limitation is further discussed in Section 5.2. Another important property of the operational semantics is that it is deterministic. Given a term and an evaluation context, at most one

rule can apply. Consequently, a failing evaluation does not depend on a particular choice of inference rules in the derivation.

5.1.1 Uninitialized or Moved Memory Errors

Uninitialized or moved memory errors occur under any of these three conditions:

- $v = C.\mu.l$ appears in the premise of a rule while $C.\mu.l \in \{\bot, \square\}$. In other words, l is the address of some uninitialized or moved memory.
- $v = C.\mu.l$ appears in the premise of a rule while $l = 0$. In other words, l is the null pointer, denoting an unallocated variable.
- An unallocated variable appears as the left operand of a mutation or move assignment.

In the two first situations, the error is due to value dereferencing. A predicate on $C.\mu.l$ denotes a predicate on a value. If such a value is uninitialized (e.g. $C.\mu.l = \bot$), moved (e.g. $C.\mu.l = \square$) or simply unallocated (e.g. $l = 0$), then evaluation is not defined. This situation may occur while evaluating an assignment's right operand, the callee of a function call, the record of a field dereferencing or the guard of a conditional term.

Just as the behavior of a typical runtime system, the operational semantics of the $\mathcal{A}$-calculus is not defined in the presence of uninitialized or moved memory errors. If either of the above situations occur, no evaluation rule can apply and the program derivation is unable to continue. Therefore we can formalize an uninitialized or moved memory error as follows:

Definition 5.1.1: Uninitialized memory errors

Let $C \vdash t \Downarrow l, C'$ denote the evaluation of a term $t \in \mathcal{AC}$ in a context C. An uninitialized memory error occurs in t's evaluation if and only if:

$$\{C \vdash t \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

such that either

Q contains a clause of the form $v \notin \{\bot, \square\}$ and $v'[/\sigma] = \bot$, or

Q contains a clause of the form $l \neq 0$ and $l[/\sigma] = 0$.

Definition 5.1.2: Moved memory errors

Let $C \vdash t \Downarrow l, C'$ denote the evaluation of a term $t \in \mathcal{AC}$ in a context C. A moved memory error occurs in t's evaluation if and only if:

$$\{C \vdash t \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

such that Q contains a clause of the form $v = v'$ and $v'[\sigma] = \square$.

Let us illustrate the above definitions with a concrete example. Recall the rule ER-Mut, which describes the semantics of the mutation operator. The value $C'.\mu.m$ in the premise represents the content stored at location m, which is the value to which the right operand (i.e. t) evaluates. If t refers to uninitialized memory, then $C \vdash t \Downarrow m, C'$ in the premise produces a location m, such that $v = C'.\mu.m = \bot$ and $v \notin \{\bot, \square\}$ is not satisfied.

Example 5.1.2: Uninitialized memory errors

Consider the following faulty program:

$$\texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ x, y\ ;\ x \leftarrow y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \bot \\ y \mapsto l_2 & l_2 \mapsto \bot \end{array}$$

In other words, both x and y are allocated but neither is bound to initialized memory. Then the assignment's evaluation is described by the following excerpt of derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash y \Downarrow l_2, C} \quad l_2 \in \mathrm{dom}(C.\mu) \quad \bot = C.\mu.l_2 \quad \bot \notin \{\bot, \square\} \quad \cdots}{C \vdash x \leftarrow y \Downarrow l, C'}$$

The derivation fails because the premise $\bot \notin \{\bot, \square\}$ cannot be satisfied. This means that the transitive closure of the resolution

$$\{C \vdash x := y \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

computes a non-empty set of clauses Q such that $(\bot \notin \{\bot, \square\}) \in Q$. The error is therefore detected.

5.1.2 Use After Free Errors

Freed locations are removed from the memory table (c.f. E-Del). Hence, determining a use after free error simply consists in checking that $l \in \mathrm{dom}(C.\mu)$ every time a clause of the form $v = C.\mu.l$ appears in a premise.

Note however that observing $l = 0$ indicates an uninitialized memory, rather than a use after free error. Variables only can be bound to the null pointer, between the moment they are declared until they are allocated. Furthermore, memory allocations never produce the null pointer 0. Therefore, observing the null pointer cannot indicate a use after free error. This particular case is handled in the formal description by constraining $l \neq 0$.

Definition 5.1.3: Use after free errors

Let $C \vdash t \Downarrow l, C'$ denote the evaluation of a term $t \in \mathcal{AC}$ in a context C. A use after free error occurs in t's evaluation if and only if:

$$\{C \vdash t \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

such that Q contains a clause of the form $l \in \mathrm{dom}(C.\mu)$ and $l[/\sigma] \notin \mathrm{dom}(C.\mu[/\sigma]) \implies l[/\sigma] \neq 0$.

Example 5.1.3: Use after free errors

Consider the following faulty program:

$$\texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ x, y\ ;\ \texttt{del}\ y\ ;\ x \leftarrow y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \bot \\ y \mapsto l_2 & \end{array}$$

In other words, x is allocated and bound to uninitialized memory, while y is bound to freed memory. Then the assignment's evaluation is described by the following excerpt of derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash y \Downarrow l_2, C} \quad l_2 \in \mathrm{dom}(C.\mu) \quad \cdots}{C \vdash x \leftarrow y \Downarrow l, C'}$$

The derivation fails because $l_2 \notin \mathrm{dom}(C.\mu)$ in the premise. This means that the transitive closure of the resolution

$$\{C \vdash x := y \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

computes a non-empty set of clauses Q such that $(l_2 \notin C.\mu) \in Q$, and $l_2 \neq 0$. The error is therefore detected.

5.1.3 Doubly Freed Memory

A doubly freed memory error occurs upon deallocating a memory location after it has already been deallocated once. In real systems, this often opens the door to security exploits, as it may corrupt the runtime system's memory bookkeeping [38]. In the $\mathcal{A}$-calculus, a doubly freed memory error occurs when `del` x is evaluated on a variable x that does no longer refers to an allocated location. Looking at E-Del, we can see that the evaluation cannot proceed if the premise $l \in \mathrm{dom}(C'.\mu)$ if unsatisfiable. Like for use after free errors, observing $l = 0$ indicates uninitialized memory rather than a doubly freed memory error. This means that the term representing the memory location to deallocate actually refers to the null pointer, which obviously cannot have been allocated.

Definition 5.1.4: Doubly freed memory errors

Let $C \vdash t \Downarrow l, C'$ denote the evaluation of a term $t \in \mathcal{AC}$ in a context C. A doubly freed memory error occurs in t's evaluation if and only if:

$$\{C \vdash t \Downarrow l, C'\}, \varnothing \longrightarrow^* Q, \sigma$$

such that Q contains a clause of the form $l \in \mathrm{dom}(C.\mu)$ and $l[/\sigma] \notin \mathrm{dom}(C.\mu[/\sigma]) \implies l[/\sigma] \neq 0$.

Example 5.1.4: Doubly freed memory error

Consider the following faulty program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ \texttt{del}\ x\ ;\ \texttt{del}\ x$$

For the sake of conciseness, we skip grayed out terms to focus solely on the sequence of deallocations, and assume it is evaluated in a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \bot \end{array}$$

In other words, x has been deallocated. The doubly freed memory error is illustrated in the following derivation:

$$\dfrac{\dfrac{\cdots \quad l_1 \in \mathrm{dom}(C.\mu)}{C \vdash \texttt{del}\ x \Downarrow l_1, C'}\ \text{E-Del} \qquad \dfrac{\cdots \quad l_1 \in \mathrm{dom}(C'.\mu)}{C' \vdash \texttt{del}\ x \Downarrow l, C''}\ \text{E-Del}}{C \vdash \texttt{del}\ x\ ;\ \texttt{del}\ x \Downarrow l, C''}$$

The intermediate evaluation context is defined by $C' = C[\mu \mapsto \mu|_{l_1}]$. The derivation fails because $l_1 \notin \mathrm{dom}(C'.\mu)$. This means that the transitive closure of the resolution

$$\{C \vdash \texttt{del}\ x\ ;\ \texttt{del}\ x \Downarrow l, C''\}, \varnothing \longrightarrow^* Q, \sigma$$

computes a non-empty set of clauses Q such that $(l_1 \in \mathrm{dom}(C.\mu)) \in Q$, and $l_1 \neq 0$. The error is therefore detected.

5.2 Signaling Memory Errors

Another approach to detect memory issues is to instrument the semantics, in order to represent execution failures explicitly. More formally, given a term t and an evaluation context C, an instrumented semantics evaluates a faulty program as $C \vdash t \Downarrow_X \xi, C'$, where ξ is a value indicating the occurence of an error.

While this technique requires a tedious modification of the operational semantics, in order to properly propagate the error to the root of the proof tree, it is more powerful than the observation strategy. First, it can model failures that do not prevent progression. For instance, a memory leak typically does not prevent a program to be successfully evaluated, and therefore cannot be detected by observing the unsatisfiable clauses of a failed execution. Secondly, it allows a simpler description of the error. While the evaluation of a faulty program (with respect to invalid memory accesses) is guaranteed to eventually fail, causes of such a failure might indicate multiple errors. Distinguishing between those and/or tracing a particular one back to its origin becomes a difficult task, requiring careful inspection of the substitutions that result from the resolution's application. Evidently, the difficulty of this exercise increases with the size of the program, as the set of substitutions grows linearly with the number of subterms. Consider for instance the following term:

$$\texttt{let}\ x, y\ \texttt{in}\ x \leftarrow y$$

Assuming the term is evaluated in an empty context, the evaluation of the assign-

ment is described by the following excerpt of derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash y \Downarrow C.\pi.y, C} \quad C.\pi.y \notin \mathrm{dom}(C.\mu) \quad \dfrac{\vdots}{C \vdash x \Downarrow C.\pi.x, C} \quad C.\pi.x \neq 0 \quad \cdots}{C \vdash x \leftarrow y \Downarrow l, C'}$$

Two uninitialized memory errors can be observed. One is provoked by y's dereferencing and indicated by the unsatisfiability of $C.\pi.y \notin \mathrm{dom}(C.\mu)$. The second is provoked by x's dereferencing and indicated by the unsatisfiability of $C.\pi.x \neq 0$. Computing $\{C \vdash \texttt{let}\ x, y\ \texttt{in}\ x \leftarrow y \Downarrow l, C'\}, \varnothing \longrightarrow^* Q', \sigma'$ will produce a set Q' with both unsatisfiable clauses. Upon meticulous inspection of σ', we can indeed infer that x was unallocated, but only after tracing all substitutions resulting from two applications of E-Let, one of E-Mut and on of E-Var.

This problem has consequences in real systems languages as well, and greatly contributes to the difficulty of debugging. Indeed, tracing back an error to its origin in a large program might involve a rigorous review of dozens of instructions executed *after* the actual point at which the error occurred. As a result, unless the error is known to be recoverable, one desirable property of a runtime system is that it *fails fast*. In other words, evaluation should report errors as soon as possible. One way to adopt this behavior is to modify potentially failing rules so that they deviate from the normal execution's path immediately after an error is detected. Such a mechanism can be implemented with *exceptions* [30]. Traditionally, an exception is a special signal a (sub)system can emit when it encounters an unexpectedly abnormal situation. This signal might also contain information about the nature of the error. For instance, an Intel CPU emits an interrupt together with an error a code when presented to a division by zero [82]. A piece of software, typically the operating system, may then catch this signal, read the error code and hopefully recover from the error. In other words, exceptions represent an additional operation outcome which deviate from the normal course of execution.

Example 5.2.1: Division by zero

Consider the following inference rules, describing the operational semantics of a simple language to express divisions. The value ξ_0 is an exception denoting a division by zero.

$$\textsc{Lit}\ \dfrac{t \in \mathbb{N}}{t \Downarrow t} \qquad \textsc{Div}\ \dfrac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2 \quad n_1, n_2 \in \mathbb{N} \quad n_2 \neq 0}{t_1 \div t_2 \Downarrow n_1 \div n_2} \qquad \textsc{Div-0}\ \dfrac{t_2 \Downarrow n_2 \quad n_2 = 0}{t_1 \div t_2 \Downarrow \xi_0}$$

Evaluating a term $(16 \div 8) \div 0$ will obviously result in an error. But instead of just stopping after unsuccessfully attempting to apply Div *on $2 \div 0$,*

the evaluation keeps progressing by applying Div-0 *instead. Therefore the system eventually proves* $(16 \div 8) \div 0 \Downarrow \xi_0$.

We propose to instrument the $\mathcal{A}$-calculus' semantics to model memory errors as exceptions. We formalize this instrumentation in the form of another evaluation operator $\Downarrow_X$ for which we redefine each inference rule. Conclusions are either of the form $C \vdash t \Downarrow_X l, C'$, mimicking the normal evaluation, or of the form $C \vdash t \Downarrow_X \xi, C'$ to describe an exceptional outcome. We define the set of exception values as follows:

Definition 5.2.1: Exception values

The set of exception values $\mathcal{X}$ is defined by $\mathcal{X} = \{\xi_\perp, \xi_\square, \xi_{uaf}, \xi_{df}\}$ where:

- $\xi_\perp$ *denotes an uninitialized memory error,*
- $\xi_\square$ *denotes a moved memory error,*
- ξ_{uaf} *denotes a use after free error,*
- ξ_{df} *denotes a doubly freed memory error, and*
- ξ_{lk} *denotes a memory leak.*

We call the semantics of $\Downarrow_X$ the "safe semantics" of the $\mathcal{A}$-calculus, while we refer to that introduced in Section 4.3 as its "unsafe semantics".

5.2.1 Uninitialized or Moved Memory Errors

As mentioned in Section 5.1.1, uninitialized or moved memory errors are generally due to a failure in value dereferencing. Such a situation may occur while evaluating assignments, function callees, field dereferencing and conditional terms. As a result, the instrumentation process is extremely similar for every rule. Three cases should be considered:

1. The term to dereference is bound to the null pointer.
2. The term to dereference is bound to uninitialized memory.
3. The term to dereference is bound to moved memory.

The first two cases raise a $\xi_\perp$ exception, while the last raises $\xi_\square$. For instance, ER-Field, that describes the dereferencing of a record field, is instrumented as following:

Uninitialized or moved memory in field dereferencing

$$\frac{C \vdash t \Downarrow_X 0, C'}{C \vdash t.x \Downarrow_X \xi_\perp, C'}\ \text{EX-Field-0} \qquad \frac{C \vdash t \Downarrow_X l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad C'.\mu.l = \perp}{C \vdash t.x \Downarrow_X \xi_\perp, C'}\ \text{EX-Field-}\perp$$

$$\frac{C \vdash t \Downarrow_X l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad C'.\mu.l = \square}{C \vdash t.x \Downarrow_X \xi_\square, C'}\ \text{EX-Field-}\square$$

Example 5.2.2: Uninitialized memory in field derefencing

Consider the following faulty program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x.y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the dereferencing, and assume it is evaluated in the context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \perp \end{array}$$

In other words, x is allocated but refers to uninitialized memory. Then the term's evaluation is described by the following excerpt of derivation:

$$\frac{\dfrac{\vdots}{C \vdash x \Downarrow_X l_1, C} \quad l_1 \in \mathrm{dom}(C.\mu) \quad \perp = \perp}{C \vdash x.y \Downarrow_X \xi_\perp, C}\ \text{EX-Field-}\perp$$

In mutation and move assignments, another type of uninitialized memory error can occur if the left operand is not allocated. In this situation, the left operand evaluates to the null pointer (i.e. 0), and should be reported as an uninitialized memory error. For instance, ER-Mut, that describes the mutation assignment, is instrumented as below:

Uninitialized or moved memory in mutation assignments

$$\text{EX-Mut-R0}\quad \frac{C \vdash t \Downarrow_X 0, C'}{C \vdash x := t \Downarrow_X \xi_\perp, C'}$$

$$\text{EX-Mut-R}\perp\quad \frac{C \vdash t \Downarrow_X m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v = \perp}{C \vdash x := t \Downarrow_X \xi_\perp, C'}$$

$$\text{EX-Mut-R}\square\quad \frac{C \vdash t \Downarrow_X m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v = \square}{C \vdash x := t \Downarrow_X \xi_\perp, C'}$$

$$\text{EX-Mut-L0}\quad \frac{C \vdash t \Downarrow_X m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\perp, \square\} \quad C'.\pi.x = 0}{C \vdash x := t \Downarrow_X \xi_\perp, C'}$$

The first three rules check memory errors on the right operand, while EX-Mut-L0 applies when the left operand refers to the null pointer. Note that the rule does not compute a copy of v, and evaluates the left operand immediately after the first one. This differs from ER-Mut, in which the left operand is evaluated after v is copied. The change is in fact irrelevant because copies only *add* information to the memory table, and therefore cannot influence the result of a dereferencing. This also means that the instrumentation of the move assignment is actually identical to that of the mutation assignment.

Example 5.2.3: Uninitialized memory exception in an assignment

Consider the following faulty program:

$$\texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ y\ ;\ y \leftarrow 2\ x := y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in the context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto 0 & l_1 \mapsto 2 \\ y \mapsto l_1 & \end{array}$$

In other words, x is declared but unallocated and y is bound to the value 2. Then the assignment's evaluation is described by the following excerpt of

derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash y \Downarrow_X l_1, C} \quad l_1 \in \mathrm{dom}(C.\mu) \quad \cdots \quad C.\pi.x = 0}{C \vdash x := y \Downarrow_X \xi_\perp, C}\ \text{EX-MUT-L0}$$

5.2.2 Use After Free Errors

Use after free errors are characterized by an attempt to dereference a (non-null) memory location that is no longer defined in the memory table. Therefore it suffices to instrument rules where a clause of the form $C.\mu.l$ appears in the premise. For instance, E-MOVE is instrumented as follows:

Use after free in move assignments

$$\text{EX-MOVE-UAF}\quad \dfrac{C \vdash t \Downarrow_X m, C' \qquad m \notin \mathrm{dom}(C'.\mu) \qquad m \neq 0}{C \vdash x \leftarrow t \Downarrow_X \xi_{uaf}, C'}$$

Not only does the rule check that the location is undefined in $C'.\mu$, but also that it is not the null pointer. Otherwise we should instead raise $\xi_\perp$, to denote an uninitialized memory error, as described in the previous section.

Example 5.2.4: Use after free error in an assignment

Consider the following faulty program:

$$\texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ y\ ;\ \texttt{del}\ y\ ;\ x \leftarrow y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in the context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto 0 & \\ y \mapsto l_1 & \end{array}$$

In other words, x is declared but unallocated and y has been freed. Then the assignment's evaluation is described by the following excerpt of derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash y \Downarrow_X l_1, C} \quad l_1 \notin C.\mu \quad l_1 \neq 0}{C \vdash x \leftarrow y \Downarrow_X \xi_{uaf}, C}\ \text{EX-MOVE-UAF}$$

5.2.3 Doubly Freed Memory Errors

Doubly freed memory errors are a specific kind of use after free errors that only occur when deallocating a freed memory location. In other words, doubly freed memory errors occur when evaluating a term of the form `del` t, where t's evaluation produces a (non-null) memory location that is not defined in the memory table. Therefore the only rule to instrument is E-Del.

Doubly freed memory

$$\text{EX-Del-DF}\ \frac{C \vdash t \Downarrow_X l, C' \qquad l \notin \mathrm{dom}(C'.\mu) \qquad l \neq 0}{C \vdash \texttt{del}\ t \Downarrow_X \xi_{df}, C'}$$

Example 5.2.5: Doubly freed memory error

Consider the following faulty program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ \texttt{del}\ x\ ;\ \texttt{del}\ x$$

For the sake of conciseness, we skip grayed out terms to focus solely on the second deallocations, and assume it is evaluated in the context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & \end{array}$$

The doubly freed memory error is illustrated in the following excerpt of derivation:

$$\dfrac{\dfrac{\vdots}{C \vdash x \Downarrow_X l_1, C} \qquad l_1 \notin \mathrm{dom}(C.\mu) \quad l_1 \neq 0}{C \vdash \texttt{del}\ x \Downarrow_X \xi_{df}, C}\ \text{EX-Del-DF}$$

5.2.4 Memory Leaks

Memory leaks are more difficult to detect because they are not caused by an unsatisfiable premise. Moreover, since the $\mathcal{A}$-calculus assumes an unbounded memory, no amount of memory leaks can ever cause evaluation to fail. Hence, we cannot use the same approach as that we have presented so far. Instead, detecting a memory leak requires additional instrumentation. The goal is to inspect the memory table to determine whether all non-freed locations are still reachable by at least

one alive variable in the program[1] (e.g. a variable in the pointer table's domain). Then, if a leak is detected, the evaluation is forced to fail. A first step toward the formalization of this idea is to define *memory accessibility*.

Definition 5.2.2: Local memory accessibility

Given an evaluation context C, a memory location l is said locally accessible *in C, written $C \rightsquigarrow l$, if there exists a variable in* $\mathrm{dom}(C.\pi)$ *that refers to it, or if it is referred by the field of a record stored at an accessible memory location in C. More formally, a memory location $l \in \mathrm{dom}(C.\mu)$ is locally accessible if and only if:*

- $\exists x \in \mathrm{dom}(C.\pi), C.\pi.x = l$, *or*
- $\exists m \in \mathrm{dom}(C.\pi)$ *such that m is accessible and $r = C.\pi.m$ is a record such that $\exists x \in \mathrm{dom}(r), r.x = l$.*

Example 5.2.6: Local memory accessibility

Consider the following evaluation context:

$$C = \begin{array}{c|l} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto v_a \\ y \mapsto l_2 & l_2 \mapsto \langle z \mapsto l_3 \rangle \\ & l_3 \mapsto v_b \\ & l_4 \mapsto v_c \end{array}$$

The location l_1 is locally accessible by x because $C.x = l_1$. The location l_3 is locally accessible by y, because the record r stored at l_2 is defined so that $r.z = l_3$, and l_2 is locally accessible by y. The location l_4 however is not locally accessible.

One could be tempted to simply check whether the non-freed memory locations in $\mathrm{dom}(C.\mu)$ are locally accessible in C, after each evaluation step. Consider for instance this program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \leftarrow 4\ ;\ \texttt{let}\ y\ \texttt{in}\ \texttt{alloc}\ y\ ;\ y \leftarrow 2\ ;\ x\ \texttt{\&-}\ y$$

The last assignment's evaluation yields an evaluation context as follows:

$$C = \begin{array}{c|l} \pi & \mu \\ \hline x \mapsto l_2 & l_1 \mapsto 4 \\ y \mapsto l_2 & l_2 \mapsto 2 \end{array}$$

[1] The reader will note this strategy is reminiscent to tracing garbage collection [142].

Because of the reassignment of x to l_2, l_1 is no longer locally accessible in C, which would indicate a leak. This approach is unfortunately unsound, because a pointer table's domain is only defined for the *current* frame (hence the phrase *locally* accessible). Indeed, if a name is shadowed in $C.\pi$, the location to which it refers cannot be retrieved. Therefore, whether or not a particular memory location is accessible in an ancestor frame is undecidable. Consider for example this slightly altered version of the program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \leftarrow 4\ ;\ \texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ y\ ;\ y \leftarrow 2\ ;\ x\ \texttt{\&-}\ y$$

Now the last assignment's evaluation yields the following context:

$$C = \begin{array}{r|l} \pi & \mu \\ \hline x \mapsto l_2 & l_1 \mapsto 4 \\ y \mapsto l_2 & l_2 \mapsto 2 \end{array}$$

Once again, the memory location l_1 is not locally accessible in C. However, there exists a variable that refers to it in the previous frame, as the outer x has not been reassigned. In other words, l_1 is locally accessible in the previous frame. There are two ways to solve this issue:

Use α-conversion The goal of this approach is to avoid name shadowing by renaming identifiers so that they are unique across the entire program. For instance, the term $\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x$ is α-equivalent (i.e. equivalent up to α-conversion) to $\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ x_0\ \texttt{in}\ \texttt{alloc}\ x_0$. The reader will recall that we have already employed this method to deal with parameters in E-Call (c.f. Section 4.3.2).

Formalize frames A second approach is to modify evaluation contexts so that they keep track of each frame's pointer map. Then, *absolute* memory accessibility can be checked by satisfying local accessibility in at least one frame.

5.2.5 Memory Leaks with α-conversion

Using α-conversion is the easiest approach, since it essentially removes name shadowing. Consequently, local accessibility is sufficient to determine memory leaks, because a pointer map necessarily "sees" all variables. This leads to the following definition:

Definition 5.2.3: Memory Leak

Let $C \vdash t \Downarrow l, C'$ be the evaluation of a term t in an evaluation context C. A memory leak is characterized by the existence of a location $l \in \mathrm{dom}(C'.\mu)$ such that $C' \not\rightsquigarrow l$.

Signaling memory leaks requires to instrument all rules, so that they check for the existence of a leaked location in the resulting context. At a meta-level, it corresponds to this transformation:

$$\frac{\Gamma}{C \vdash t \Downarrow l, C'} \quad \rightsquigarrow \quad \begin{cases} \dfrac{\Gamma \qquad \forall l \in \mathrm{dom}(C'.\mu), C' \rightsquigarrow l}{C \vdash t \Downarrow_X l, C'} \\ \dfrac{\Gamma \qquad \exists l \in \mathrm{dom}(C'.\mu), C' \not\rightsquigarrow l}{C \vdash t \Downarrow_X \xi_{lk}, C'} \end{cases}$$

For instance, the evaluation of aliasing assignments is instrumented as follows:

Memory leak in aliasing assignments

$$\textsc{Ex-Alias}\ \frac{C \vdash t \Downarrow_X l, C' \qquad \forall l \in \mathrm{dom}(C'.\mu), C' \rightsquigarrow l}{C \vdash x \,\&\!-\, t \Downarrow_X l, C'[\pi.x \mapsto l]}$$

$$\textsc{Ex-Alias-LK}\ \frac{C \vdash t \Downarrow_X l, C' \qquad \exists l \in \mathrm{dom}(C'.\mu), C' \not\rightsquigarrow^* l}{C \vdash x \,\&\!-\, t \Downarrow_X \xi_{lk}, C'}$$

Example 5.2.7: Memory leak

Consider the following program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \leftarrow 2$$

Assume C and C' are given as follows:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 4 \end{array} \quad \textit{and} \quad C' = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto 4 \\ & l_2 \mapsto 2 \end{array}$$

Then the term's evaluation is described by the following excerpt of deriva-

tion:

$$\dfrac{\dfrac{\dfrac{\vdots}{C[\pi.x_0 \mapsto 0] \vdash x_0 \leftarrow 2 \Downarrow l_2, C'}}{C[\pi.x_0 \mapsto 0] \vdash \texttt{alloc}\ x_0\ ;\ x_0 \leftarrow 2 \Downarrow l_2, C'}}{C \vdash \texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ; x \leftarrow 2 \Downarrow l_2, C'}$$

The memory location l_2 corresponds to the allocation made for x's redeclaration (renamed x_0), and is no longer accessible in C', which indicates a memory leak.

5.2.6 Memory Leaks with Frame-Aware Contexts

An alternative approach to model memory leaks consists in modeling the stack of pointer tables first need to redefine evaluation contexts in that direction.

Definition 5.2.4: Frame-aware evaluation context

Given $\mathbb{V}$ the set of semantic values, a frame-aware evaluation context $\widetilde{C}$ is record of a composite type $\langle \pi, \mu \rangle$, where:

$\pi : (\mathbb{X} \rightharpoonup \mathbb{N})^*$ *is a sequence of pointer tables, starting with that of the youngest frame, and*

$\mu : \mathbb{N} \rightharpoonup \mathbb{V}$ *is a memory table.*

A frame-aware evaluation context should always define at least one frame. More formally, Let $\widetilde{C}$ be context, $\widetilde{C}.\pi \neq \epsilon$ is an invariant. We also adapt the table notation for frame-aware evaluation contexts, and use dotted lines to distinguish between frames. Older frames are represented below younger ones. Consider for instance the following expression:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{alloc}\ x\ ;\ x \leftarrow 4\ ;\ \texttt{let}\ x, y\ \texttt{in}\ \texttt{alloc}\ y\ ;\ y \leftarrow 2\ ;\ x\ \texttt{\&-}\ y$$

The following table represents the frame-aware evaluation context that stems from the evaluation of the aliasing assignment at the end of the program:

$\widetilde{C} =$

π	μ
	$l_1 \mapsto 4$
$x \mapsto l_1$	$l_2 \mapsto 2$
$x \mapsto 0$	
$x \mapsto l_2$	
$y \mapsto l_2$	

Each variable declaration creates a new frame in the evaluation context. Recall that terms of the form $\texttt{let}\ x_1 \ldots x_n\ \texttt{in}\ t$ are sugars for for $\texttt{let}\ x_1\ \texttt{in}\ \ldots \texttt{let}\ x_n\ \texttt{in}\ t$, which explains the presence of the last three frames. The oldest frame is necessarily empty, as it corresponds to the situation before the first variable's declaration is evaluated. Finally, notice that the mapping for the first x's declaration is still defined in $\widetilde{C}$, even if x is being shadowed. Consequently, it can be observed that the memory location l_1 is in fact *absolutely* accessible. We define such a notion formally.

Definition 5.2.5: Absolute memory accessibility

Given a frame-aware evaluation context $\widetilde{C}$, a memory location l is said absolutely accessible *in $\widetilde{C}$, written $\widetilde{C} \rightsquigarrow^* l$, if there exists a pointer table p in $\widetilde{C}.\pi$ such that l is locally accessible in π.*

Example 5.2.8: Absolute memory accessibility

Consider the following frame-aware evaluation context:

$$\widetilde{C} = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \langle z \mapsto l_2 \rangle \\ \hdashline y \mapsto l_2 & l_2 \mapsto v_a \\ x \mapsto l_4 & l_3 \mapsto v_b \\ & l_4 \mapsto v_c \end{array}$$

The locations l_2 and l_4 are locally accessible in the last frame, and therefore absolutely accessible as well. Although the location l_1 is locally inaccessible in the last frame, it is globally accessible by x from the first one. The location l_3 however is neither locally nor absolutely accessible.

Obviously, the semantics must be modified to accommodate frame-aware evaluation contexts. This can be done systematically by applying two modifications. First, references to $C.\pi.x$ must be replaced with $\widetilde{C}.\pi_1.x$. In less formal terms, $\widetilde{C}.\pi_1$ refers to the pointer table of the youngest frame. For example, E-Var is modified as below:

Frame-aware variable dereferencing

$$\text{EFA-Var}\ \frac{x \in \mathrm{dom}(\widetilde{C}.\pi_1)}{\widetilde{C} \vdash x \Downarrow_{FA} \widetilde{C}.\pi_1.t, C}$$

Secondly and more importantly, rules describing the semantics of scope de-

limiting terms must also be altered to represent the creation and removal of a new frame. The modification consists in creating a new frame in which subterms are evaluated. This newly created frame is then removed from the resulting context. The process is described by two functions push and pop:

$$\text{push}(p_1\overline{p}) = p_1 p_1 \overline{p} \qquad \text{pop}(p_1 p_2 \overline{p}) = p_1 \overline{p}$$

The function push duplicates the current frame and prepends it to the sequence of frames. As a result, the newly pushed frame is a perfect copy of its predecessor. This means that popping is achieved by removing the parent of the current frame. For example, EC-Cond-T and EC-Cond-F are modified as follows:

Frame-aware conditional terms

EFA-Cond-T

$$\frac{\widetilde{C} \vdash t_1 \Downarrow_{FA} l_1, \widetilde{C}' \quad \widetilde{C}'.\mu.l_1 = \texttt{true} \quad \widetilde{C}'[\pi \mapsto \text{push}(\widetilde{C}'.\pi)] \vdash t_2 \Downarrow_{FA} l_2, \widetilde{C}''}{\widetilde{C} \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_{FA} l_2, \widetilde{C}''[\pi \mapsto \text{pop}(\widetilde{C}''.\pi)]}$$

EFA-Cond-F

$$\frac{\widetilde{C} \vdash t_1 \Downarrow_{FA} l_1, \widetilde{C}' \quad \widetilde{C}'.\mu.l_1 = \texttt{false} \quad \widetilde{C}'[\pi \mapsto \text{push}(\widetilde{C}'.\pi)] \vdash t_3 \Downarrow_{FA} l_2, \widetilde{C}''}{\widetilde{C} \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_{FA} l_2, \widetilde{C}''[\pi \mapsto \text{pop}(\widetilde{C}''.\pi)]}$$

Name shadowing shall also be taken into account. Fortunately this can be done in the same way as in the regular semantics, by the means of the unset function. For example, E-Let is modified as follows:

Frame-aware variable declarations

EFA-Let

$$\frac{p_1\overline{p} = \text{push}(\widetilde{C}.\pi) \quad \widetilde{C}[\pi \mapsto p_1[x \mapsto 0]\overline{p}] \vdash t \Downarrow_{FA} l, \widetilde{C}' \quad p_1'\overline{p}' = \widetilde{C}'.\pi}{C \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow_{FA} l, \widetilde{C}'[\pi \mapsto \text{pop}(\text{unset}(x, p_1', p_1)\overline{p}')]}$$

Let us break down this rule in several steps:

1. The premise $p_1\overline{p} = \text{push}(\widetilde{C}.\pi)$ computes the creation of a new frame p_1.
2. p_1 is updated so that it maps x to the null pointer, therefore introducing x in the new scope, and potentially shadowing x in $\widetilde{C}.\pi_1$.
3. The update is prepended to $\overline{p}$, which actually corresponds to $\widetilde{C}.\pi$ (i.e. the sequence of pointer tables before the creation of a new frame in step 1).
4. The evaluation of t is carried out, producing a new context $\widetilde{C}'$ from which a frame has to be popped.

5. The youngest frame in $\widetilde{C}'$ is fed to the function pop to carry over the modifications on non-shadowed names.

In other words, steps 1 to 3 push a new frame in which the declared identifier is defined. Step 4 is the evaluation of the declaration's body and step 5 restores the evaluation context.

Example 5.2.9: Frame-aware name shadowing

Consider the following term

$$\texttt{let } x, y \texttt{ in}$$
$$\quad \texttt{alloc } x, y \texttt{ ; let } x \texttt{ in alloc } x \texttt{ ; } x \leftarrow 42 \texttt{ ; } y \texttt{ \&- } x$$

We skip grayed out parts and focus solely x's second declaration. Let C be a frame-aware evaluation context defined as below:

$$\widetilde{C} = \begin{array}{c|c} \pi & \mu \\ \hline & l_1 \mapsto \bot \\ x \mapsto 0 & l_2 \mapsto \bot \\ x \mapsto l_1 & \\ y \mapsto l_2 & \end{array}$$

In other words, x and y are allocated in the last frame. Note that the frame above is introduced by x's first declaration, and that the oldest frame correspond to the initial situations. The last declaration's evaluation is described below. For spatial reasons, subparts of the derivation are described separately.

$$\frac{(1) = \text{push}(\widetilde{C}.\pi) \quad (2) \vdash \cdots \Downarrow_{FA} l, \widetilde{C}' \quad p_1'\overline{p} = \widetilde{C}'.\pi}{\widetilde{C} \vdash \texttt{let } x \texttt{ in } \cdots \Downarrow_{FA} l, (3)}$$

(1) *corresponds to the pushing of a new frame, and is given below.*

$$\begin{aligned}(1) &\rightsquigarrow p_1\overline{p} \\ &\rightsquigarrow \{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto 0\}\varnothing \\ &\rightsquigarrow \{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto 0\}\varnothing\end{aligned}$$

(2) *corresponds to the shadowing of x in the new frame, before the evaluation of the declaration's body:*

$$\begin{aligned}(2) &\rightsquigarrow \widetilde{C}[\pi \mapsto p_1[x \mapsto 0]\overline{p}] \\ &\rightsquigarrow \widetilde{C}[\pi \mapsto \{x \mapsto 1, y \mapsto l_2\}[x \mapsto 0]\{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto 0\}\varnothing] \\ &\rightsquigarrow \widetilde{C}[\pi \mapsto \{x \mapsto 0, y \mapsto l_2\}\{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto 0\}\varnothing]\end{aligned}$$

(3) *corresponds to the frame popping:*

$$
\begin{aligned}
(3) &\rightsquigarrow \widetilde{C}'[\pi \mapsto \mathrm{pop}(\mathrm{unset}(x, p_1', p_1)\overline{p}')] \\
&\rightsquigarrow \widetilde{C}'[\pi \mapsto \mathrm{pop}(\mathrm{unset}(x, \{x \mapsto l_3, y \mapsto l_3\}, \{x \mapsto l_1, y \mapsto l_2\})\overline{p}')] \\
&\rightsquigarrow \widetilde{C}'[\pi \mapsto \mathrm{pop}(\{x \mapsto l_1, y \mapsto l_3\}\overline{p}'] \\
&\rightsquigarrow \widetilde{C}'[\pi \mapsto \mathrm{pop}(\{x \mapsto l_1, y \mapsto l_3\}\{x \mapsto l_1, y \mapsto l_2\}\{x \mapsto 0\}\varnothing] \\
&\rightsquigarrow \widetilde{C}'[\pi \mapsto \{x \mapsto l_1, y \mapsto l_3\}\{x \mapsto 0\}\varnothing]
\end{aligned}
$$

The resulting context is given as follows:

$$
(3) = \begin{array}{c|c}
\pi & \mu \\
\hline
 & l_1 \mapsto \bot \\
x \mapsto 0 & l_2 \mapsto \bot \\
x \mapsto l_1 & l_3 \mapsto 42 \\
y \mapsto l_3 & l_4 \mapsto \square
\end{array}
$$

5.2.7 Forwarding Exceptions

So far we have only altered rules that may encounter memory errors, so that they generate exceptions if they do. However, we have not yet dealt with errors that may occur during the evaluation of a subterm. For instance, if an error occurs in a function call whose result is computed in an assignment, then the error signaled in the function call must be forwarded by the assignment's evaluation. This means that *all* rules should be defined to forward *all* possible errors occurirng in the evaluation of their subterms. Fortunately, this tedious task can be avoided by altering the resolution rule (c.f. Section 3.3). The advantage of this approach is that it is agnostic of the exact evaluation semantics, and operates at a meta-level instead.

The goal is to replace the rule so that evaluation either stops as soon as an exception appears in the proof obligations, or continues if no errors have been signaled. More formally, this is described as follows:

$$
\frac{(C \implies q') \in I \qquad \exists \sigma', q[\sigma'] \equiv q'[\sigma'] \qquad \exists l \in \mathrm{dom}(\sigma'), \sigma'.l \in \mathcal{X}}{\{q\} \cup Q, \sigma \longrightarrow \sigma'.l, \sigma'}
$$

$$
\frac{(C \implies q') \in I \qquad \exists \sigma', q[\sigma'] \equiv q'[\sigma'] \qquad \nexists l \in \mathrm{dom}(\sigma'), \sigma'.l \in \mathcal{X}}{\{q\} \cup Q, \sigma \longrightarrow Q[\sigma'] \cup C[\sigma'], \sigma \uplus \sigma'}
$$

In plain English, this rule states that if the resolution of a proof obligation q leads

to a substitution table σ' that maps a variable l to an exception, then the resolution concludes the exception.

Example 5.2.10: Forwarding

Consider the following term:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{del}\ x\ ;\ \texttt{alloc}\ x$$

A uninitialized memory error will occur during this term's execution, specifically when the deallocation is evaluated. This will be revealed by applying our altered version resolution. The process starts with $\{\varnothing \vdash \texttt{let}\ x\ \texttt{in}\ \texttt{del}\ x\ ;\ \texttt{alloc}\ x \Downarrow l, C'\}$ *as the initial proof obligations, and a substitution table. The first step matches with* E-Let*:*

$$\longrightarrow \{C \vdash \texttt{del}\ x\ ;\ \texttt{alloc}\ x \Downarrow l, C'\}, \{C \mapsto \{\pi \mapsto \{x \mapsto 0\}, \mu \mapsto \varnothing\}\}$$

The second step matches with E-Seq *and produces the proof obligations for each subterm:*

$$\longrightarrow \{(C \vdash \texttt{del}\ x \Downarrow l_1, C_1), (C_1 \vdash \texttt{alloc}\ x \Downarrow l, C')\}, \{C \mapsto \ldots\}$$

Attempting to prove $C \vdash \texttt{del}\ x \Downarrow l_1, C_1$ *will trigger the exception, which will bind* l_1 *to* ξ_0 *after unifying the proof obligation with the corresponding inference rule's head:*

$$\longrightarrow \xi_0, \{C \mapsto \ldots, l_1 \mapsto \xi_0\}$$

No more steps are required, and the process stops, revealing the uninitialized memory error.

5.3 Summary

We have illustrated the use of the $\mathcal{A}$-calculus' semantics to detect memory errors in programs' evaluations. Two techniques have been presented. One is based on a post-mortem examination of failed execution to determine their cause. Such an examination is done by observing the derivation of a program's evaluation to find the hypotheses that do not hold. The second technique consists in instrumenting the $\mathcal{A}$-calculus' operational semantics to model memory errors explicitly. We proposed two approaches to handle shadowing variable declarations. The first simply relies on α-conversion to remove shadowed names. The second modifies evaluation contexts so that they keep track of each frame's pointer table.

In the following chapter, we discuss a capability-based type system to statically guarantee that such memory errors cannot happen at execution. Memory lifecycle is handled at compile-time, and aliasing restrictions are put in place to prevent dangling references.

Chapter 6

Static Safety

Well-typed expressions do not go wrong.

Robin Milner [101]

Memory errors are notoriously difficult to debug, because they may induce undefined behaviors. Therefore the consequences of an error are not necessarily predictable nor easily observable. It follows that many memory errors can remain undetected during development, slip past testing and cause malfunctions in production. Security exploits are perfect examples of this issue [138]. As a result, much effort has been put toward methods that eradicate memory errors in software, materialized in industry-proven tools ranging from runtime memory debuggers [123, 125, 107] to static analyzers [33, 130, 131]. However, while these solutions are unquestionably capable, research suggests they are underused in practice [85]. Reasons for this shy adoption often mention result understandability and annoyance about false positives. Furthermore, code analyzers present themselves as opt-in solutions. Consequently, frustration with a tool's use and/or reporting thereof may lead to plain and simple abandons [74]. In response, a more recent trend is to propose programming languages that completely eliminate the threat of memory errors, enforcing conservative compiler-checked constraints on memory accesses (e.g. Rust, discussed further below). Overapproximation (i.e. false positive) remains an issue, but as safety mechanisms are baked into the language design, seasoned developers are less tempted to bypass them [97].

These programming languages offer powerful type systems to guarantee their safety claims, typically relying on type capabilities. In particular, affine types [135] and uniqueness have wind in their sail, and have made their way into mainstream programming languages such as Rust and C++. Unfortunately, some experiences report issues with the use of such restrictive type systems [96]. Specifically, lin-

ear references can turn simple data structures into complex implementation challenges, requiring to juggle with a plethora of concepts that are still relatively new in the mainstream software community. Move and borrow semantics are notably hard to master in Rust, while C++'s smart pointers bring their fair share of incidental complexity. This suggests there is still room for improvement in terms of flexibility and understandability.

We postulate that misunderstood assignments certainly account for some of the fog that surrounds affine semantics. Hence, we propose to study static memory control mechanisms in the context of the $\mathcal{A}$-calculus. Our analysis results in a static type system directed toward static garbage collection. Ownership [39] is used to track memory lifetime, and uniqueness [28] guarantees memory safety. The type system is informally introduced in Section 6.2 and formalized in Section 6.3. Section 6.4 then discusses the main challenges related to the support of records for static garbage collection.

We continue our exploration in Section 6.5 and suggest a type system to guarantee immutability with type capabilities. Although we do not formalize it in this thesis, we presented a formal description of a similar system for JavaScript in [117], and implemented a dynamic variant in Anzen's runtime system.

To get in line with the nomenclature that is commonly used in literature related to aliasing control, we use the term *reference* to refer to both variables and record fields in the remainder of this chapter.

6.1 Rust's Type System in a Nutshell

Before we delve into the details of the $\mathcal{A}$-calculus's type systems, let us first have a look at Rust [89]. Rust is a modern programming language that puts emphasis on a powerful type system to guarantee strong memory safety properties. Although it is still relatively young in comparison to more established languages such as C++ or Java, Rust has gained momentum in recent years, as witnessed by its appearance in numerous online polls[1].

Rust features both high-level concepts, such as generic types and higher-order functions, and a close control over memory. As it targets performance-critical applications, its runtime cannot afford the luxury of expensive dynamic memory management algorithms. Instead, Rust opts for a comprehensive program analysis to achieve static garbage collection. Ownership and uniqueness occupy a critical role in the language's type system, and are used to guarantee memory and data safety. While move assignment semantics are chosen as the default strategy, copy assignments are also possible (although not syntactically distinguishable),

[1] E.g. `https://insights.stackoverflow.com/survey/2019`

and aliasing is supported by the means of borrowing. In other words, Rust's type system is defined over all three assignment semantics from the $\mathcal{A}$-calculus.

Although Rust does not have a complete formal semantics, recent efforts have been directed towards a formalization and proof of its safety claims [88]. Furthermore, its use in real-world applications are early empirical evidences of the type system's soundness.

6.1.1 Ownership in Rust

Rust handles garbage collection statically. Similar to C, the language primarily allocates objects on the stack, so they are automatically removed from memory at the end of the call frame. Upon assignment, the language distinguishes between two kind of data types, namely copyable and non-copyable types. Copyable types are, as their name suggests, copied when assigned. On the other hand, non-copyable types are treated linearly, with a semantics close to that of the move operator in the $\mathcal{A}$-calculus.

Objects whose representation requires heap allocations (e.g. a dynamic array) cannot be mechanically deallocated when a call frame is popped off the stack. Instead, the language automatically inserts a call to a specific function, named `drop`, that is responsible for executing the object's deallocation. Rust uses ownership to determine where such calls must be inserted. Each object is attributed a single owning reference. When it goes of scope, a call to `drop` is inserted so that the object is properly disposed of. A critical implication of this design is that memory management is deeply intertwined with lexical scopes. Just like in the $\mathcal{A}$-calculus, the scope of a reference begins right after its declaration. Therefore, sequences of variable declarations effectively denote a scope hierarchy.

Example 6.1.1: Static garbage collection

`x` is given ownership on a heap-allocated number at line 2.

In a nested scope, `y` gets ownership of `x`'s object.

As `y` goes out of scope, the box is removed from memory.

Line 6 is illegal, because `x`'s value was moved at line 4.

```
1 {
2   let x = Box::new(1);
3   {
4     let y = x;
5   }
6   println!(":?", x);
7 }
```

The above code can be translated to the following program in the $\mathcal{A}$-calculus. Notice that line 9, which corresponds to line 6 in Rust's snippet, provokes a moved memory error.

$\mathcal{A}$-calculus

```
1  let x in {
2    alloc x
3    x <- 1
4    let y in {
5      alloc y
6      y <- x
7      del y
8    }
9    println(arg &- x)
10 }
```

Parameters passing in Rust also behave like assignments, and thus transfers ownership from non-copyable types. However, the language offers borrowed references as mechanism to implement a pass-by-alias semantics, without jeopardizing its static garbage collection mechanism. Similarly, a return-by-alias strategy is indicated by a reference type as the function's codomain. Rust does not abstract over the notion of pointers. Instead, pointers are represented by distinct types, denoted by an ampersand prepended to pointed type. Hence, typing a parameter with `&T` instructs Rust's type checker that the function expects to borrow access to an object from its call site.

Objects allocated in a function and assigned to its local variables are owned by that function, and therefore cannot be returned by alias. It follows that, discarding static memory, a function that returns an alias necessarily returns one that it borrowed from its call site. As Rust's type checker does not inline functions at each of their call sites, it cannot determine which borrowed argument is returned from a function and has instead to infer the return value's lifetime. If the function borrows only a single reference, the type checker (or more precisely the *borrow checker*) can infer that the return value has to be that exact same reference. Hence, the lifetime of the function's return value can also be inferred. On the contrary, if the function borrows more than one reference, the compiler can no longer infer the return value's lifetime and the function signature has to be annotated. Annotations are given in the form of generic parameters, that are added to the function's domain and codomain. As a result, function signatures act like boundary interfaces. In the function's body, annotations are used to type check assignments with an assumed knowledge of the arguments' lifetimes. At call sites, they form a constraint system that has to be solved to type check the call.

Example 6.1.2: Lifetime annotation

`'a` and `'b` are generic parameters that are instantiated at each call site with `x` and `y`'s lifetimes.

The codomain indicates that the return value has the same lifetime as the first argument.

```
fn first_of<'a,'b>(
  x: &'a i32,
  y: &'b i32)

  -> &'a i32
{
  return x;
}
```

Lifetime annotations are more than simple placeholders for arguments' lifetimes. They are *bounds*, for which Rust's compiler must prove a reference's validity. For instance, the codomain of the function `first_of` in the above example reads as "a reference on an object that lives *at least* as long as `x`". In other words, lifetime annotations describe a constraint system, that the borrow checker aims to satisfy with the largest possible bounds. This results in a flexible system, capable of describing elaborate lifetime constraints on multiple references.

Example 6.1.3: Lifetime bounds

The lifetime bound `'a` indicates that either argument should have at least *a lifetime as long as that of the other.*

The codomain is annotated to indicate that the return value has a lifetime at least *as long as that of the shortest living argument.*

```
fn either_of<'a>(
  x: &'a i32,
  y: &'a i32)

  -> &'a i32
{
  return if x > y
    { x } else { y };
}
```

Note that nothing prevents `x` and `y`'s respective arguments to have different lifetimes. The annotation simply instructs the borrow checker to find some lifetime that does not outlive either argument, so that the return values can be assumed to live at least *as long.*

The support for composite types (i.e. structures) requires more elaborate annotations. A record may store references to objects whose lifetimes have to be tracked as well. Hence, composite types also have to be augmented to provide information about their captures. Rust solves this issue by adding lifetime annotations on structure definitions, which are used to specify the fields' lifetimes.

Example 6.1.4: Lifetime bounds on structures

The lifetime bound `'a` is used to denote the minimum lifetime of the field `bar`.

Line 6 instantiates an instance of `Foo`, whose `bar` member lives in the same scope as `x`.

Line 10 is illegal because `y` does not live long enough to be captured by `f`.

```
1  struct Foo<'a> {
2    bar: &'a i32
3  }
4
5  fn main() {
6    let x = 32;
7    let mut f = Foo {
8      bar: x
9    };
10   {
11     let y = 13;
12     f.bar = &y;
13   }
14 }
```

6.1.2 Mutability and Uniqueness in Rust

Rust's references are immutable by default. Mutability is expressed explicitly by the means of a keyword `mut`.

Rust

```
1 let x = 42; // An immutable variable.
2 let mut y = 1337; // A mutable variable.
```

Similarly, mutable borrowed references also have to be declared explicitly. In order to prevent data races, the type system enforces that there can ever be only one mutable reference to an object. Furthermore, mutable borrowed references cannot coexist with immutable borrowed ones. Simply put, there can be either multiple readers and no writer, or one writer and no readers. An important consequence of that restriction is that mutable self-referential structures (e.g. graphs or doubly linked lists) cannot be represented with references in (safe[2]) Rust.

Example 6.1.5: Mutable references

`x` is declared as a mutable reference, so line 2 is legal.

```
1 let mut x = 2;
2 x = 4;
```

[2]Rust offers a compiler annotation to turn its borrow checker off, which allows unrestricted aliasing in "unsafe" portions of the code.

`r1` is a mutable reference on `x`'s value, so line 4 is legal.

Line 5 is illegal because `r1` is the only active mutable reference.

Line 6 is illegal because an immutable reference cannot coexist with a mutable one.

```
3 let r1 = &mut x;
4 *r1 = 8;

5 x = 16;

6 let r2 = &x;
```

6.2 Static Garbage Collection

This section informally introduces a type system to handle static garbage collection in the $\mathcal{A}$-calculus. Our approach borrows heavily from that of Rust's and uses ownership and uniqueness to determine where allocations and deallocations should be inserted. While the type system is able to track ownership across conditional terms and function boundaries, it does not support records. Causes for such a limitation are explained in Section 6.4, followed by suggestions of solutions.

6.2.1 Ownership and Uniqueness

The core of the approach is to statically determine where allocations and deallocations must be inserted, so that `alloc` and `del` terms are no longer explicitly required. Obviously, the semantics of the $\mathcal{A}$-calculus' operators must be taken into account to avoid memory errors.

Example 6.2.1: Static garbage collection

An allocation should be inserted at line 2 so that `x` is ready to be initialized.

No allocation should be made for `y`, as it is only an alias.

```
1 let x in {
2   // <= alloc x
3   x <- 42
4   let y in {

5     y &- x
```

A deallocation should be inserted at line 7, as no other reference can access `x`'s memory.

```
6     }
7 } // <= del x
```

Static garbage collection is achieved by tracking ownership and uniqueness. Upon allocation, a memory location's ownership is given to a single particular reference, which thereby becomes responsible for its deallocation. Aliases on the other hand do not hold ownership on the value to which they refer, and therefore do not need to be deallocated. In other words, lexical scopes describe memory regions [26, 134], thereby bounding the memory lifetime to owning references, whereas aliases do not participate in the determination of the memory lifecycle.

Memory errors are avoided by restricting the operators that can be used on a particular reference, depending on its state [127] and lexical scope. For instance, as `y` is not an owner in the above example, placing it as the right hand side of a move assignment should be flagged as an illegal operation. References can be in either of these five states:

Unallocated denotes a reference that is not bound to any memory.

Unique denotes a reference to an unaliased memory location.

Shared denotes a reference to a memory location bound to other references.

Borrowed is similar as shared, but for references without ownership.

Moved denotes a reference that has been moved, by move assignment.

All references start in the **unallocated** state. An allocation is inserted when an unallocated reference is assigned by move (i.e. $\leftarrow$) or copy (i.e. $:=$). The newly allocated location is immediately placed under the assigned reference's ownership, which goes into the **unique** state. A reference goes into the **shared** state if it simultaneously holds ownership and refers to an aliased location. Conversely, a reference goes into the **borrowed** state if it does not hold ownership on the location to which it refers. A deallocation is inserted when an owning reference goes out of scope. Of course, the type system must guarantee that the reference is **unique** at this point. Unlike in Rust, where borrowed references are statically typed to reflect the fact that they do not hold ownership, the $\mathcal{A}$-calculus does not define any "pointer type". Hence references may be owning and then borrowed in the course of their lifetime. When an owning reference is assigned by alias, a deallocation must be inserted to free the memory it owns, and the reference goes to the **borrowed** state. Finally, if ownership is taken from an owning reference,

then it goes into the **moved** state. Table 6.1 summarizes how these transitions can take place. Each cell represents the state in which a reference goes if placed on the left or right side of the operator indicated by the column. The position of the bullet denotes the side of the operator on which the reference should be. For instance, • := denotes the left operand of a mutation assignment.

↗	as left operand			as right operand		
	• &-	• :=	• ←	&- •	:= •	← •
unalloc.	borrowed	unique	unique	×	×	×
unique	borrowed	unique	unique	shared	unique	moved
shared	×	shared	shared	shared	shared	×
borrowed	borrowed	borrowed	borrowed	borrowed	borrowed	×
moved	borrowed	unique	unique	×	×	×

Table 6.1: Reference state transitions

Example 6.2.2: State transitions

Variables `x`, `y` *and* `z` *start in the* ***unallocated*** *state.*

`x` *and* `y` *become* ***unique*** *at line 2 and 3, respectively.*

`x` *becomes* ***shared*** *while* `z` *becomes* ***borrowed*** *at line 4.*

No state transition occurs at line 5 and 6.

`y` *becomes* ***shared*** *while* `x` *goes back to* ***unique*** *at line 7.*

`x` *becomes* ***moved*** *at line 8.*

```
1 let x, y, z in {
2   x := 10
3   y <- 20
4   z &- x
5   y := 1337
6   z <- 42
7   z &- y
8   y <- x
9 }
```

Recall from the operational semantics that literals are evaluated as newly allocated objects. Consequently, evaluating a literal allocates a new memory location that is not owned by anyone, which of course constitutes a memory leak. A simple solution to this problem is to create a "virtual" owner every time a literal value is allocated, so that it can dispose of the memory location it owns at the end of the scope, with the same mechanism as other regular references.

Example 6.2.3: Disposal of literal values

At line 2, an allocation is inserted for `x`*, after a virtual owner is given ownership on the location holding the value* 42*.*

At line 5, both the virtual owner and `x` *are deallocated.*

```
1 let x, y in {
2   // <= alloc x
3   x := 42
5 } // <= del 42's owner
6   // <= del x
```

Although borrowed references are not involved in the lifecycle of the memory to which they refer, the type system guarantees that access to this memory is safe for the duration of the borrow. To that end, creating an alias on an unallocated or moved reference is forbidden. Furthermore, the type system prevents aliasing on memory locations whose owner lives in a nested scope, as those could be deallocated before the borrowed reference goes out of scope. These two restrictions are of course necessary (yet not sufficient) to avoid dangling references.

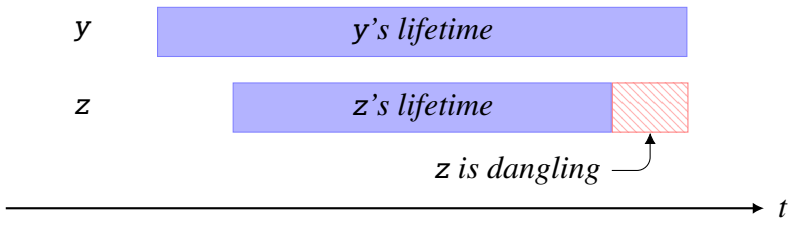
Example 6.2.4: Aliasing to a nested scope

An allocation is inserted at line 3, attributing ownership to `z`*.*

Line 5 is illegal because `z` *has a shorter lifetime than* `y`*.*

```
1 let y in {
2   let z in {
3     // <= alloc z
4     z <- 42
5     y &- z
6   } // <= del z
7 }
```

Representing lifetimes graphically, one can see how allowing line 5 could lead to a user after free error:



Ownership transfer is carried out by the means of move assignments (i.e. ←). The right operand is lifted from its responsibilities, which are placed onto the left operand. Move operations can take a value out of a nested scope, effectively prolonging its lifetime. For instance, a function can move a return value owned

by one of its local variables to avoid an expensive copy. Once ownership has been transferred, the former owner enters the moved state and can no longer appear as any assignment's right operand until it is reassigned. A deallocation is inserted to dispose of the moved memory location when it goes out of scope.

An owner cannot abandon its responsibilities to assume ownership of another value. Hence, if one appears as a left operand, a deallocation is inserted before the assignment takes place. This implies that an owner cannot transfer ownership to itself, as its deallocation would take place right before the move operation, thus provoking a memory error.

Example 6.2.5: Ownership transfer

An allocation is inserted at line 3, attributing ownership to `z`.

Ownership is transferred from `z` *to* `y`*, which is now responsible for deallocation*

`z` *remains moved until line 7, so a deallocation is inserted.* `y` *is freed because it bears ownership.*

```
1 let y in {
2   let z in {
3     // <= alloc z
4     z <- 42
5     // <= alloc y
6     y <- z
7   } // <= del z
8 } // <= del y
```

Transferring ownership from a shared owner is forbidden. This restriction preserves the linearity assumption of the move operator, as shared references are not unique, by definition. Besides, it also prevents dangling aliases. Remember that a move assignment assigns a moved value (i.e. □ in the operational semantics) to its right operand. As a result, even if the new owner still lived longer than the alias, any alias would refer to the move value.

Example 6.2.6: Illegal ownership transfer

Transferring `x`*'s ownership at line 6 would leave* `y` *referring to a moved location.*

```
1 let x, y, z in {
2   // <= alloc x
3   x <- 1337
4   y &- x
5   // <= alloc z
6   z <- x
7 } // <= del z
```

The type system determines whether or not an owner is aliased based on a

notion of uniqueness. Uniqueness in the $\mathcal{A}$-calculus is treated as a fractionable property, similar to Boyland's proposal [28]. The intuition is that it is initially obtained in full at allocation, but gets *fractioned* for each alias. The loss of the uniqueness capability is only temporary, as an owner can later gather all fragments back to rebuild it. Put another way, each aliasing assignment creates a new fragment of uniqueness, while one is sent back to its owner every time an alias goes out of scope or is reassigned. Aliases themselves are allowed to further fraction their fragment to allow additional aliases. However this does not create a hierarchy, and a fragment is always directly sent back to its owner. This mechanism effectively implements borrowing and reborrowing. As lexical scopes are used to delimit memory regions and automate deallocation, our model does not require an elaborate pointer analysis to determine when fragments should be reclaimed within function boundaries.

Example 6.2.7: Fractionable uniqueness

`x` *gets a full uniqueness capability at line 3.*

`y` *and* `z` *fraction* `x`*'s capability at line 5 and 7, respectively.*

`z` *goes out of scope at line 8, so* `x` *gets a fragment back, but full uniqueness is not recovered until* `y` *goes out of scope at line 9.*

```
1  let x in {
2    // <= alloc x
3    x <- 42
4    let y in {
5      y &- x
6      let z in {
7        z &- y
8      }
9    } // <= del y
10 } // <= del x
```

For the same reason as transferring ownership from a shared owner is forbidden, reassigning a shared owner to an alias is also prohibited. Doing so would trigger the deallocation of the owner's memory, leaving all aliases on it to refer to a freed memory location. Furthermore, reassigning an owner to an alias of itself is also illegal, even if it is unique. The operation would leave the memory location unowned, thereby resulting in a memory leak.

Example 6.2.8: Owner reassignment

`x` *gets ownership at line 3.*

```
1  let x in {
2    // <= alloc x
3    x <- 42
```

Line 4 is illegal because it leaves `x`'s memory unowned and thereby leaking.

```
4     x &- x
5   } // <= no del x
```

6.2.2 Static Nondeterminism

Remember that all the mechanisms that have been discussed so far are carried out statically (e.g. at compile time). Therefore the compiler must be able to prove whether or not a reference is expected to hold ownership, in order to insert appropriate allocation and deallocation instructions. Errors in this process would have dire consequences on the resulting program, that would run the risk to trigger uninitialized memory errors and/or leak memory at runtime. This is problematic in the presence of conditional terms, because determining whether a reference gets or loses ownership becomes statically nondeterministic. Imagine for instance a situation in which a reference gets ownership in one branch but is left unassigned in the other. Furthermore, the assignment (or absence thereof) of a reference may also induce additional implications on the state of other references, should the states of the latter change in one branch but not in the other. Indeed, if uniqueness cannot be properly tracked, then one can no longer determine whether or not move operations are safe.

Example 6.2.9: Statically nondeterministic ownership

If `c` holds, then `x` is owner and `y` is an alias.

If `c` does not hold, then `y` is owner and `x` is an alias.

Line 10 is legal anyway, but which deallocation need to be inserted at line 12 is nondeterministic.

```
1  let x, y in {
2    let z in {
3      if c then {
4        x <- 42
5        y &- x
6      } else {
7        y <- 42
8        x &- y
9      }
10     z := y // legal
11   }
12 } // <= del x, y?
```

The most conservative approach is to reject a program whenever both branches of a conditional term do not lead to the exact same situation with respect to ownership. While this technique would address the issue rather easily, it would sig-

nificantly hinder the usability of the language. Another, more relaxed idea is to analyze both branches of a conditional term, in order to determine a conservative scenario that can be satisfied no matter which is taken at runtime. For instance in above the example, line 10 is legal no matter which branch is executed, because `y` is assigned in either of them. Unfortunately, while this approach can solve definite initialization [64], it does not address the ownership nondeterminism. Our type system should not only ensure that a reference is initialized before it is used, but it should also determine whether or not a reference has ownership when it is reassigned or goes out of scope. Put differently, we need a *definite ownership analysis*.

Nonetheless, we are not forced to adopt the most restrictive approach. We define a notion of *transient* ownership. An owner is transient if its ownership cannot be guaranteed beyond one branch of a conditional term. In other words, it denotes an owner whose ownership is nondeterministic. Unlike unique or shared references, transient owners cannot be aliased. However they differ from unallocated and moved references in that they are allowed to be copied, i.e. placed on the right side of a mutation assignment. Furthermore, we apply a few rules to deal with conditional terms:

1. If a reference is allocated at the end of one branch but not at the end of the other, it is assumed unallocated after the conditional term.

2. If a reference holds ownership at the end of one branch but not at the end of the other, it is transient and assumed unallocated after the conditional term.

3. If a non-transient owner is aliased at the end of one branch, it is assumed aliased after the conditional term.

4. A reference borrowed from a transient owner cannot survive after the conditional term.

Notice the subtle difference between the two first rules. While both prescribe that a reference assigned in a branch shall be assigned in the other as well, the second adds another requirement when ownership is involved, so that acquiring or losing ownership in one branch does not lead to an inconsistent situation after the conditional term. In Example 6.2.9, this means that both references `x` and `y` are considered unallocated at line 9. While this turns line 10 into an illegal statement, which is overly conservative, the nondeterministic issue at line 12 is solved by issuing `x` and `y`'s deallocations earlier. Note that assigning `z` is possible if the assignment is pushed to the end of both branches, as `y` can be copied even if it is transient.

Example 6.2.10: Aliasing on transient owner

`y`*'s assignment at line 4 is illegal because* `x` *is transient.*

`x`*'s assignment at line 8 is illegal because* `y` *is transient.*

No deallocation is inserted, because `x` *and* `y` *are assumed unallocated after line 10.*

```
1  let x, y in {
2    if c then {
3      x <- 42
4      y &- x
5      // <= del x
6    } else {
7      y <- 42
8      x &- y
9      // <= del y
10   }
11 }
```

The first rule guarantees that aliases formed in a branch can be assumed initialized after the conditional term. However it does not entirely remove nondeterminism. If both branches form aliases on different owners, then one cannot determine uniqueness statically. Fortunately, such situations can be supported conservatively if borrowing references take a fragment of uniqueness from all owners susceptible to be aliased.

6.2.3 Lifetime Annotations

A common technique to speed up source compilation is to process files (or groups thereof) separately. This allows some parts of a program to be compiled in separate modules, that are then linked together to constitute the complete output. In languages that undergo static compilation, this strategy is key to the support of reusable language libraries, as all code would have to be recompiled every time a program is dealt with otherwise. However, the drawback is that it hinders potential for inference and type checking, as the compiler no longer "sees" all of the program at once. The solution is to provide interfaces of the actual underlying code in a format that is faster to process, with enough information to make up for the missing material to which the compiler would otherwise have access. For instance, a C header file typically describes function signatures, so that the compiler can assume the existence of an actual function with the same name, domain and codomain. Another use for module interfaces is to link different languages together. It is possible for instance to reuse a function written in C from a Swift program using an interface that lets Swift's compiler perform its usual type infer-

ence and produce the necessary glue.

In a statically typed language supporting first-class functions, signatures are also necessary to type check function applications. Unlike in a first-order language, inlining is not an option, because the actual function a term represents cannot be determined statically. Therefore a type checker has to rely on the term's type to check whether a call expression is well-typed. This is illustrated in the following Swift excerpt, in which the function that is applied at line 5 cannot be determined statically:

Swift

```
1 typealias Fn = (Int, Int) -> Int
2 func either(_ f: Fn, _ g: Fn) -> Fn {
3   return Bool.random() ? f : g
4 }
5 print(either(+, -)(6, 4))
```

In the $\mathcal{A}$-calculus, modularity is drawn at function boundaries. In other words, functions are regarded as black boxes that can be swapped for another. As our type system has to keep track of ownership and uniqueness across function applications, it needs some aliasing information to be described in function interfaces. For instance, statically determining whether a term $x \,\&\text{-}\, f(y \,\&\text{-}\, x)$ is legal requires to know the lifetime of f's return value, so as to ensure that it can outlive that of x. If f returns a value allocated within its own scope, then it will be freed before x, leaving it dangling. On the other hand, if f returns a reference on its argument, then x is simply reassigned to itself [3]. Adding detailed lifetime annotations on every function parameter and return value would be cumbersome, hinder reusability and defeat the purpose of modularization. Fortunately, describing *bounds* on lifetimes is sufficient to prescribe safety and preserve genericity. Furthermore, given some safe defaults and assumptions, most annotations can be inferred automatically.

Lifetime Bounds in the $\mathcal{A}$-Calculus

We draw inspiration from Rust's lifetime annotations' system, but bring a few amendments to accommodate the $\mathcal{A}$-calculus' semantics. As mentioned in Section 6.1, Rust does not abstract over the notion of pointers. Instead, pointers are represented as distinct types, denoted by a prepended ampersand. Thus, typing a return value with `&T` instructs Rust's compiler that the function will necessarily return an alias on one of its arguments, or fields thereof. Conversely, if the return value is not a pointer, then it is known to be returned with ownership. The

[3] As mentioned earier, the statement is still illegal if x is owner.

$\mathcal{A}$-calculus on the other hand cannot afford such assumptions a priori, as it cannot distinguish between owners and aliases statically. As a result, we are left with a nondeterministic choice to identify whether arguments and return values are passed with ownership. Furthermore, since functions are seen as black boxes and only described by their signature, one cannot resolve conflicting situations with transient ownership, similar to how conditional terms are handled. We have no choice but to annotate incoming and outgoing references to distinguish between owned and borrowed ones. Unfortunately, this means that some of the $\mathcal{A}$-calculus' versatility has to be sacrificed, with respect to parameter and return passing policies.

We introduce a reference qualifier `@own` to annotate references that should be passed with ownership. Those can only be passed by move or copy (i.e. $\leftarrow$ or $:=$). The dual qualifier `@brw` annotates references that should be passed borrowed. In the absence of any annotation, `@own` is assumed by default.

Example 6.2.11: Function without lifetime bounds

Each parameter is marked `@brw`, and is therefore expected to be provided borrowed.

The codomain is marked `@own`, so return values are assumed to be passed with ownership.

Line 6 is legal because it temporarily reborrows `x`. Line 7 is legal, because it returns a copy *of `x`.*

Line 9 is illegal, because it attempts to return an alias *on `x`.*

```
1  fun (
2    x: @brw,
3    p: @brw)
4    -> @own
5  {
6    if p(x &- x) then {
7      return := x
8    } else {
9      return &- x
10   }
11 }
```

Choosing the pass-by-alias policy as the default strategy on parameters would be an equally valid pick, roughly in line with the passing semantics of reference-based languages (discarding primitive types). However, it would be a poor choice for return values. On codomain annotations, `@brw` must be parameterized to describe *lifetime bounds*. This parameterization enables the inference of the return value's lifetime, it also serves to track uniqueness fragments beyond function boundaries. Choosing a pass-with-ownership as the default strategy preserves

symmetry between parameters and codomain annotations, and does not require any parameterization by default.

Unlike Rust, our type system does not rely on generic placeholders to describe lifetimes. Instead, we directly refer to the name of the function's parameters. While Rust allows for more expressiveness, in particular when dealing with composite types, this approach is sufficient in the context of the $\mathcal{A}$-calculus. For instance, given a function with two parameters `x` and `y`, `@brw(x, y)` reads as "an alias that lives *at least* as long as the shortest of `x` and `y`'s arguments".

Note that we cannot determine statically whether or not uniqueness is still fragmented after the function call. This problem is similar to the so-called static nondeterminism observed in conditional terms, and relates to the inability to know whether a function actually returns one of the borrowed reference it gets as an input. Fortunately, because functions in the $\mathcal{A}$-calculus do not have any closure, all possible situations can be inferred from the function's signature alone. Hence, if a borrowed reference returned by a function is reborrowed, taking a fragment of uniqueness from all arguments that may have been returned conservatively covers all aliasing scenarios.

Example 6.2.12: Lifetime bounds in the $\mathcal{A}$-calculus

Parameters' annotations do not require parameterization.

Return values' lifetimes are bound to `x` and `y`'s lifetimes.

Line 8 is illegal because it attempts to return ownership. Line 10 is legal, because it returns an argument with the proper lifetime.

```
1  fun (
2    x: @brw,
3    y: @brw,
4    p: @brw)
5    -> @brw(x, y)
6  {
7    if p(x &- x) then {
8      return := x
9    } else {
10     return &- y
11   }
12 }
```

6.3 Type Checking

We now formalize the type system that we presented in the previous section. Type checking is performed in two steps. The first one focuses on the so-called *flow-insensitive* typing information, and does not depend on the program's evaluation context at runtime. The second step relates to the so-called *flow-sensitive* typing information, and depends on the runtime evaluation context. In other words, flow-

insensitive properties are purely lexical, while flow-sensitive properties should be inferred from the program's semantics.

Consider for instance a reference declared to hold numbers, that has not been initialized yet. Its flow-insensitive type defines the operations supported by numbers (e.g. $+, -, \times, \ldots$), while its flow-sensitive type indicates that it cannot be used for read accesses. Assuming the type system is not dynamically typed, binding the reference to a value will obviously not change the fact that it is supposed to hold numbers. Hence, its flow-insensitive type will not change during the course of the program's execution. On the other hand, its flow-sensitive type will be updated to reflect that it now supports read accesses.

As we assume all allocations and deallocation primitives to be inserted implicitly, following the strategy we have discussed in the previous section, they do not need to be considered in the type system's formalization. Instead, all rules are designed precisely so that the implicit insertion cannot cause memory issues.

6.3.1 Flow-Insensitive Types

Flow-insensitive describe two kinds of information. One relates to the semantic nature of the expression (e.g. a natural number), and is therefore called the *semantic type* of an expression. The other relates to its scope, and is therefore called the *scope type* of an expression. Scope types are flow-insensitive because they are delimited lexically.

Note that typing the $\mathcal{A}$-calculus has important consequences on its operational semantics. First, it means that variables can no longer be assigned with any values of different types. Consider for instance the following term:

$$\texttt{let}\ x\ \texttt{in}\ x \leftarrow 2\ ;\ x \leftarrow\ \lambda y \triangleright y$$

Though the term is well-formed in the untyped $\mathcal{A}$-calculus, it is ill-typed in the typed $\mathcal{A}$-calculus because it violates x's semantic type. While this restriction can be lifted by the use of type unions, as available in TypeScript [19] for instance, taking this path would add a considerable complexity to our type system.

Secondly and more consequential, typing functions require recursive types. It follows that the $\mathcal{A}$-calculus cannot be simply typed (within the meaning of the simply typed λ-calculus). Consider for instance the following term:

$$\texttt{let}\ f\ \texttt{in}\ f \leftarrow (\lambda x, f \triangleright \texttt{ret}\ \leftarrow f(x \leftarrow x, f\ \texttt{\&-}\ f))\ ;\ f(x \leftarrow 2, f\ \texttt{\&-}\ f)$$

The semantic type of f recursively refers to itself, thus resembling a type of the form $\tau \times (\tau \times (\tau \times \cdots \to \tau) \to \tau) \to \tau$. Dropping the support for recursion would significantly reduce the expressiveness of the language, to the point that it could no longer be considered Turing complete.

Semantic Types

Formalizing type correctness and/or inference with respect to semantic types is beyond the scope of this thesis. For a language as simple as the $\mathcal{A}$-calculus, semantic typing fits the standard Hindley-Milner system [101]. Nonetheless, annotated function signatures are required to check lifetime bounds at function boundaries. As discussed in the previous section, function signatures are annotated with reference qualifiers to specify bounds on parameters and return values. We start with a definition of such qualifiers.

Definition 6.3.1: Reference qualifiers

The set of reference qualifiers is given by $\mathbb{Q}_R = \{\texttt{brw}, \texttt{own}\}$*, where* `brw` *denotes a borrowed reference and* `own` *denotes an owned reference.*

Definition 6.3.2: Semantic types

Let $\mathbb{A}$ *denote the set of atomic literals,* $\mathbb{X}$ *denote the set of identifiers,* $\mathbb{Q}_R$ *denote the set of reference qualifiers, and* $X \subseteq \mathbb{X}$ *be a subset of identifiers. Let* $\mathcal{T}$ *denote a set of type variables. The set of semantic types* $\mathbb{T}$ *is defined as the minimal set such that:*

$$\begin{aligned} \mathbb{A} &\in \mathbb{T} && \textit{atomic values' type} \\ \langle dom : X \to \mathbb{T}_\mathbb{Q}, codom : \mathbb{T}_\mathbb{Q}, bds : \mathcal{P}(X) \rangle &\in \mathbb{T} && \textit{a function type} \\ \alpha \in \mathcal{T} &\implies \alpha \in \mathbb{T} && \textit{a type variable} \\ \tau \in \mathcal{T} &\implies \mu\alpha.\tau \in \mathbb{T} && \textit{a recursive type} \end{aligned}$$

where $\mathbb{T}_\mathbb{Q} = \langle qual : \mathbb{Q}_R, type : \mathbb{T} \rangle$ *denotes qualified semantic types.*

Notation: Recursive types

The notation $\mu\alpha.\tau$ describes a recursive type τ, in which occurrences of α denote $\mu\alpha.\tau$ recursively.

Example: *Consider the following type:*

$$\begin{aligned} \tau &= \mu\alpha.f \\ f.dom &= \langle x \mapsto \langle qual \mapsto \texttt{brw}, type \mapsto \mathbb{A} \rangle, f \mapsto \langle qual \mapsto \texttt{brw}, type \mapsto \alpha \rangle \rangle \\ f.codom &= \langle qual \mapsto \texttt{own}, type \mapsto \mathbb{A} \rangle \\ f.bds &= \varnothing \end{aligned}$$

τ describes a recursive function type that accepts a borrowed reference to

an atomic value as its x parameter, and a borrowed reference to a function with the same signature as its f parameter.

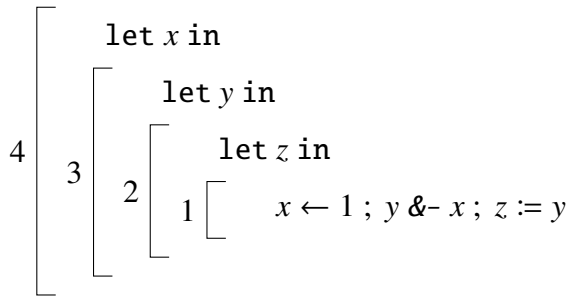
Notice that we represent function domains in the form of records that map parameter names to their expected type. Consequently, parameter names are inherently part of the function signature. Although this is a departure from most strong type systems, where parameters are usually positional, it is in line with the way function application is defined in the $\mathcal{A}$-calculus, where parameter names are used as assignments' left operands to specify the passing policy. Another advantage is that lifetime bounds are more easily expressed, as they can unambiguously refer to parameter names.

Scope Types

We now focus on the second constituent of flow-insensitive types. We mentioned in Section 4.3.1 that variable declarations and functions delineate lexical scopes. In turn, those delimit identifiers' lifetimes and are used to derive static garbage collection. Therefore the notion of lifetime should be formalized. We associate each scope with a *rank i* that is as low as its depth. In plain English, the more nested a scope is, the lower its rank gets. Intuitively, a scope rank i denotes a lifetime that outlives any lifetime associated with a lower rank $i - n$. By abuse of terminology, we sometimes use the terms "scope" and "scope rank" interchangeably.

Example 6.3.1: Scope rank

The picture below depicts a term with four lexical scopes, and shows their respective rank.



The whole expression is enclosed in a scope with rank 4. The first declaration delimits a scope with rank 3. The second declaration delimits a scope with rank 2. The third declaration delimits a scope with rank 1.

Notation: Scope belonging

We say that a term $t \in \mathcal{AC}$ *lives* in a scope ranked i, written $t \sqsubset i$, if the value to which it is evaluated is bound by i, that is if t is defined in the scope ranked i.

Recall that, in order to avoid dangling reference, the type system does not allow a variable to be bound by alias to a memory location whose owner lives in a nested scope. This can be enforced by guaranteeing that an aliasing assignment's left operand lives in a scope with a rank lower than that in which its right operand lives. More formally, the type system has to prove that $t \mathbin{\&^-} u \implies t \sqsubset i \wedge u \sqsubset j \wedge i < j$. Verifying this property for simple identifiers is trivial, and only depends on the scope in which the identifiers live. However, information contained in type signatures are necessary to check function boundaries. Codomain annotations form a constraint system that type checking must solve at each call site.

6.3.2 Scope Checking

We describe scope checking with inference rules. Typing judgements are of the form $\Sigma, i \vdash t \sqsubset j$, where Σ is a scoping context, i the rank of the scope in which t is evaluated and j the rank of the scope in which t lives.

Definition 6.3.3: Scoping context

Let $\mathbb{X}$ be the set of reference identifiers. A scoping context $\Sigma : \mathbb{X} \nrightarrow \mathbb{N} \cup \{-\infty, +\infty\}$ is a table that maps identifiers to scope ranks. Furthermore, we define a total order relation $<$ on $\mathbb{N} \cup \{-\infty, +\infty\}$ such that:

$$-\infty < +\infty$$
$$\forall n \in \mathbb{N}, n < +\infty$$
$$\forall n \in \mathbb{N}, -\infty < n$$

Unlike the operational semantics, our type system does not deal with name shadowing. Instead we assume that all identifiers are declared only once in a given lexical scope. This does not restrict the set of programs that can be type checked, as any shadowing declaration can be replaced with α-conversion. For instance, the term `let` x `in let` x `in` $x \leftarrow 2$ is α-equivalent (i.e. equivalent up to α-conversion) to `let` x `in let` x_0 `in` $x_0 \leftarrow 2$.

There is no base case that fixes i in the typing judgement $\Sigma, i \vdash t \sqsubset j$. Put differently, there is not any rule that computes a concrete value for i. This is unnecessary because we are only interested in the relations between scope ranks, which are sufficient to determine whether an expression lives shorter or longer

than another.

Within Function Boundaries

Inferring variables' scopes consists in consulting the scoping context. Like in the operational semantics, we do not consider programs with undefined identifiers. Hence S-Var does not apply if $x \notin \mathrm{dom}(\Sigma)$. The scoping context Σ is updated by variable declarations. Those map the declared identifier to a scope with an inferior rank, in which their body is then type checked. Although a variable declaration may delimit a scope, the term constituting its body does not necessarily live in the same, because the latter may refer to an expression (e.g. a variable) defined in another scope. Consider for example the term `let` x `in let` y `in` x, and assume y's declaration delimits a scope with rank 3. The declaration's body consists of the identifier x, that is declared in a nesting scope and is therefore necessarily associated with a higher rank.

Variables

S-Let
$$\frac{\Sigma[x \mapsto i-1], i-1 \vdash t \sqsubset j}{\Sigma, i \vdash \texttt{let}\ x\ \texttt{in}\ t \sqsubset j}$$

S-Var
$$\frac{x \in \mathrm{dom}(\Sigma)}{\Sigma, i \vdash x \sqsubset \Sigma.x}$$

Example 6.3.2: Variable declarations

Consider the following term:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ x$$

Let Σ be the empty scoping context, scope checking is described by the following derivation:

$$\frac{\dfrac{\dfrac{x \in \mathrm{dom}(\{x \mapsto i-1, y \mapsto i-2\})}{\{x \mapsto i-1, y \mapsto i-2\}, i-2 \vdash x \sqsubset i-1}}{\{x \mapsto i-1\}, i-1 \vdash \texttt{let}\ y\ \texttt{in}\ x \sqsubset i-1}}{\varnothing, i \vdash \texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ x \sqsubset i-1}$$

As assignments return the memory location of their left operand, they obviously live in the same scope. Aliasing assignments additionally require that the scope of the left operand live shorter than the right operand. It is, in fact, the most important constraint of the scope checking judgement, as it contributes to the mechanism that guarantees the absence of dangling references, by preventing borrowed references on shorter-living owners.

Assignments

$$\textsc{S-Mut}\ \frac{\Sigma, i \vdash x \sqsubset j \qquad \Sigma, i \vdash u \sqsubset j'}{\Sigma, i \vdash x := u \sqsubset j} \qquad \textsc{S-Move}\ \frac{\Sigma, i \vdash x \sqsubset j \qquad \Sigma, i \vdash u \sqsubset j'}{\Sigma, i \vdash x \leftarrow u \sqsubset j}$$

$$\textsc{S-Alias}\ \frac{\Sigma, i \vdash x \sqsubset j \qquad \Sigma, i \vdash u \sqsubset j' \qquad j < j'}{\Sigma, i \vdash x \,\&\text{-}\, u \sqsubset j}$$

Example 6.3.3: Scope checking aliasing assignments

Consider the following term:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ y\ \&\text{-}\ x$$

The aliasing assignment is well-typed because x lives in a scope with a higher rank than x, as demonstrated by the following excerpt of derivation:

$$\dfrac{\dfrac{\dfrac{\vdots}{x \sqsubset i-1} \qquad \dfrac{\vdots}{y \sqsubset i-2} \qquad i-2 < i-1}{\{x \mapsto i-1, y \mapsto i-2\}, i-2 \vdash y \,\&\text{-}\, x \sqsubset i-2}}{\{x \mapsto i-1\}, i-1 \vdash \texttt{let}\ y\ \texttt{in}\ y\ \&\text{-}\ x \sqsubset i-2}}{\varnothing, i \vdash \texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ y\ \&\text{-}\ x \sqsubset i-2}$$

The scope checking succeeds because there exists i such that $i - 2 < i - 1$.

Atomic literals are always assumed living in a nested scope, regardless of the scope in which they are evaluated, because virtual owners are deallocated before any actual owning reference.

Atomic values

$$\textsc{S-Atom}\ \frac{a \in \mathbb{A}}{\Sigma, i \vdash a \sqsubset -\infty}$$

Example 6.3.4: Aliasing an atomic literal

Consider the following term:

$$\texttt{let}\ x\ \texttt{in}\ x\ \&\text{-}\ 2$$

The aliasing assignment is ill-typed because x lives in a scope with a rank superior to that in which the literal 2 lives. The type error is demonstrated by the following derivation:

$$\dfrac{\dfrac{\dfrac{x \in \mathrm{dom}(\{x \mapsto i-1\})}{x \sqsubset i-1} \qquad \dfrac{2 \in \mathrm{dom}(\mathbb{A})}{2 \sqsubset -\infty} \qquad i-1 < -\infty}{\{x \mapsto i-1\}, i-1 \vdash x \mathbin{\&-} 2 \sqsubset i-1}}{\varnothing, i \vdash \texttt{let}\ x\ \texttt{in}\ x \mathbin{\&-} 2 \sqsubset i-1}$$

The scope checking fails because there is no i such that $i - 1 < -\infty$.

Conditional terms live in the shortest living of either of their branches and sequences however live in the same scope as the last executed statement.

Conditional term

$$\textsc{S-Cond}\ \dfrac{\Sigma, i \vdash t_1 \sqsubset j_1 \qquad \Sigma, i \vdash t_2 \sqsubset j_2 \qquad \Sigma, i \vdash t_3 \sqsubset j_3}{\Sigma, i \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \sqsubset \min(j_2, j_3)}$$

Sequences

$$\textsc{S-Seq}\ \dfrac{\Sigma, i \vdash t_1 \sqsubset j_1 \qquad \Sigma, i \vdash t_1 \sqsubset j_2}{\Sigma, i \vdash t_1\ ;\ t_2 \sqsubset j_2}$$

At Function Boundaries

Recall that modularity is drawn at function boundaries. Therefore type checking has to extract all lifetime information from a function's signature to determine the correctness of the assignments performed within the function's body. Borrowed parameters necessarily live in the call site, and therefore can be assumed to have a longer lifetime that any local variable or parameter passed with ownership.

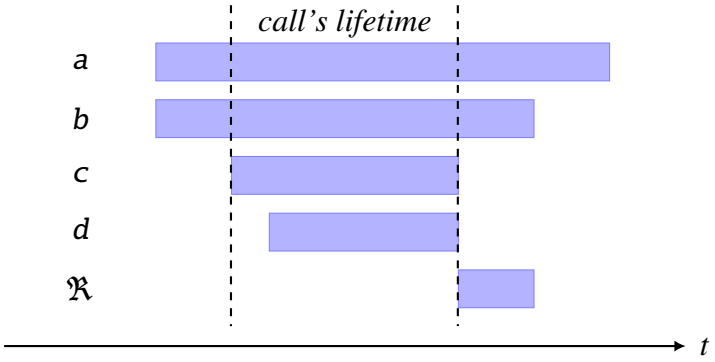
Example 6.3.5: Lifetime at function boundaries

Consider the following function:

$\mathcal{A}$-calculus

```
fun (a: @brw, b: @brw, c: @own) -> @brw(a,b) {
  let d in {
    // ...
  }
}
```

The two first parameters `a` *and* `b` *are borrowed and therefore can be expected to live longer than the function call, whereas the parameter* `c` *is passed with ownership and therefore lives in the same scope as the function's body. Since both parameters* `a` *and* `b` *appear in the codomain's bounds, the return identifier* $\mathfrak{R}$ *can be assumed to live at least as long as the shortest living parameter. Finally, the local variable* `d` *necessarily lives shorter than any of the parameters.*



For function declarations, the goal is to extract a constraint system from the type signature to create a scoping context that can type check the function's body. We define a procedure funscope(τ) to compute such a context. Given a function signature τ, let $P_\tau^{\texttt{brw}}$ be a set composed of the name of the parameters passed by alias (i.e. those annotated with `@brw`), and $P_\tau^{\texttt{own}}$ be a set composed of the name of the parameters passed by ownership (i.e. those annotated with `@own`). For instance, let f be a function signature whose domain $f.dom$ is defined as follows:

$$f.dom = \langle x, y \mapsto \langle qual \mapsto \texttt{brw}, type \mapsto \mathbb{A}\rangle, z \mapsto \langle qual \mapsto \texttt{own}, type \mapsto \mathbb{A}\rangle\rangle$$

Then $P_f^{\texttt{brw}} = \{x, y\}$ and $P_f^{\texttt{own}} = \{z\}$.

Since assigning the return identifier $\mathfrak{R}$ within the function's body is nonsensical, one could take the same approach as that we used to infer atomic literals' scopes, and assume $\mathfrak{R} \sqsubset -\infty$ in all function bodies. Conversely, the lifetime of arguments passed by alias can be assumed greater than that of any local variable, or argument passed with ownership. As a result, one could simply consider all

arguments passed by reference to live in a scope with a rank $+\infty$.

Function's scoping context (attempt)

$$\begin{aligned} \text{funscope}(\tau) = \Sigma \implies & \quad \forall p \in P_\tau^{\texttt{own}}, -\infty < \Sigma.p < \infty \\ \wedge & \quad \forall p \in P_\tau^{\texttt{brw}}, \Sigma.p = +\infty \\ \wedge & \quad \Sigma.\mathfrak{R} = -\infty \end{aligned}$$

The procedure computes a scoping context that maps each borrowed argument to $+\infty$, and each owned argument to some rank in $\mathbb{N}$. Note that the value onto which owned parameters are mapped does not need to satisfy any relation with the rank in which the function declaration is evaluated. As functions are not allowed to capture closures, they can in fact be type checked in a completely unrelated scoping context.

Unfortunately, this approach fails to type check return values, because it does not take lifetime bounds into account. Since all borrowed parameters are assumed living in a scope $+\infty$, type checking return statements will always succeed, whereas returning an alias to a parameter whose name does not appear in the lifetime bounds annotation should result in a type error. A better solution is to choose a value of $\mathfrak{R}$ such that all borrowed parameters that appear in the lifetime bound are mapped to a greater value, and all others are mapped to a smaller value, including owned parameters. The intuition is that only borrowed parameters specified in the bounds annotation should be assumed to live longer than the function call. Although this does no longer prevent the return identifier to appear as a right operand, writing such assignment is in fact syntactically ill-formed (i.e. $x \,\&\!-\, \mathfrak{R} \notin \mathcal{AC}$). Additional constraints must also be set to guarantee that all owned parameters are assumed living shorter than borrowed ones.

Function's scoping context

$$\begin{aligned} \text{funscope}(\tau) = \Sigma \implies & \quad \forall p \in P_\tau^{\texttt{own}}, \forall q \in P_\tau^{\texttt{brw}}, \Sigma.p < \Sigma.q \\ \wedge & \quad \forall p \in \tau.bds, \Sigma.\mathfrak{R} < \Sigma.p \\ \wedge & \quad \forall p \in \text{dom}(\tau.dom), p \notin \tau.bds \implies \Sigma.p < \Sigma.\mathfrak{R} \end{aligned}$$

Then, we can define scope checking for function declarations. Notice that the function's body is type checked in a scope with a shorter lifetime than any other parameter, so as to guarantee that local variables get a shorter lifetime.

Function declarations

S-Fun

$$\frac{(\lambda \overline{x} \triangleright t) : \tau \qquad \Sigma' = \text{funscope}(\tau) \qquad \Sigma', \min(\{\Sigma'.x \mid x \in \overline{x}\}) - 1 \vdash t \sqsubset j}{\Sigma, i \vdash \lambda \overline{x} \triangleright t \sqsubset -\infty}$$

Example 6.3.6: Parameter scope checking

Consider the function below. The two first parameters `a` *and* `b` *appear in the lifetime bound of the function's codomain. The third parameter* `c` *does not appear in the lifetime bound. The two last parameters* `x` *and* `y` *are expected to receive ownership.*

$\mathcal{A}$-calculus

```
fun (
  a: @brw, b: @brw, c: @brw, x: @own, y: @own)
  -> @brw(a,b)
{
  // ...
}
```

The application of S-Fun *requires to find a scoping context* Σ *such that:*

- `x` *and* `y` *are mapped to a smaller value than that of* `a`, `b` *and* `c`,
- `x` *and* `y` *are mapped to a smaller value than* $\Sigma.\mathfrak{R}$, *and*
- `a` *and* `b` *are mapped to a greater value than* $\Sigma.\mathfrak{R}$.

The constraint system is satisfiable with, for instance, $\Sigma = \{a \mapsto i+1, b \mapsto i+1, \mathfrak{R} \mapsto i, x \mapsto i-1, y \mapsto i-1\}$. *Therefore the function declaration is well-typed.*

As S-Fun sets a virtual scope in which the return identifier is assumed to live, return statements can be scope checked like regular assignments.

Function returns

$$\textsc{S-Ret}\ \frac{\Sigma, i \vdash \mathfrak{R} \diamond t \sqsubset j}{\Sigma, i \vdash \texttt{ret} \diamond t \sqsubset j}$$

Scope checking parameter assignments is unnecessary. Since all parameters are assumed living in a nested scope, those must always succeed. However, the scope in which arguments live is required to determine that of the call, if the callee's codomain is annotated with the `brw` qualifier. Otherwise the call is assumed living in a nested scope.

Function calls

S-Call

$$\frac{f : \tau \qquad \forall (x \diamond t) \in \overline{u}, \Sigma, i \vdash t \sqsubset j_x}{\Sigma, i \vdash f(\overline{u}) \sqsubset \max(\min(\{j_x \mid x \in \tau.bds\} \cup \{+\infty\}), i - 1)}$$

Example 6.3.7: Scope checking function calls

Consider the following term:

$$f(x \;\&\text{-}\; a, y \;\&\text{-}\; b, z \leftarrow c)$$

Assume f to be an identifier typed with a function signature τ defined as follows:

$$\begin{aligned} \tau.dom.x &= \langle qual \mapsto \texttt{brw}, type \mapsto \mathbb{A} \rangle \\ \tau.dom.y &= \langle qual \mapsto \texttt{brw}, type \mapsto \mathbb{A} \rangle \\ \tau.dom.z &= \langle qual \mapsto \texttt{own}, type \mapsto \mathbb{A} \rangle \\ \tau.codom &= \langle qual \mapsto \texttt{brw}, type \mapsto \mathbb{A} \rangle \\ \tau.bds &= \{x, y\} \end{aligned}$$

In other words, f is a function that accepts three parameters x, y and z, such that x and y are borrowed while z is owned, and returns either x or y by alias. Assume the f's call is type checked in a scoping context Σ, defined as follows:

$$\Sigma = \{a \mapsto i, b \mapsto i - 1, c \mapsto i - 2\}$$

Then the scope checking of the function call is described by the following of derivation:

$$\frac{f : \tau \quad \dfrac{a \in \mathrm{dom}(\Sigma)}{\Sigma, i - 2 \vdash a \sqsubset i} \quad \dfrac{b \in \mathrm{dom}(\Sigma)}{\Sigma, i - 2 \vdash b \sqsubset i - 1} \quad \dfrac{c \in \mathrm{dom}(\Sigma)}{\Sigma, i - 2 \vdash c \sqsubset i - 2}}{\Sigma, i - 2 \vdash f(x \;\&\text{-}\; a, y \;\&\text{-}\; b, z \leftarrow c) \sqsubset \max(\min(\{i, i - 1, +\infty\}), i - 3)}$$

The scope checking concludes that the call lives in a scope $\max(\min(\{i, i - 1, +\infty\}), i - 4) = i - 1$*, which corresponds to the scope associated with the argument b, passed as the second parameter y.*

6.3.3 Flow-Sensitive Types

We now move on to the flow-sensitive part of the type checking. Unlike its flow-insensitive counterpart, the flow-sensitive type of a reference evolves along with the program. In particular, it indicates whether a reference can be safely dereferenced, aliased or moved at some given point of the program. We use type capabilities to describe this information.

Type Capabilities

Each reference is associated with a type capability. A type capability describes whether a particular reference is assigned to a location, and whether it is that location's owner or if it is an alias. A non-owning reference's capability further indicates the owners from which the reference is borrowed. We keep track of all type capabilities in a typing environment.

Definition 6.3.4: Type capabilities

Let $\mathbb{X}$ be the set of reference identifiers. The set of type capabilities $\mathbb{C}$ is defined as follows:

$$\mathbf{o} \in \mathbb{C} \qquad \textit{denotes ownership}$$
$$X \subseteq \mathbb{X} \implies (\mathbf{b}, X) \in \mathbb{C} \qquad \textit{denotes borrowing}$$

A borrowed capability $(\mathbf{b}, X)$ is associated with a parameter X that designates the owners from which the reference is assumed borrowed. While at runtime a reference can never borrow ownership from more than one owner, recall that static nondeterminism forces the type system to consider situations in which ownership *might* have been borrowed from different owning references.

Example 6.3.8: Type capabilities

Consider the term below. Next to each line are inscribed the capabilities associated with each reference, using the notation $r \mapsto c$ to denotes that a reference r holds capability c, where $\ominus$ denotes the absence thereof.

1	`let` x `in`	$x \mapsto \ominus$
2	`let` y `in`	$x \mapsto \ominus, y \mapsto \ominus$
3	`let` z `in`	$x \mapsto \ominus, y \mapsto \ominus, z \mapsto \ominus$
4	$x \leftarrow 42$;	$x \mapsto \mathbf{o}, y \mapsto \ominus, z \mapsto \ominus$
5	y `&-` x ;	$x \mapsto \mathbf{o}, y \mapsto (\mathbf{b}, \{x\}), z \mapsto \ominus$
6	z `&-` y	$x \mapsto \mathbf{o}, y \mapsto (\mathbf{b}, \{x\}), z \mapsto (\mathbf{b}, \{x\})$

After their declaration, variables are left with no capabilities, since they

have yet to be assigned. At line 4, x obtains ownership as it is assigned to the value 42*. y obtains a borrowed reference on x at line 5, which is reflected by the capability* $(\mathbf{b}, \{x\})$*. z reborrows y's reference at line 6, and therefore also obtains a* $(\mathbf{b}, \{x\})$ *capability.*

6.3.4 Type Capabilities Checking

The inscriptions next to each line in Example 6.3.8 correspond in fact to the typing environment we mentioned before. We call this environment a *type capability context* (or just capability context for short), and define it as follows:

Definition 6.3.5: Type capability context

Let $\mathbb{X}$ *be the set of reference identifiers. A type capability context is a table* $K : \mathbb{X} \nrightarrow \mathbb{C} \cup \{\ominus\}$ *that maps identifiers to capabilities, or lack thereof.*

Notation: Reference Uniqueness

Given an identifier x and type capability context K, we write $x!K$ to denote that x is unique in the context K. The property holds if x is owner and there is no borrowed capability on x in K. More formally:

$$x!K \Leftrightarrow K.x = \mathbf{o} \wedge \nexists y \in \mathrm{dom}(K), K.y = (\mathbf{b}, X) \wedge x \in X$$

Note that determining uniqueness this way is only possible for identifiers. For all intent and purposes, this is sufficient because references cannot borrow from other constructions (e.g. borrowing from a literal value). Moreover, since those cases are already rejected by the flow-insensitive type checking step, they can be safely ignored here.

We describe type capabilities checking with inference rules. Typing judgements are of the form $K \vdash t \sqsubset c, K'$, where K and K' are the capability contexts before and after evaluating the term t, respectively, and $c \in \mathbb{C} \cup \{\ominus\}$ is the capability held by the reference to which t evaluates.

Variables and Literals

Variable declarations must insert the declared name into the capability context, indicating that it does not have any capability yet. Remark that we do not need to restore the previous mapping for x in K', as we make the assumptions that the programs we type check are free from name shadowing. Therefore it is safe to assume that $x \notin \mathrm{dom}(K)$.

Variable declarations

$$\textsc{K-Let}\ \frac{K[x \mapsto \ominus] \vdash t \Downarrow c, K'}{K \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow c, K'|_x}$$

Dereferencing an identifier only requires that it has either of the capabilities. If it does not, then the reference is either unallocated or moved.

Variable dereferencing

$$\textsc{K-Var}\ \frac{x \in \mathrm{dom}(K)}{K \vdash x \Downarrow K.x, K}$$

Recall that the ownership of a memory location allocated by a literal's evaluation is given to a new virtual owner, assumed to be living in some nested scope. It is therefore safe to assume that atomic literals always represent unique references.

Literal values

$$\textsc{K-Atom}\ \frac{a \in \mathbb{A}}{K \vdash a \Downarrow \mathbf{o}, K} \qquad \textsc{K-Fun}\ \frac{}{K \vdash \lambda\overline{x} \triangleright t \Downarrow \mathbf{o}, K}$$

Assignments

All assignments prescribe that the right operand be initialized. In other words, it must hold either of the type capabilities. This is carried over by first type checking the right operand, before verifying that the resulting capability is not $\ominus$.

Mutation assignments are the simplest to type check, because they do not impose any additional constraint on the right operand. However, two situations must be considered on the other side. If the left operand already has a capability, nothing changes in the capability context. If the left operand has no capability, then an allocation has to be inserted and the reference gets ownership and uniqueness.

Mutation assignments

$$\textsc{K-Mut-A}\ \frac{K \vdash t \Downarrow c, K' \quad c \neq \ominus \quad K'.x = \ominus}{K \vdash x := t \Downarrow \mathbf{o}, K'[x \mapsto \mathbf{o}]} \qquad \textsc{K-Mut-B}\ \frac{K \vdash t \Downarrow c, K' \quad c \neq \ominus \quad K'.x \neq \ominus}{K \vdash x := t \Downarrow K'.x, K'}$$

Example 6.3.9: Mutation assignments

Consider a simple mutation assignment $x \coloneqq 1$. *In a context* $K_A = \{x \mapsto \ominus\}$, *the assignment will be typed by the rule* K-Mut-A, *because* x *has no capability, indicating that it has yet to be initialized.*

$$\frac{K_A \vdash 1 \Downarrow, \mathbf{o}, K_A \qquad \mathbf{o} \neq \ominus \qquad K_A.x = \ominus}{K_A \vdash x \coloneqq 1 \Downarrow \mathbf{o}, K_A[x \mapsto \mathbf{o}]}\ \text{K-Mut-A}$$

In a context $K_B = \{x \mapsto (\mathbf{b}, X)\}$, *the assignment will be typed by the rule* K-Mut-B, *because* x *has the borrowing capability, indicating that it represents a value that must be mutated.*

$$\frac{K_B \vdash 1 \Downarrow, \mathbf{o}, K_B \qquad \mathbf{o} \neq \ominus \qquad K_B.x = (\mathbf{b}, X)}{K_B \vdash x \coloneqq 1 \Downarrow (\mathbf{b}, X), K_B}\ \text{K-Mut-B}$$

Move and aliasing assignments are slightly more complex because they must take care of right operands' capabilities bookkeeping. If the right operand is borrowed, then the capability context's update is related to the borrowed owners, which can be retrieved from the capability itself. On the other hand, if the right operand is unique, then two situations must be considered:

1. The right operand is an identifier, in which case the capability context can be updated directly.

2. The right operand is another term, in which case we do not know which owner's capability must be updated.

Fortunately, cases where the right operand is a literal or a call to a function returning with ownership can be ignored, because they either correspond to a virtual owner or to an owner from a function scope. In both cases, such owners cannot be referenced again, which is why they do not even appear in the type capability context in the first place. Move assignments can therefore safely transfer ownership to their left operand, and aliasing assignments to such references are already rejected during the flow-insensitive typing step.

Remember that an owner cannot be moved to itself. Since a move can only happen if the right operand has uniqueness, this situation is only possible if the same identifier appears on both sides of a move assignment, which is trivially detected. An owner cannot alias itself either. If the right operand is an owner with uniqueness, guaranteeing this constraint consists in making sure both operands are not the same identifier. If the right operand is a borrowed reference, then the identifier on the left should not appear in the set of owners defined by the borrowed capability.

Move assignments (from identifiers)

K-Move-A
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x = \ominus}{K \vdash x \leftarrow y \Downarrow \mathbf{o}, K'[x \leftarrow \mathbf{o}, y \mapsto \ominus]}$$

K-Move-B
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x \neq \ominus}{K \vdash x \leftarrow y \Downarrow K'.x, K'[y \mapsto \ominus]}$$

Move assignments (from calls and literals)

K-Move-C
$$\frac{K \vdash t \Downarrow c, K' \quad t \notin \mathbb{X} \quad c = \mathbf{o} \quad K'.x = \ominus}{K \vdash x \leftarrow t \Downarrow \mathbf{o}, K'[x \leftarrow \mathbf{o}]}$$

K-Mut-D
$$\frac{K \vdash t \Downarrow c, K' \quad t \notin \mathbb{X} \quad c = \mathbf{o} \quad K'.x \neq \ominus}{K \vdash x \leftarrow t \Downarrow K'.x, K'}$$

Aliasing assignments

K-Alias-A
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x \neq \mathbf{o} \vee x!K'}{K \vdash x \,\&\!\!-\, y \Downarrow \mathbf{o}, K'[x \leftarrow (\mathbf{b}, \{y\})]}$$

K-Alias-B
$$\frac{K \vdash t \Downarrow c, K' \quad c = (\mathbf{b}, O) \quad x \notin O \quad K'.x \neq \mathbf{o} \vee x!K'}{K \vdash x \,\&\!\!-\, t \Downarrow \mathbf{o}, K'[x \leftarrow (\mathbf{b}, O)]}$$

Function Calls

Like in the operational semantics, we decompose function calls' capabilities checking into three steps, namely consisting the type checking of the callee, of the arguments and finally of the function's body. The first step does not require any additional machinery, as it ensures that the callee can be dereferenced. Handling arguments on the other hand requires more attention.

All parameter assignments need to be type checked. However, since the parameters' names either do not exist or must be shadowed in the capability context, we first need to prepare the context so that the assignments can be properly checked. Furthermore, if the codomain of the function reveals a return-by-alias

strategy, the arguments' capabilities must also be collected, as they will serve to determine the owners' set of the resulting borrowed capability. Unlike in the operational semantics, the callee's evaluation cannot be leveraged to retrieve the parameter names. Fortunately, those appear in the function's signature. We define a judgement of the form $K, A \vdash \overline{u} \Downarrow^{\text{args}} K', A'$, where $\overline{u}$ is a parameter assignments, represented as a word, K and K' the type capability contexts before and after the checking of the sequence, and finally A and A' are tables $\mathbb{X} \twoheadrightarrow \mathbb{C}$ that maps parameter names to the capability of their arguments.

Argument lists

$$\textsc{K-Args-0}\ \frac{}{K, A \vdash \epsilon \Downarrow^{\text{args}} K, A} \qquad \textsc{K-Args-N}\ \frac{y \notin \text{dom}(K) \qquad K[y \mapsto \ominus] \vdash y \diamond t \Downarrow c_x, K_x \qquad K_x, A[x \mapsto c_x] \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K, A \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} K', A'}$$

The final step is to compute the capabilities of the return value. If the function returns with ownership, the call is associated with the ownership capability. On the other hand, if it returns by alias, then we need to compute the union of all borrowed arguments' owners. Two cases should be considered. If the borrowed argument is an owning reference, then it must be added to the returned capability. If the borrowed argument is a borrowed reference, then its *owners* must be added to the returned capability. We define a procedure owners to handle these two situations, defined as below:

$$\text{owners}(x, A) = \begin{cases} \{x\} & \text{if } A.x = \mathbf{o} \\ X & \text{if } A.x = (\mathbf{b}, X) \end{cases}$$

We further define a procedure $\text{owners}^{\cup}$ that extends owners to a set of identifiers. More formally, $\text{owners}^{\cup}(X, A) = \bigcup_{x \in X} \text{owners}(x, A')$.

Function calls

$$\textsc{K-Call-A}\ \frac{f : \tau \quad \tau.codom.qual = \texttt{own} \qquad K \vdash f \Downarrow c_f, K_f \quad c_f \neq \ominus \quad K_f, \varnothing \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K \vdash f(\overline{u}) \Downarrow \mathbf{o}, K'}$$

$$\textsc{K-Call-B}\ \frac{f : \tau \quad \tau.codom.qual = \texttt{brw} \qquad K \vdash f \Downarrow c_f, K_f \quad c_f \neq \ominus \quad K_f, \varnothing \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K \vdash f(\overline{u}) \Downarrow (\mathbf{b}, \text{owners}^{\cup}(\tau.codom.bds, A')\}), K'}$$

Conditional Terms

As discussed above, conditional terms pose the problem of static nondeterminism, which we resolve with conservative assumptions. Hence, a first step is to type check both branches, so that the resulting capabilities contexts can be compared. Note that if K_t and K_e respectively denote the capability contexts obtained after evaluating each branch of a conditional term, then $\text{dom}(K_t) = \text{dom}(K_e)$ is an invariant as variables and parameters are never accessible beyond the scope that declares them.

We define a procedure $\text{merge}(K_t, K_e) = K'$ that attempts to find a capability context K' that satisfies our conservative constraints, given two contexts K_t and K_e. The constraints are defined as follows:

1. If a reference is allocated at the end of one branch but not at the end of the other, it is assumed unallocated after the conditional term. More formally:

$$\forall x \in \text{dom}(K_t), (K_t.x = \ominus \vee K_e.x = \ominus) \implies K'.x = \ominus$$

2. If a reference has ownership at the end of one branch but not at the end of the other, it is transient in the the former, and assumed unallocated after the conditional term. More formally:

$$\forall x \in \text{dom}(K_t), (K_t.x = \mathbf{o} \wedge K_e.x \neq \mathbf{o}) \vee (K_t.x \neq \mathbf{o} \wedge K_e.x = \mathbf{o}) \implies K'.x = \ominus$$

3. If a non-transient owner is aliased at the end of one branch, it is assumed aliased after the conditional term. More formally:

$$\forall x \in \text{dom}(K_t), (K_t.x = (\mathbf{b}, O_1) \vee K_e.x = (\mathbf{b}, O_2)) \implies K'.x = (\mathbf{b}, O_1 \cup O_2)$$

4. If a reference gets ownership at the end of both branches, it is assumed holding ownership after the conditional term. More formally:

$$\forall x \in \text{dom}(K_t), (K_t.x = \mathbf{o} \wedge K_e.x = \mathbf{o}) \implies K'.x = \mathbf{o}$$

5. A reference borrowed from a transient owner cannot survive after the conditional term. More formally:

$$\forall x \in \text{dom}(K_t), K'.x \neq \ominus \implies \nexists y \in \text{dom}(K'), K'.y = (\mathbf{b}, O) \wedge x \in O$$

We then define capability checking for conditional terms as follows. Notice that conditional terms do not bear any type capability, as indicated by the conclusion of the rule. This effectively prevents conditional terms to be used as a

first-class expression. Lifting this restriction would entail additional constraints to *merge* the capabilities of both branches, in particular to handle cases where they would both produce different owning references.

Conditional terms

K-Cond

$$\frac{K_1 \vdash t_2 \Downarrow c_t, K_t \qquad K \vdash t_1 \Downarrow c_1, K_1 \qquad K_1 \vdash t_3 \Downarrow c_e, K_e \qquad K' = \text{merge}(K_t, K_e)}{K \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow \ominus, K'}$$

6.3.5 Key Properties

We now discuss the key properties of the type system, in the context of memory safety. We start by defining the notion of well-formedness for capability contexts.

Definition 6.3.6: Well-formedness of capability contexts

A capability context K is well formed if and only if owners of a borrowed reference in K are owners in K. More formally:

$$\forall x \in \text{dom}(K), K.x = (\mathbf{b}, O) \implies O \subseteq \text{dom}(K) \land \forall y \in O, K.y = \mathbf{o}$$

Example 6.3.10: Ill-formed capability context

Consider the capability context:

$$K = \{x \mapsto (\mathbf{b}, \{y\}), y \mapsto (\mathbf{b}, \{z\}), z \mapsto \mathbf{o}\}$$

K is ill-formed because it states that x is a borrowed reference, whose owner is y, yet y is not owner, that is $K.y \neq \mathbf{o}$.

The reader will notice that the definition of well-formedness implies that borrowed references cannot borrow from themselves.

Lemma 6.3.1. *If K is a well-formed capability context, then borrowed references in K do not borrow from themselves. More formally, $\forall x \in \text{dom}(K), K.x = (\mathbf{b}, O) \implies x \notin O$.*

Proof. The proof is by contradiction. Let K be a well-formed capability context. Assume x borrows from itself, that is $K.x = (\mathbf{b}, O) \land x \in O$. Then x is not an owner in K, which contradicts the definition of well-formedness. □

A first key property of the type system is that it preserves well-formedness, as formalized by the following property. The full proof is given in Appendix C.

Property 6.3.2 (Well-formedness preservation)**.** *Let K be a well-formed capability context. If $K \vdash t \Downarrow c, K'$, then K' is well-formed.*

Proof Sketch. The proof is by induction over the typing judgements. □

Two other key properties relates to aliasing safety.

Property 6.3.3 (Moving safety)**.** *Let K be a well-formed capability context. If $K \vdash x \leftarrow y \Downarrow c, K'$ and $y \in \mathbb{X}$, then $y!K$.*

Proof. The proof is trivial by K-Move-A and K-Move-B. □

Property 6.3.3 states that the right operand of a move assignment has to be a unique owner. Consequently, this guarantees that a move assignment cannot leave aliases dangling, as shared owners (i.e. owners that are not unique in a given context) cannot be moved.

Property 6.3.4 (Reassignment safety)**.** *Let K be a well-formed capability context. If $K \vdash x \mathbin{\&-} t \Downarrow c, K'$ and $K.x = \mathbf{o}$, then $x!K$.*

Proof. The proof is trivial by K-Alias-A and K-Alias-B. □

Property 6.3.4 states that reassigning an owning reference implies that it is unique. Consequently, this guarantees that an aliasing assignment cannot leave other aliases dangling, as shared owners cannot be reassigned.

The next key property relates to outliving aliases, and guarantees that non-owning references always live shorter than the owner from which they borrow. We first state two lemma to establish its proof. First, an identifier is never defined in the context that results from the evaluation of its declaration.

Lemma 6.3.5. *If $K \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow c, K'$, then $x \notin \mathrm{dom}(K')$.*

Proof. The proof is trivial by K-Let. □

Secondly, scope typing guarantees that if a variable y is declared in the body of a variable x's declaration, then the rank of the scope in which y lives is smaller than that in which x lives.

Lemma 6.3.6. *Assume $\Sigma, i \vdash \texttt{let}\ y\ \texttt{in}\ t \sqsubset j$. Then $\forall x \in \mathrm{dom}(\Sigma), \Sigma.x \geq i$.*

Proof. The proof is trivial by K-Let. □

Property 6.3.7 (Outliving aliases freedom)**.** *Let K be a well-formed capability context. If $K \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow c, K'$, then $\nexists y \in \mathrm{dom}(K'), K'.y = (\mathbf{b}, O) \wedge x \in O$.*

Proof. The proof is by contradiction. Assume $K \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow c, K'$ and $\exists y \in \mathrm{dom}(K'), K'.y = (\mathbf{b}, O \cup \{x\})$. By Lemma 6.3.5, $y \in \mathrm{dom}(K') \implies y \in \mathrm{dom}(K)$, that is y is already defined before x's declaration. By Lemma 6.3.6, $\Sigma, i \vdash \texttt{let}\ x\ \texttt{in}\ t \sqsubset j \implies \Sigma.y \geq i$, that is y lives in a scope with a higher rank than x. Since y is an alias on x, there must be a subterm of t that is an aliasing assignment of the form $y\ \texttt{\&-}\ z$ such that either $z = x$ or z is an alias on x.

- If $z = x$, then $\Sigma, i \nvdash \texttt{let}\ x\ \texttt{in}\ t \sqsubset j$ because $x \sqsubset i - 1$, and S-Alias does not apply.
- If $z \neq x$, then there exists $z \in \mathrm{dom}(\Sigma)$ such that $\Sigma.y < \Sigma.z$, and therefore there must be $K'.z = (\mathbf{b}, O \cup \{x\})$. This does not hold, by induction on the first hypothesis.

□

A last key property relates to safety with respect to uninitialized memory error. Our typing rules guarantee that dereferencing cannot occur on references that do not hold any capability. The property is obviously guaranteed for the evaluation of a function call's callee and the guard of a conditional expression.

Lemma 6.3.8 (Safe dereferencing of callees)**.** *Let K be a well-formed capability context. If $K \vdash f(\overline{u}) \Downarrow c, K'$, then $K \vdash f \Downarrow c_1, K_1$ and $c_1 \neq \ominus$.*

Proof. The proof is by trivial by K-Call-A and K-Call-B. □

Lemma 6.3.9 (Safe dereferencing of conditional guards)**.** *Assume K is a well-formed capability context. If $K \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow c, K'$, then $K \vdash t_1 \Downarrow c_1, K_1$ and $c_1 \neq \ominus$.*

Proof. The proof is by trivial by K-Cond. □

Assignments always prescribe that the right operand must hold a capability, which trivially guarantees that dereferencing of uninitialized or moved memory cannot occor on the right side.

Lemma 6.3.10 (Safe dereferencing of right operands)**.** *Let K be a well-formed capability context. If $K \vdash x \diamond t \Downarrow c, K'$, where $\diamond \in \mathbb{O}$ is an assignment operator, then $K \vdash t \Downarrow c_1, K_1$ and $c_1 \neq \ominus$.*

Proof. The proof is by trivial by all assignment rules. □

Although dereferencing occurs on the side of mutation and move assignments, recall that the type system assumes the implicit insertion of memory allocations before references to uninitialized or moved memory are assigned. Consequently, such cases can be discarded.

Property 6.3.11 (Safe dereferencing)**.** *Let K be a well-formed capability context. Then dereferencing a reference r used as the callee of a function call, the guard of a conditional term or the right operand of an assignment implies that $K.r \neq \ominus$.*

Proof. The proof is by Lemma 6.3.8, Lemma 6.3.9 and Lemma 6.3.10. □

6.3.6 Soundness Result

We can build upon the properties enunciated in the previous section to express a strong soundness result. More formally, we define soundness with the following theorem.

Theorem 6.3.12 (Soundness Result)**.** *If a program $t \in \mathcal{AC}$ can be type-checked, that is $\Sigma, i \vdash t \subseteq j$ for some flow-insensitive typing context Σ and $K \vdash x \diamond t \Downarrow c, K'$ for some well-formed flow-sensitive typing context K, then $C \vdash t \Downarrow l, C'$ for some evaluation context C without memory errors.*

The formal proof of this theorem cannot be established directly solely using our typing jugements description of the operational semantics. The problem is that our type system assumes that memory allocations and deallocations are inserted implicitly, and can therefore elude some of the cases that would, under the definition proposed in Chapter 4, lead to a memory error.

Example 6.3.11: Ill-formed capability context

Consider the following program:

$$\texttt{let}\ x\ \texttt{in}\ x \leftarrow 42$$

This program's evaluation should trigger an uninitialized memory error, as x is assigned before being allocated. However, our type system will not fail to typecheck this program. More specifically, the rule K-Move-A *applies, as x does not hold any type capability, and gives it ownership in the resulting context. The explanation of this behavior lies in the fact that, as described in Section 6.2, an allocation can be automatically inserted before an assignment if the left operand is unallocated.*

Consequently, it follows that our type system alone is not sufficient to show a strong soundness result, as a formal description of the algorithm that inserts allocation and deallocation is required. Moreover, this algorithm relies on typing information that should be inferred by our type system, namely reference ownership. Hence, a solution to break this circularity would be to devise a deduction system to first infer flow-sensitive types, which would be used by the static memory management algorithm to transform the input, and to establish soundness on

this modified input. While such an approach is beyond the scope of this work, we can nonetheless illustrate how our type system guarantees the absence of memory errors intuitively, by the means of example, to show how the memory errors we formalized in Chapter 5 can be prevented.

Uninitialized and moved memory errors Recall that uninitialized (resp. moved) memory errors occur when a reference being dereferenced has yet to be allocated (resp. refers to moved memory). For instance, such a situation may occur in the process of a move assignment's evaluation, as illustrated in the example of Section 5.1.1. However, such an assignment will be rejected by our type system, which prescribes that the right operand be unique, and therefore initialized.

Example 6.3.12: Uninitialized memory errors

Consider the following faulty program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ x \leftarrow y$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto l_1 & l_1 \mapsto \bot \\ y \mapsto 0 & \end{array}$$

Applying the rule E-Move *fails, as $C \vdash y \downarrow l, C$ yet $l = 0$, indicating an uninitialized memory error. However, the flow-sensitive typing context K that type checks this assignment is given by $K = \{x \mapsto \mathbf{o}, y \mapsto \ominus\}$. Consequently,* K-Move-B *cannot apply, as the premise $y!K$ is not be satisfied. Note that the other type rules for move assignments do not apply neither, either because x is already allocated, therefore invalidating $K.x = \ominus$, or because y is an identifier, therefore invalidating $y \notin \mathbb{X}$.*

Use after free and doubly freed memory errors Use after free errors occur when a variable that has been freed is dereferenced again. Lexical scoping already prevents variables defined in a particular scope (typically delimited by a `let ⋯ in` construct) from being accessed outside of said scope. However, our operational semantics does not prevent visible variables referring to deallocated memory to be used again.

Two mechanisms prevent this situation. First, recall that variables that hold ownership at the end of their lexical scope are automatically deallocated.

Hence, by removing explicit deallocations from the language, situations where such an instruction would be called before an owning reference is dereferenced again are de facto prevented. Similarly, this also protects a program from doubly freed memory. Secondly, aliases on the memory to which they were assigned must be guaranteed not to be dereferenced again. Our flow-insensitive typing rules guarantee this invariant by forbidding aliases to variables bound to scope that lives shorter, as illustrated by Example 6.3.3.

Memory leaks Our type system guarantees that all references are either owning, or owned by an owning reference, as formalized by the well-formedness property. Thanks to the automatic insertion of deallocation instructions for all owning references, it follows that all allocated memory must be freed eventually, which prevents memory leaks.

Example 6.3.13: Memory leak prevention

Consider the following program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ x \leftarrow 42\ ;\ y \leftarrow 1337$$

The type checking of this program gives ownership to x before it is assigned to the value 42, based on the assumption that an allocation will be automatically inserted at this point. Similarly, ownership is given to y before the second assignment.
It is then assumed that deallocation instructions for both y and x will be inserted, as these will have been determined to go out of scope with ownership.

Limitation

We draw the reader's attention on the fact that our type system is not complete. In other words, there exist programs whose execution would not fail under the operational semantics of Chapter 4, but that cannot be checked by our type system. This is due to the conservative assumptions that have to be made, in order to statically track type capabilities beyond function boundaries and across conditional terms.

Example 6.3.14: False positive

Consider the following program:

$$\texttt{let}\ x\ \texttt{in}\ \texttt{let}\ y\ \texttt{in}\ y \leftarrow 42\ ;\ y\ \texttt{\&-}\ x$$

For the sake of conciseness, we skip grayed out terms to focus solely on the assignment, and assume it is evaluated in a context C such that:

$$C = \begin{array}{c|c} \pi & \mu \\ \hline x \mapsto 0 & l_1 \mapsto \bot \\ y \mapsto l_1 & \end{array}$$

Although our flow-insensitive typing rules fail to type check this program, due to the fact that y lives in a shorter scope than x, this program will not not crash at runtime because x is not read after y is deallocated. In other words, while this program creates a dangling reference, it never dereferences it.

6.4 Records and Static Garbage Collection

Manipulating records invites a plethora of complications, and would require the definition of additional constructions to guarantee memory safety. A first consideration asks how a field can be guaranteed to be initialized statically. Consider for instance the following function:

$\mathcal{A}$-calculus

```
fun f(cond: @ref) -> @own {
  let x in {
    x <- new <m>
    if cond {
      x.m <- 12
      return <- x
    } else {
      return <- x
    }
  }
}
```

Keep in mind that functions are considered as black boxes. Consequently from the call site, there is no way to determine whether the member `m` of `f`'s return value is initialized. Annotations could be used to solve this issue, but fail in the presence of linked data structure. Imagine for instance a linked list implemented by a chain of records, each representing one particular node and referring by alias to the next one. Annotations cannot help, because such a linked list returned by alias could have an arbitrary depths, which cannot be determined statically. The inability to statically determine the state of a particular field poses other problems as well. In fact, none the rules we described above can be safely applied.

Real programming languages solve this issue by the means of two mechanisms: definite initialization [64] and optional (a.k.a. nullable) types. Definite initialization is a static analysis technique that aims to guarantee that a variable is initialized at a given point, no matter what happens in the program before. Roughly speaking, it consists of checking that it is initialized in all possible execution paths[4]. Definite initialization can be used to guarantee that a particular function necessarily initializes all fields of a record. If in addition record allocations are contained within such functions, usually called *constructors*, then a record can be used safely anywhere else. Nonetheless this approach raises a question as to how linked and self-referential data structures can be handled. Indeed, if definite initialization prescribes that all fields be initialized before a constructor can return, then assigning a self-referential field is impossible. This is where optional types come handy. Optional types denote values that may or may not exist, allowing definite initialization to ignore fields that should be initialized after the constructor. As a result, sequences and cycles of references can be broken.

Example 6.4.1: Constructor

Assume a hypothetical value `nil` *denoting optional values. Then, initializing a node for a linked list could be done with the following a constructor, defined as follows:*

$\mathcal{A}$-calculus

```
ctor <- fun(v: @own) -> @own {
  let n in {
    n <- new <val,nxt>
    n.val <- v
    n.next <- nil
    return <- n
  }
}
```

Although optional types enable definite initialization analysis, they do not solve static safety. In fact, they only help the type system identify *where* initialization cannot be assumed statically, while runtime checks are still the only way to test the validity of a field. As it stands, the $\mathcal{A}$-calculus cannot express such checks, because the language's semantics states that any access to a reference's value requires a dereference, which will obviously fail if memory is unallocated. We therefore need the addition of a construct `exists(x)` for this task, which checks

[4]The reader will remark that the rules described to handle static nondeterminism actually implement a form of definite initialization analysis.

whether a particular reference is initialized (i.e. not in an uninitialized or moved state). This enables conditions to be expressed on the runtime state of a particular reference without dereferencing.

Another consideration relates to garbage collection. As fields are references as well, they shall bear ownership of a particular memory location. However, a field is not bound to a scope lexically, like a variable is. Instead, it lives as a member of a record value, which is itself owned by a particular reference. Consequently, lexical scopes cannot be leveraged to determine fields' deallocations. One solution is to delegate field ownership to the reference holding the record they constitute, thereby establishing ownership trees, necessarily rooted by a lexically scoped variable. Unfortunately this approach cannot deal with arbitrary deeply nested data structures either. It follows that determining how many deallocations should be inserted once the root variable goes out of scope is also impossible. Real programming languages solve this issue with either of two ways, both applied at runtime. Dynamically garbage collected systems simply ignore deallocations altogether, expecting their memory management mechanism to get rid of unused memory. Others handle the deallocation with functions dedicated to the deallocation of a particular record instance, usually called *destructors*. On an arbitrary deeply nested structures, destructors can be applied recursively until reaching the bottom of the structure. Languages featuring static garbage collection (e.g. Rust) adopt the second approach, but insert destructor calls automatically.

Example 6.4.2: Destructor

Assume a hypothetical predicate `exists(x)` *that holds if and only if* `x` *is neither unallocated nor moved. Then, deallocating an arbitrary long linked list could be done with a destructor* `dtor`*, defined as follows:*

$\mathcal{A}$-calculus

```
dtor <- fun(n: @mut @brw, dtor: @brw) {
  if exists(n.next) {
    dtor(n &- n.nxt, dtor &- dtor)
    del n.nxt
  }
  del n.val
}
```

The reader will notice that the deallocation scheme we propose is reminiscent to Ownership Types [39]. Although fields can be owners, they are under the control of the record's owner, that is thereby transitively responsible to deallocate its fields. This means that a record's owner naturally delimits an ownership context [42], that encloses all its members. Although we do not use ownership to

restrict fields' aliasing (within the meaning of an owners-as-$*$ policy), ownership contexts give us an appropriate abstraction to reason about records.

Placing a record on the right hand side of a mutation assignment is always safe, as the transitive copy does not preserve any alias. In other words, not only the ownership context is copied, but also the objects that are aliased from within this context. Updating a record on the other hand requires additional precautions, as to avoid breaking possible aliases to and from its fields. Consider for instance a tree node whose grandchildren are aliased by references from outside its ownership context. If the node's children are mutated, then these aliases will be invalidated. Therefore mutating a record whose fields are aliased externally must be forbidden.

Example 6.4.3: External uniqueness

`z` *gets ownership on a record at line 5, and* `z.m` *borrows a reference on* `y` *at line 6. The situation is valid since* `y` *lives longer than* `z`.

Transferring `z`*'s ownership to* `x` *is illegal, otherwise it would leave* `x.m` *dangling after line 9.*

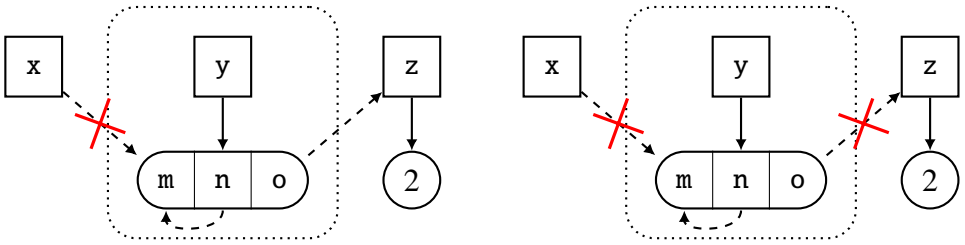
```
1  let x in {
2    let y in
3      y <- 0
4      let z in
5        z <- new <n,m>
6        z.m &- y
7        x <- z
8      }
9    } // <= del y
10 }
```

Moving assignments add yet another constraint, because the validity of its internal aliases may be put in jeopardy. Consider for instance moving the tree to a reference with a longer lifetime, while its children contain aliases referring to references outside of the ownership context. If these references live shorter than the one to which ownership is transferred, children nodes risk containing dangling references. Therefore, moving a node whose fields contain aliases to the outside must be forbidden.

The reader will remark both these strategies correspond to the notions of external [40] and separate [73] uniqueness, whose constraints are illustrated in Figure 6.4.1.

6.5 Capabilities for Immutability

The term "immutability" can be a placeholder for vastly different concepts, depending on the programming language. We briefly describe them to establish the vocabulary we use in the remainder of this chapter. A more comprehensive discussion about immutability is given in [116].



(a) External uniqueness. (b) Separate uniqueness.

Figure 6.4.1: Uniqueness of ownership contexts. Ownership is denoted by solid arrows, while borrowing is drawn dashed. The dotted rounded box symbolizes ownership contexts.

Reassignability is a property of references, indicating whether or not they may be reassigned to another memory location after having being assigned once.

Reference immutability is a property of references that indicates whether or not the memory location to which they refer might be modified *through* them. In other words, an immutable reference is an alias through which the object's mutation is not allowed.

Memory immutability (a.k.a. object immutability) is a restriction on a given memory location. *Shallow immutability* prevents the mutation of the content stored at the location, while *deep immutability* transitively applies to the other memory locations that are referred by the stored object.

While both reassignability and reference immutability can be used to avoid common programming mistakes, they do not pose any restriction on the mutability of the referred memory location itself. Therefore, one might be under the impression to be manipulating immutable objects, while in fact those may be mutated via another reference. Once again, assignment semantics contribute to the confusion. In particular, a non-reassignable reference to a primitive type can easily be misunderstood for an immutable object, as there is no obvious way to modify it (e.g. by the means of a self-modifying method).

Example 6.5.1: Reference immutability vs value immutability

Consider the following C structure.

```
1 typedef struct {
2   int bar;
3 } Foo;
```

`x` is a reference denoting an instance of `Foo`, and `r` is an immutable reference on foo.

Line 8 is illegal because the object cannot be mutated through `r`.

Line 9 is legal, despite the immutable reference `r`.

```
4  void fn() {
5    Foo x;
6    x.bar = 1;
7    Foo const* r = &foo;
8    r.bar = 2;
9    x.bar = 3;
10 }
```

6.5.1 The Single Writer Constraint

Combining reference mutability with uniqueness offers a better protection. For instance, readers cannot coexist with writers in Rust, because mutable references must be unique. However, immutable references can be borrowed arbitrarily many times, with the guarantee that the underlying object is immutable[5]. This is in fact the essence of Rust's mechanism for data safety. The drawback is that self-referencial mutable structures (e.g. mutable graphs or doubly linked lists) can no longer be represented with references.

The single writer constraint is necessary to prescribe data race freedom in a parallel context [35]. However, while this paradigm might seem to be the inevitable direction of modern software development, due the democratization of multi-core architecture, *cooperative multitasking* [49] (a.k.a. non-preemptive multitasking) offers a promising alternative. In this paradigm, concurrent processes do not work in parallel but transfer control between each other over a single core. This differs from multithreading in that control is transferred voluntarily in cooperative multitasking, whereas this task is left to the operating system in the former, more traditional approach. Perhaps counterintuitively, cooperation can outperform its preemptive counterpart in I/O-intensive applications [54]. Performance aside, another advantage is that cooperation offers synchronization for free, which provides a parade against data races and opens the door to more relaxed immutability models. If multiple *cooperative* process agree to share a mutable memory location, concurrent accesses to this location cannot happen simultaneously (within the meaning of multithreading) and data safety is preserved. Nonetheless, this is not to say that immutability is unnecessary. The ability to guarantee the validity of an invariant across successive resumption of a cooperative process is a

[5]The reader proficient in Rust will remark that exceptions to this rule shall be made for types encapsulating an `UnsafeCell` member to implement interior mutability.

desirable property. For instance, a cooperative process could transfer control from the middle of a loop whose iterator depends on an invariant.

6.5.2 Many Readers or Many Writers

We suggest a capability-based type system that relaxes the single writer constraint, intended to bring memory immutability to the $\mathcal{A}$-calclus. The goal is to guarantee that a memory location is at any given time referred to by either n readers and 0 writers, or n writers and 0 readers. Readers are called *non-mutating* references, and refer to immutable memory locations. Writers are called *mutating* references, and refer to mutable memory locations. Two capabilities are defined: the *read-only* capability and the *read-write* capability. Those can be held by references to control the mutability of the value they represent. At declaration, a variable, field or parameter can be either annotated with `@cst` to designate it non-mutating, or with `@mut` to designate it mutating. Unannotated variables, field and parameters are assumed annotated with `@cst`. Non-mutating references receive the *read-only* upon allocation, whereas references to mutable locations receive both and are called *mutating*.

Example 6.5.2: Mutability capabilities

`x` *is declared mutating.*

`x` *gets both mutability capabilities at line 2, so line 3 is legal.*

`y` *is declared non-mutating.*

`y` *gets the read-only capability at line 5, so line 6 is illegal.*

```
1 let x: @mut in {
2   alloc x with <- 2
3   x := 4
4   let y: @cst in {
5     alloc y with <- 2
6     y := 4
7   }
8 }
```

Notice that allocation constructions slightly differ from the regular $\mathcal{A}$-calculus, as they are suffixed with an initial value. Conceptually, `alloc` t_1 `with` $\diamond$ t_2 simply expands to `alloc` x ; $x \diamond t_2$. However, the construction is not only a syntactic sugar. It also serves to atomically represent allocation and initialization, which is necessary to initialize the value of a non-mutating references.

Record fields are references too, and therefore must be qualified as well to indicate whether they are mutating or non-mutating.

Example 6.5.3: Mutability capabilities on fields

`x` is declared mutating.

`x` is allocated and initialized with a record in which `m` is mutating and `c` is not.

Both fields are initialized

Line 6 is legal, as `x.m` is mutating. Line 7 is not, as `x.c` is non-mutating.

```
1 let x: @mut in {
2   alloc x with <- new
3     <m:@mut, c:@cst>
4   alloc x.m with <- 2
5   alloc x.c with <- 4
6   x.m := 8
7   x.c := 16
8 }
```

Non-mutating locations assume the deep immutability of the object to which they refer. In other words, a memory location referred to by a non-mutating reference cannot be modified. Moreover, if that location stores a record, then its fields are considered non-mutating as well, regardless of their declaration.

Example 6.5.4: Transitive immutability

`y` is declared non-mutating.

`x` is allocated and initialized with a record in which `m` is mutating and `c` is not.

Both fields are initialized

Both lines 6 and 7 are illegal, as `y`'s immutability applies transitively.

```
1 let y: @cst in {
2   alloc y with <- new
3     <m:@mut, c:@cst>
4   alloc y.m with <- 2
5   alloc y.c with <- 4
6   y.m := 8
7   y.c := 16
8 }
```

Mutability capabilities are borrowed with the aliasing operator (i.e. `&-`). In order to enforce the multiple readers *or* multiple writers constraint, aliasing is only allowed in two situations:

1. The right operand's mutability is the same as that of the left, that is both references are declared with the same qualifier.

2. The right operand is an unaliased mutating reference.

The latter case allows unique mutating references to be shared non-mutating temporarily (e.g. for the duration of a function call). Recall that mutating references receive both mutability capabilities upon allocation. Hence they can lend the read-only capability to a non-mutating alias, and refrain from using their read-write capability for the duration of the loan. Conversely, an alias is not allocated and only holds the capability it borrows, thus preventing it to be reborrowed mutating. Note that reassigning a field by alias mutates its record. Therefore the operation is obviously prohibited if the record is held by a non-mutating reference.

Example 6.5.5: Mutability and aliasing

`x` *is declared mutating.*

`y` *is declared non-mutating.*

`x` *gets both capabilities at line 3.*

`y` *borrows* `x`*'s read-only capability at line 4 and inhibits* `x`*'s read-write capability. Hence line 5 is illegal.*

`y` *goes out of scope at line 6, so* `x`*'s read-write capability is restored and line 7 is legal.*

```
1 let x: @mut in {
2   let y: @cst in {
3     alloc with x <- 2
4     y &- x
5     x := 4
6   }
7   x := 4
8 }
```

This model shares a few similarities with the notions of uniqueness and ownership we have discussed in the previous sections. Unaliased references act like unique owners and can fraction either of their mutability properties. These fragments are sent back their owner when borrowed references are either reassigned or go out of scope. Consequently, the lifetime bounds mechanism we devised to track uniqueness beyond function boundaries has to be carried over to track mutability capabilities.

Example 6.5.6: Mutability at function boundaries

Function parameters are annotated to indicate their expected mutability.

```
1 fun (
2   x:@brw @mut,
3   y:@brw @mut)
```

The codomain is annotated to indicate that the function returns mutating references that may borrow `x` mutably.

Line 6 is illegal, as `y` is not mentioned in the function's codomain. Conversely, line 7 is legal.

```
4  -> @brw(x:@mut) @mut
5  {
6    return &- y
7    return &- x
8  }
```

6.6 Summary

In this chapter, we have studied the application of memory and aliasing control mechanisms to the $\mathcal{A}$-calculus, in order to guarantee static properties. First, we have bootstrapped our analysis with a review of Rust's type system. Rust promotes linear assignment semantics to guarantee memory and data safety, and therefore solves similar issues as the $\mathcal{A}$-calculus. Next, we have presented a type system to support static garbage collection and enforce memory safety. Lastly, we have suggested additional of capabilities to guarantee immutability, and discussed how the single-writer constraint could be relaxed in the context of single threaded co-operative multitasking.

Chapter 7

Anzen

The last three chapters introduced a minimal calculus for imperative languages, described its formal semantics and proposed dynamic and static approaches to detect and prevent memory errors. Although these results constitute a solid basis to formally analyze existing languages, it remains to see whether the core concept of the $\mathcal{A}$-calculus, namely its different assignment operators, can be integrated into an actual practical programming language. Mixing multiple assignment semantics with high-level programming concepts such as classes and higher-order functions invites complex challenges to preserve sound assumptions on memory management. Another legitimate question asks whether the use of three distinct assignment operators is practical in an actual programming language.

This chapter introduces Anzen, a general purpose language based on the $\mathcal{A}$-calculus. Anzen features all three assignment operators from the $\mathcal{A}$-calculus, and combines them with more elaborate constructs, such as generic composite types and higher-order functions. The language also supports notions of ownership, uniqueness and immutability, although these properties are checked dynamically. A powerful type inference engine allows for most type annotations to be eluded, including those related to memory safety. Input sources are transpiled into an intermediate representation, called the Anzen's Intermediate Representation (AIR), that heavily borrows from Swift Intermediate Language [9] and LLVM's Intermediate Representation [93]. AIR maps high-level programming concepts onto a language close to the $\mathcal{A}$-calculus. This permits a painless implementation of the error detection techniques presented in Chapter 5. Although in its current version the compiler does not generate machine code, an interpreter for AIR can execute Anzen programs.

The remainder of this chapter is divided as follows. We start with a brief review on compiler architectures and fix the nomenclature used throughout the following sections. Section 7.2 proposes a quick tour of Anzen's features to informally introduce its syntax and semantics, before a couple of examples are pre-

sented in Section 7.3. Next, Section 7.4 continues with a description of Anzen's intermediate representation. Finally, Section 7.5 gives an overview of Anzen's compiler architecture and implementation.

7.1 Compilers and Interpreters

A compiler is a program whose primarily objective is to translate source code written in a particular programming language into another programming language. For instance, a C compiler (e.g. Clang [133]) typically turns source code written in C into machine code for a particular processor (e.g. Intel x86). Other compilers may target more abstract languages, such as the bytecode of a virtual machine, or even another human-readable high-level language[1] (e.g. the TypeScript compiler [19]). An interpreter on the other hand is a slightly different piece of software that, instead of translating its input to another language, acts as a virtual machine and executes it directly.

7.1.1 Overview of a Compiler Architecture

A compiler can be seen as a chain of smaller programs, each carrying out a particular part of the whole translation process. Figure 7.1.1 gives a schematic overview.

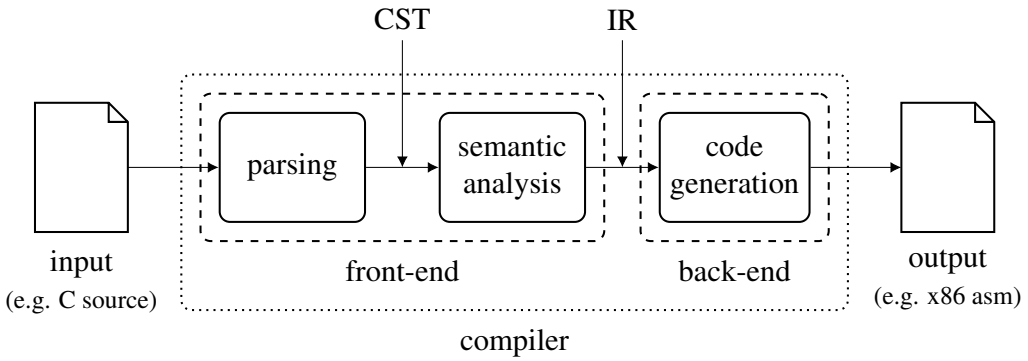


Figure 7.1.1: Architecture of a compiler.

Lexical and syntax analysis are likely the most recognizable tasks of a compiler, and are commonly referred together as simply *parsing*. These consists in transforming a stream of characters (e.g. from a file) into a sequence of tokens, and ultimately into a concrete syntax tree (CST) representing the syntactic structure of the input. While this goes past the scope of this thesis, a significant body

[1] Such compilers are often called *transpilers* in both literature and common parlance.

of work has being dedicated to the study of parsers (see [70, Chapter 3] for a comprehensive review).

One essential property of a compiler is that it must preserve the semantics of the program it translates. Therefore, it should create a semantic representation of its input, which it can use to rephrase the program in the target language. In other words, the objective is to attach a meaning to the program. This is typically done by building a more abstract representation of the CST, aptly named an abstract syntax tree (AST), before running one or more analysis passes, focusing on particular aspects of the semantics[2] (e.g. lexical scoping or type inference). Obviously, only programs that respect the source grammar can be turned into consistent ASTs. As a result, this can be interpreted as a first step into the verification of the input's correctness, as faulty programs (w.r.t to the grammar) are de facto rejected. However, for most programming languages, a *syntactically* correct program may still be *semantically* nonsensical. Hence, additional steps may be required to understand its intended meaning, which is usually done on one or several IRs of the program. Note that an AST *is* already an IR. However, trees are not always the most suitable data structures for all kinds of analysis, and therefore other forms of IRs are sometimes preferred (e.g. [47, 6]). Platform-agnostic code optimizations may also be also considered at this stage, such as constant propagation, dead code elimination or loop unrolling, to cite a few. We refer collectively to these additional steps as *semantic analysis*.

Parsing a program and building its semantic representations relate to the *front-end* of a compiler workflow. The *back-end* represents the final step, which consists of translating this representation into the target code. Just as parsing is highly dependent on the source language, code generation is highly dependent on the target language, and potentially the system on which the latter is expected to be executed as well. For instance, targeting assembly code for a particular architecture requires a comprehensive knowledge of available registers, execution units, calling conventions, etc. The same goes for target-specific optimizations, for which the execution model should also be considered. Fortunately, the use of standardized IR between front-end and back-end (e.g. LLVM [93] or Microsoft's CIL [106]) encourages code reuse and allows for a better code compartmentalization.

Rather than rephrasing the input program into some other target language, one may directly "execute" the semantic representation produced by a front-end. A program that operates this way is commonly named an *interpreter*. Note that the boundary between interpretation and actual program execution is blurry, as one could argue a CPU is in fact an interpreter for machine code. The general understanding of the definition is, however, that an interpreter is itself a program that runs on the top of some existing architecture. While some programming

[2]The reader will notice we took this approach to type check garbage collection in Section 6.3.

languages are historically designated as interpreted or compiled, both notions are not mutually exclusive. In fact, implementations of compilers and interpreters alike are now available for most mainstream programming languages (e.g. [92]).

7.1.2 Challenges of Compilation

Compilers occupy a very interesting place in software development, as they are completely unavoidable. They must extract a precise and unambiguous semantics from inputs designed to be as human friendly as possible. This poses several important challenges:

- Compilers need to be fast. They are on the front line of every program build, which means all executions are part of the development time. This constraint often disqualifies expensive analysis techniques, such as software model checking [43].

  They also need to scale for arbitrarily large inputs, as compilers may have to ingest millions of code. Linux has for example roughly 25 millions lines of C, and can be expected to be compiled in less than two hours on a modern personal computer.

- Compilers must be reliable. They are a piece of software that creates software, hence incorrect behaviors in the former are bound to introduce incorrect behaviors in the latter. Moreover, errors due to a faulty compiler might be hard to track down, as they are likely to be silently carried out in the translation [129]. As a result, a particular care must be brought into their correctness, often by the means of a formal approach [95], which also requires a formal description of their expected inputs and outputs. Testing is also challenging, due to the variety of execution paths available.

- Compilers are most often programs facing humans, and therefore should relay information in a way that is understandable to a human developer. While properties are formally expressed as mere facts that should be shown true or false, a simple `invalid_program` answer is of little to no help for a human. Instead, errors should be explained and put in context, which means a compiler not only has to detect *whether* a property does not hold, but also has to identify *why*.

7.2 Anzen in a Nutshell

This section introduces Anzen's syntax and semantics informally. A complete concrete syntax is given in Appendix B.

Decades of tradition suggest that the first example of a programming language be a program that prints “Hello, World!”. However, because Anzen aims at describing precise assignment semantics, we break with the establishment and start describing variables.

Anzen has a comprehensive notion of immutability, and supports both reassignability and memory immutability. Non-reassignable references are declared with the keyword `let`, and can be assigned only once, whereas reassignable references are declared with the keyword `var`, and can be freely reassigned throughout their lifetime. Memory immutability on the other hand is specified with type qualifiers. `@cst` describes an non-mutating reference and `@mut` describes a mutating one. Table 7.1 illustrates all possible combination of reassignability and memory mutability. Anzen is an opinionated language, and one of its opinionated choices is to prefer immutability over mutability. Accordingly, omitting the mutability qualifier is interpreted as a `@cst` by default, and parameters are always assumed non-reassignable. The three assignment operators of the $\mathcal{A}$-caluclus are carried along in Anzen, with the same semantics.

	Reassignable	Non-reassignable
Mutating	`var a: @mut <- 0`	`let b: @mut <- 0`
Non-mutating	`var c: @cst <- 0`	`let d: @cst <- 0`

Table 7.1: Reference and value mutability

Each reference in Anzen is associated with a semantic type. The type checker allows for most types to be inferred automatically. For instance in Table 7.1, all variables have the semantic type `Int`. Nonetheless, type annotations can be added, either to disambiguate between different solutions, or purely for the sake of legibility. Semantic type correctness is checked statically, so a declaration like the following will be rejected during compilation:

Anzen

```
let pkmn: Int <- "Pikachu"
// ^^^ Compile-time error: 'pkmn' has type Int
```

Reassignability is also checked statically, meaning that reassigning a parameter or reference declared with `let` will be rejected during compilation:

Anzen

```
let pkmn1: String <- "Pikachu"
let pkmn2: String <- "Bulbasaur"
pkmn2 &- pkmn1
// ^^^ Compile-time error: 'pkmn1' is not
    reassignable
```

Memory mutability is checked dynamically, meaning that modifying an immutable object will trigger an error during execution:

Anzen

```
let pkmn: String <- "Pikachu"
pkmn := "Bulbasaur"
// ^^^ Runtime error: 'pkmn' is not mutating
```

The runtime system also keeps track of ownership and uniqueness to prevent access to uninitialized or dangling references. Memory garbage collection is carried out automatically by the means of reference counting.

Anzen

```
var x <- 1
let y &- x
let z <- x
// ^^^ Runtime error: x is borrowed
```

Anzen supports custom types in the form of structures (a.k.a. composite types). Those must be declared and referred by name in variable declarations and function signatures. Structures' fields are references as well, so their declarations resembles that of variables, reusing the `let` and `var` keywords, as well as the `@cst` and `@mut` mutability qualifiers. Borrowing from object-oriented programming languages, methods, constructors and destructors can be declared along with a structure, and used to compartmentalize an object's behavior. Inside these functions, a reserved `self` reference allows access to the instance's internals.

Anzen

```
struct Pokemon {
  let name: String
  let level: @mut Int

  new(name: String, level: Int) {
    self.name <- x
    self.level <- y
  }
}

let pkmn = Pokemon(name <- "Pikachu", level <- 5)
```

`self` is always an non-reassignable reference, but whether or not it designates an immutable object depends on the methods declaration. By default, methods cannot mutate their internals via `self`, but their definition can be prefixed with `mutating` to turn `self` into a mutating reference. Value immutability is applied transitively. In other words, mutating a field of an immutable structure instance is an illegal operation, and provokes an error at runtime. Consequently, calling a mutating method on a reference that is not typed with `@mut` is forbidden.

Anzen

```
struct Pokemon {
  // ...
  mutating fun level_up() {
    self.level <- self.level + 1
  }
}

let pkmn = Pokemon(name <- "Pikachu", level <- 5)
pkmn.level_up()
// ^^^ Runtime error: 'pkmn' is not mutating
```

Functions and method signatures can be annotated to specify arguments and return values' expected ownership and mutability. The `@own` qualifier denotes ownership and is assumed by default, while the `@brw` qualifier denotes borrowed references. Just as for variables and fields, the `@cst` qualifier denotes non-mutating references and is assumed by default, whereas the `@mut` qualifier denotes mutating references. Note that methods cannot return non-mutating references on mutating fields. While doing otherwise would not impede the enforcement of memory immutability, the restriction is imposed for the sake of code legibility. Indeed, because a non-mutating reference can assume the memory to which it refers re-

mains immutable for its entire lifetime, any mutation of the aliased field would have to be forbidden. However, there is no way to "see" where the non-mutating borrowing took place or where it ends from within a method.

Anzen

```
struct Pokemon {
  // ...
  fun get_level() -> @brw @cst Int {
    return &- self.level
    // ^^^ Compile-time error: cannot form a non-
    // mutating alias on a mutating field
  }
}
```

The language supports function overloading (i.e. different function implementations with the same name), and uses function signatures to determine which function should actually be called at compile-time. Generic types can be used to further increase code reusability. Functions and structures accept generic placeholders that can be used to type parameters and fields. Upon instantiation, a generic type is specialized by replacing its placeholders with concrete types.

Anzen

```
fun swap<T>(a: @brw @mut T, b: @brw @mut T) {
  let tmp := a
  a := b
  b <- tmp
}

let x: @mut <- "Vaporeon"
let y: @mut <- "Jolteon"
swap(a &- x, b &- y) // <= swap is instantiated here
print(line &- x)
// Prints "Jolteon"
```

Anzen is a higher-order programming language, meaning that all functions are not only first-class citizen, but may also capture references by closure. Any identifier that is neither a parameter nor a variable declared within the function's body is captured by closure. Captured identifier are in fact borrowed references on the variables available in the function's declaration context. It follows that reassigning a variable captured by a function closure does not modify the borrowed reference inside the closure.

Anzen

```
var counter: @mut <- 0
fun next_number() -> Int {
  counter <- counter + 1
  return := counter
}

print(line &- next_number())
// Prints "1"

counter := 10
print(line &- next_number())
// Prints "11"

let counter2: @mut <- 100
counter &- counter2
print(line &- next_number())
// Prints "12"
```

A function is not allowed to create aliases on its arguments that live longer than its call. This mechanism is used to prevent methods accepting higher-order functions from accidentally leaking a non-mutating reference on a field.

Anzen

```
struct Pokemon {
  // ...
  fun do_with_level(fn: (x: @brw Int) -> Nothing) {
    fn(x &- self.level)
    // ^^^ The loan of 'self.level' is guaranteed not
    // to persist after fn returns
  }
}

let pkmn = Pokemon(name <- "Pikachu", level <- 5)
var i <- 0
pkmn.do_with_level(fn: fun (x: @brw Int) {
  i &- x
  // ^^^ Runtime error: 'x' cannot escape
})
```

7.3 Examples

We now return to the security breach of JDK 1.1 that we presented in Section 1.1.1 and present two alternative implementations in Anzen to further showcase its features. In the original Java implementation, the flaw was caused by the exposition of an object's internal representation through one of its methods, allowing attackers to modify the said internal representation. We remind the incriminated code:

Java

```java
public class Class {
  public Identity[] getSigners() {
    return this.signers;
  }
  private Identity[] signers;
}
```

Calling the method `getSigners` creates a mutable alias on the `signers` field, allowing one to manipulate the object's internal representation. There are two ways to prevent this situation in Anzen[3]. The first is to return a deep copy of `signers`'s value by the means of the mutation operator, therefore removing any potential alias to the internal representation.

Anzen

```
struct Class {
  fun get_signers() -> @mut List<Identity> {
    return := self.signers
  }
  let signers: @mut List<Identity>
}

let class <- Class()

// 's' is a copy of 'class.signers'
let s: @mut <- class.get_signers()
print(line <- signers.count())

// Mutating 's' does not affect 'class.signers'
s.append(element <- Identity(name <- "Judas"))
```

The drawback of this first approach is that it involves a deep copy, potentially

[3] Admittedly, Anzen does not currently support access modifiers (e.g. `private`), therefore the language is unable to forbid a direct access to the field. We only describe protection mechanism on the method's return value.

expensive in terms of time and/or space. Although optimization techniques such as copy-on-write [136] can alleviate this issue, a better angle would be to use a non-mutating alias. However, as returning non-mutating aliases on mutating fields is prohibited, one cannot write simply return `signers` by alias. Instead, the solution is to rely on Anzen's support for higher-order functions to inverse control, and accept a function that manipulates an immutable reference to the field. That way, the non-mutating alias does no longer escape the method's scope. As aliases to mutating fields cannot be passed to escaping parameters, mutatibilty is guaranteed to be restored after the higher-order function returns.

Anzen

```
struct Class {
  fun do_with_signers(
    fn: (signers: @brw List<Identity>))
  {
    fn(signers &- self.signers)
    // ^^^ temporarily lend 'self.signers' immutable
  }
  let signers: @mut List<Identity>
}
```

Here, the drawback is that the higher-order function cannot return a value. One could capture an identifier by closure and mutate it, thereby using it as a sort of inout parameter, but such approach is unnatural. A more elegant solution is to leverage generic types, so that `do_with_signers` can accept a higher-order function with any codomain, and simply forward its return value. As the higher-order function's codomain is annotated with `@own` (i.e. the default value), it is safe to use the move operator to forward results, without any copy overhead.

Anzen

```
struct Class {
  fun do_with_signers<R>(
    fn: (signers: @brw List<Identity>) -> @own R)
    -> R
  {
    return <- fn(signers &- self.signers)
  }
  let signers: @mut List<Identity>
}

let class <- Class()
let count <- class.do_with_signers(
  fn <- fun (signers: List<Identity>) {
    return <- signers.count()
  })
print(line <- count)
```

7.4 Anzen's Intermediate Representation

While ASTs are the most widespread form of IR, virtually present in all compilers, modern toolchains often use other IRs. In particular, a growing interest is directed toward intermediate languages. An intermediate language is usually a lower-level programming language which is more suitable to perform static analysis and code optimizations. In addition, they can be leveraged to adopt a divide and conquer approach to the translation of high-level constructs. For instance, it is simpler to first translate iterators into simple loops, rather than going from iterators to machine code directly. Consequently, many modern compilers rely on an intermediate language (e.g. Swift, Rust or Julia to cite a few). Anzen's compiler is no exception, and translates Anzen into AIR.

AIR is a low-level instruction set similar to assembly code, but that preserves Anzen's assignment operators, together with a handful of high-level concepts such as structures and higher-order functions. Although the language is not supposed to be used for writing programs directly, it has a concrete syntax, described in Appendix B, that can aid debugging Anzen's compiler, and that we will use to illustrate various examples.

The most significant difference between AIR and Anzen is that an expression cannot contain other sub-expressions. Instead, it should be decomposed into simpler instructions that store intermediate results into temporary registers. Like in LLVM, registers are SSA variables [47] (i.e. variables that are assigned exactly

once) and represent memory locations that store references. In the words of the C language, a register is a pointer to a pointer. The advantage is that unlike a reference identifier in Anzen, a register in AIR uniquely designates one and only one reference. Because of this design, code in AIR explicitly exposes how references are manipulated.

Example 7.4.1: Simple AIR

Consider the following Anzen program:

Anzen

```
1 fun id(x: Int) -> Int { return <- x }
2 id(x <- 42)
```

The function call at line 2 is transpiled to the following AIR code:

AIR

```
1 %1 = make_ref (Int) -> Int
2 bind @id_Fxi2i, %1
3 %2 = make_ref Int
4 move 42, %2
5 %3 = apply %1, %2
```

At line 1, the instruction allocates a new reference for a function value, and stores its address into a register `%1`*. Line 2 assigns by alias the function* `id` *to the reference stored at* `%1`*. Line 3 creates a second reference. Line 4 moves the value* 42 *to the reference stored at* `%2`*. Finally line 5 applies the function stored at* `%1` *with the argument stored at* `%2`*.*

We can draw a parallel with the $\mathcal{A}$-calclus' semantics. Recall that evaluation is described by inference rules concluding statements of the form $C \vdash t \Downarrow l, C'$, where t is a term and l the memory location to which it evaluates. In AIR, most instructions are of the form `%x = inst`, where `inst` is an instruction and `%x` the memory location of a reference that points to its result.

7.4.1 Memory Management

As mentioned above, AIR uses SSA registers to represent references. Those are created with the instruction `make_ref`, which returns the address of the newly allocated reference. Once allocated, the reference itself is a null pointer that has yet to be assigned to an initialized memory location, meaning that dereferencing it will obviously provoke an uninitialized memory error.

```
1  %1 = make_ref Int
2  // ...
```

Stack

%1	0x0

Heap

Like in the $\mathcal{A}$-calculus, memory for atomic literals is allocated when said literals are evaluated. Note that Anzen, and by extension AIR, does not treat atomic values (e.g. numbers and booleans) differently than other objects. Therefore they are also allocated on the heap. Function literals however are treated a little differently. First-order functions are represented as global objects, and live as long as the program. Consequently, moving them is in fact forbidden, and silently interpreted as an alias creation. Higher-order functions on the other hand are actual objects, which are allocated by the `partial_apply` instruction. This mechanism is described later.

Instances of structures are allocated with the `alloc` instruction, which allocates a sequence of pointers that will be used to represent references to each member. This instruction only appears in constructors, and correspond to the allocation of the `self` reference.

```
1  %1 = alloc Pair
2  // ...
```

Stack

%1	0x1

Heap

0x1	0x0
0x2	0x0

Fields dereferencing is carried out with the `extract` instruction. It takes as argument a reference to a structure instance and the offset of the field.

```
1  %1 = alloc Pair
2  %2 = extract %1, 1
3  move 42, %2
```

Stack

%1	0x1

Heap

0x1	0x0
0x2	0x3
0x3	42

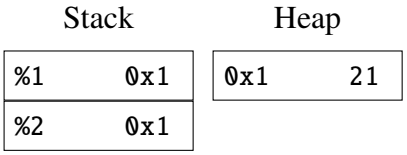
Registers' lifetimes are bound to call frame in which they are defined. Consequently, a reference is automatically dropped every time a function returns. Heap memory is managed using reference counting [142].

7.4.2 Assignments

AIR features one instruction for each assignment operator in Anzen. Note that their operands appear in reverse order, so that the right operand is indicated before the left one. For instance, `move 42, %1` reads as "move the value `42` into the

reference `%1`". Aliasing assignments are represented by `bind`. Mutation assignments are represented by `copy`.

```
1  // ...
2  move 42, %1
3  bind %1, %2
4  copy 21, %1
```



7.4.3 Routines

Every instruction in AIR is executed in a routine. In fact, code written outside of any function in Anzen is implicitly declared in the "main" routine in AIR, except for type and function declarations, which are defined separately. AIR routines are not first-class values, although they can be referred by their address, like in C. The following listing shows the AIR of a simple "Hello, World!" program (i.e. `print(line <- "Hello, World!")` in Anzen):

AIR

```
fun @main : () -> Nothing {
entry#0:
  %1 = make_ref (Anything) -> Nothing
  bind @__builtin_print, %1
  %2 = make_ref String
  copy "Hello, World!", %2
  %3 = apply %1, %2
  jump exit#0
exit#0:
  ret
}
```

In memory, a routine is represented as a type signature and a body, which consists of a sequence of instruction blocks, labeled by a name. A well-formed routine necessarily contains at least two blocks, labeled `entry#0` and `exit#0`, that denote the entry and the exit points of the routine, respectively. Naturally, `entry#0` is always the first block and `exit#0` the last one. By default, all instructions of the block are executed in order until the end of the block, and the `jump` instruction allows to jump from one block to another. An alternative `branch` instruction allows for conditional jumps. Except for the `exit#0` block, which should always contain a single `ret` instruction, the last instruction of each block must be a jump, otherwise the runtime behavior is undefined.

Functions with empty closures are called *thin*, and can be simply represented as routine pointers. As seen in the following example above, applying a function consists in performing all parameter assignments, before the corresponding references are passed to the routine, using the `apply` instruction. This triggers the runtime to push a new stack frame and jump into the called routine's `entry#0` block. Parameters are received into local registers, starting from `%1`.

Functions with closures are called *thick* and need to be instantiated with the `partial_apply` instruction. It accepts a reference to a routine followed by a sequence of references for the closure, and returns a reference to a structure instance representing a higher-order function[4], that can be fed to `apply`.

Example 7.4.2: Simple AIR

Consider the following Anzen program:

Anzen

```
let counter: @mut <- 0
fun next_number() -> Int {
  counter <- counter + 1
  return := counter
}
let x <- next_number()
```

The above program is transpiled to the following AIR code:

[4]The reader will notice this process essentially implements defunctionalization [120].

AIR

```
fun @next_number_F2i : (Int) -> Int {
  // ...
}

fun @main() -> Nothing {
entry#0:
  %1 = make_ref Int
  move 0, %1
  %2 = make_ref Int
  %3 = make_ref () -> Int
  %4 = partial_apply @next_number_F2i, %1
  bind %4, %3
  %5 = apply %3,
  move %5, %2
exit#0:
  ret
}
```

Recall that Anzen supports overloading and generic types. In AIR however, each version of an overloaded function is given a unique name, obtained by appending its signature. This technique is usually referred to as *name mangling* in other compilers. Generic functions require more attention. Two techniques are used. The first consists in generating a *monomorphized* version of a generic function every time a particular specialization is needed, similar to how code for C++ templates is generated. In other words, each reference to a generic function instructs the compiler to *clone* its body, substituting generic types for specialized ones. One drawback of this approach is that it may produce very large AIR units in the presence of highly generic code. Furthermore and more importantly, monomorphization requires that the body of a called function be known during compilation, thus impeding the separate processing of each unit. To solve these issues, a second technique is to replace genericity with polymorphism, relying on boxing and virtual tables [57, Chapter 3] for functions and method dispatching. Implementation details are described in [118]. Although monomorphization is preferred for performance reasons [56], it can only occur under two conditions. First, the generic function must be declared in the same compilation unit, so that the compiler can have access to its body during AIR generation. Second, the monomorphization of a function must not lead to another monomorphization of itself, with a different type. The second constraint guarantees that there is a finite amount of code duplication.

7.5 Anzen's Compiler

The Anzen's compiler is written in the Swift programming language, and distributed as an executable for macOS and Linux. The compiler's architecture does not deviate from the usual design schematized in Figure 7.1.1. The source code is split across separate libraries with minimum interdependencies:

AST This library is the largest and most essential component of the compiler. It contains all classes and interfaces related to the AST (i.e. the `AST.Node` class) and annotations thereof.

Parser This library implements Anzen's parsing. The most important part of its public API are the `Lexer` and `Parser` classes. The former breaks a sequence of characters into individual tokens, and the latter turns a sequence of tokens into an AST.

Sema This library implements the semantic analysis, which mostly relates to (semantic) type inference. It provides several classes representing distinct "passes" of the semantic analysis, in the form of AST visitors [65]. Further details are discussed below.

AnzenIR This library contains the code related to the translation of a fully annotated AST into AIR. It exposes a class `AIRBuilder` to ease the creation of a well-formed AIR program.

Interpreter This library simply offers a class `Interpreter`, that can execute an AIR program.

We will not delve into the details of the parsing task, as it is relatively straightforward. The implementation consists of a handwritten recursive descent parser, that directly produces an AST, but keeps information about the concrete syntax in the form of annotations. This allows to identify what part of the input source corresponds to a given node in the AST, which is paramount to produce meaningful error messages. Figure 7.5.2 depicts the Anzen AST for a program that prints "Hello, World!" on the console.

7.5.1 Semantic Analysis

The main objective of the semantic analysis is to perform the type inference. The module takes an untyped AST as an input, and either outputs a fully typed AST or produces a set of compilation errors. Semantic analysis is decomposed into three stages. Each stage is implemented as an AST visitor [65], attaching information to the AST by the means of node annotations.

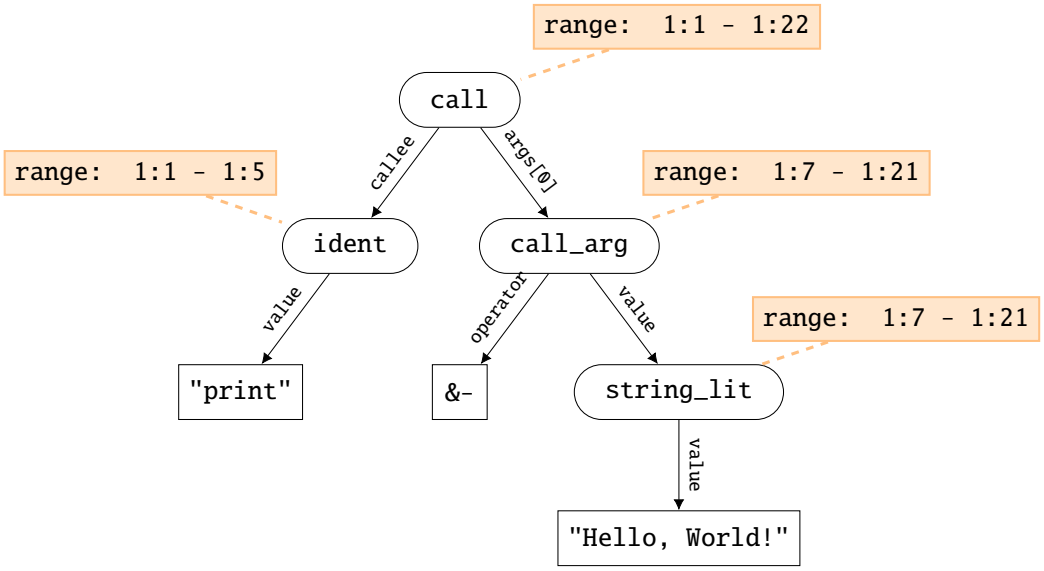


Figure 7.5.2: AST of the program `print("Hello, World!")`. Rounded rectangles represent AST nodes, while squares represent leaves. Annotations drawn in orange contain the position of each node in the source.

Symbol Creation In this stage, nodes that delimit a lexical scope (e.g. function declarations) are associated with a scope object. Additionally, a unique symbol is associated with each identifier declaration, which will be used in the subsequent stages to resolve name shadowing and function overloading.

Name Binding In this stage, reference and type identifiers are attached to the scope that declares them, effectively linking identifiers to their declaration.

Type Inference In this stage, the AST is scanned to build a constraint system, which is solved by deduction. The resulting solution is used to annotate each node with their type, concluding the semantic analysis. This stage is the most complex, in particular with respect to error reporting.

Each stage can produce a different set of errors, as depicted in Figure 7.5.3. The compiler does not stop at the first error encountered, but attempts to progress further into the semantic analysis instead, so as to provide more thorough error reports. For instance, the detection of an undefined identifier does not stop the compiler from trying to complete the second stage, so that other undefined identifiers might be found and reported at the same time. As each node of the AST is annotated with its corresponding range in the source, the compiler can accurately report at which line and column errors from the first two stages occur. Typing errors are however more difficult to report.

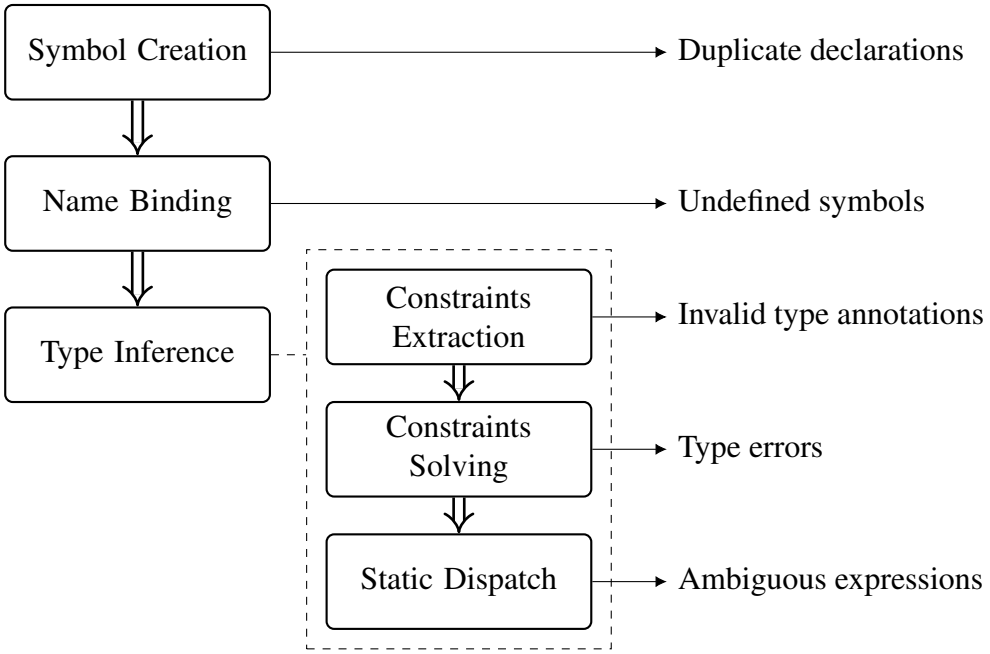


Figure 7.5.3: Possible errors encountered at each stage of the semantic analysis.

7.5.2 Type Inference

Type inference, the third stage of semantic analysis, poses a number of issues. In particular, overloading implies that there cannot be a simple one-to-one mapping between an identifier and its type. The use of unique symbols partially solves the issue, but generic types add yet another layer of complexity, as these must be specialized in context.

Example 7.5.1: Inference with overloading and specialization

Consider the following program, overloading a reference `f` *with three definitions, one of which being generic.*

Anzen

```
fun f(x: Int, y: Int) -> Bool { /* ... */ }
fun f(x: Int, y: String) -> Int { /* ... */ }
fun f<T>(x: T, y: String) -> T { /* ... */ }

let a <- f(x <- 2, y <- 2)
let b <- f(x <- "R2D2", y <- "C3P0")
let c <- f(x <- 0, y <- "BB8")
```

Type inference has to distinguish between these cases based on the type of the arguments that are passed for the parameters `x` *and* `y`*. Line 5 only matches the first definition, and is therefore dispatched accordingly. Line 6 only matches the third definition but requires a specialization. Line 7 is ambiguous because two different definitions match* `f` *in that context.*

Anzen compiler's overcomes this issue by solving a constraint system. Every expression is first associated with a unique type variable, before the AST is scanned to extract constraints between these type variables. Five kinds of type constraints can be added to the system:

- **Equality constraints** are added to indicate that two type variables must be equal. This corresponds to unification.
- **Conformance constraints** denote a relaxed notion of equality, that prescribes that one type's interface might be compatible with another[5].
- **Construction constraints** require that one type be the signature of a constructor for the other.
- Let T and U be two types, **member constraints** require that T be a composite type, with a member whose type is equal to U.
- **Disjunction constraints** represent a choice between different ways to type an expression.

Once built, the constraint system can be solved by breaking constraints into groups of smaller constraints, until only equality constraints remain to be solved. Those correspond in fact to unification. The process is bootstrapped by fixing some of the variables, when the type of the corresponding expression can be trivially inferred (e.g. integer literals have type `Int`).

Example 7.5.2: Type constraint system

Consider the following declaration:

Anzen

```
1 let x <- 9 + 3
```

Each expression is first associated with a type variable. For instance:

$$\texttt{x} \mapsto \alpha, \texttt{9} \mapsto \beta, \texttt{3} \mapsto \gamma$$

[5] As of this writing, such a constraint can only be solved if the two types are equal, or if one is `Anything`, a type that can represent an instance of any type.

Then, the AST is scanned to extract typing constraints between these variables, based on the context in which corresponding expressions are used. In this particular example, this will result in the following constraints:

$$\alpha = \delta \quad \wedge \quad \beta.+ = \gamma \to \delta \quad \wedge \quad \beta = \texttt{Int} \quad \wedge \quad \gamma = \texttt{Int}$$

As the system is solved, the second constraint (i.e. $\beta.+ = \gamma \to \delta$) will be broken into a constraint that unifies the type of the built-in method `Int.+` *(i.e.* `Int` $\to$ `Int`[a]*), so that α, the type of* `x`*, is eventually unified with* `Int`*.*

[a] In Anzen, infix operators are implemented as methods of the left operand.

While this approach is simple, its main drawback relates to error reporting. If one of the broken pieces of a larger constraint is unsatisfiable, it is difficult to trace an error back to the point in the source code from which the original constraint was extracted. Consider for instance a constraint aiming to unify two function types `Int` $\to$ `Int` = `Bool` $\to$ `Int`. The constraint will eventually be broken into two equality constraints, one of which attempting to unify the codomains. While this constraint is certainly unsatisfiable, an inability to trace back the error to the original larger constraint will result in a poor error reporting. The solution we adopt is to attach to each constraint a log keeping a record of the reasons that brought it into existence. This information can then be used to create accurate error reports.

If all type constraints can be solved, the last step is to replace the type variables associated with each expression with actual concrete types. This process is straightforward if there is only a single solution to the constraint system. Otherwise, the compiler attempts to select the solution that requires the less generic specializations. In plain English, this implies that non-generic types are preferred. In Example 7.5.1 for instance, the type system will choose to use the second function definition rather than the third one. While both could be applied, using the former requires one less generic specialization. Unfortunately this criterion is not sufficient to eliminate all ambiguous situations. In these cases, the type system gives up and reports the ambiguity.

7.5.3 AIR Generation

If semantic analysis succeeds, the final step is to transform a fully annotated AST into an AIR program. This is achieved by the means of an AST visitor, called the *AIR emitter*, that goes down the tree and generates the AIR code corresponding to each node visited. Althought AIR generation comes with a fair share of implementation challenges, those are quite unremarkable and are therefore not detailed

in this document. Roughly speaking, the translation consists in the allocation of a SSA register for each expression, which are then appropriately passed as operands to AIR instructions.

7.5.4 Limitations

Though in its current state, the Anzen language and its compiler can be used to write simple programs, both suffers certain limitations in terms of performance and expressiveness, that leave Anzen unsuitable to write actual complex applications. Three of them are discussed below.

Performance

There are currently no optimization performed at any stage of the compilation. Besides, the language can only be interpreted from its intermediate representation, further increasing the execution overhead. As a result, Anzen programs' executions are extremely slow, even compared to other interpreted languages, with experiences showing Anzen running up to 90% slower than Python. This is obviously unacceptable for any actual application.

There are countless language-agnostic code optimization techniques that could improve these performances. Moreover, generating actual machine code from AIR would evidently yield a significant performance increase as well. One potential lead toward both directions is to translate AIR into LLVM IR, in order to benefit from a large collection of proven optimization techniques.

Besides the absence of even simple code optimizations, one of the causes for Anzen's catastrophic performances is that AIR statements induce a lot of memory traffic. As each instruction is represented as a reference, assigning and dereferencing variables involves many expensive pointer indirections. Representing all values as heap allocated objects allows for a simple implementation of the assignment operators, that does not have to take operands type into account. However, as Anzen is statically typed and since the expected semantics of every reference assignment is known, the compiler could promote heap allocations to stack allocations when the aliasing semantics is either unneeded or contained locally.

Debugging and Error Reporting

Troubleshooting Anzen programs is currently very difficult. For instance, type errors are reported as a set of unsolvable constraints, with the locations in the source from which they originate, which can certainly appear cryptic to the untrained eye.

Debugging also requires improvements. The current approach consists in inserting breakpoints into the interpreter's source to halt an AIR program's evalu-

ation. The interpreter's structure is simple enough that the state of its memory registers can be inspected easily with a Swift debugger (e.g. Xcode [10]). Figure 7.5.4 illustrates this approach. However, controlling execution is difficult, and requires some knowledge about the interpreter's implementation. Furthermore, although the location in Anzen's source from which a particular AIR instruction originates can be retrieved from its metadata, consulting it is inconvenient at best.

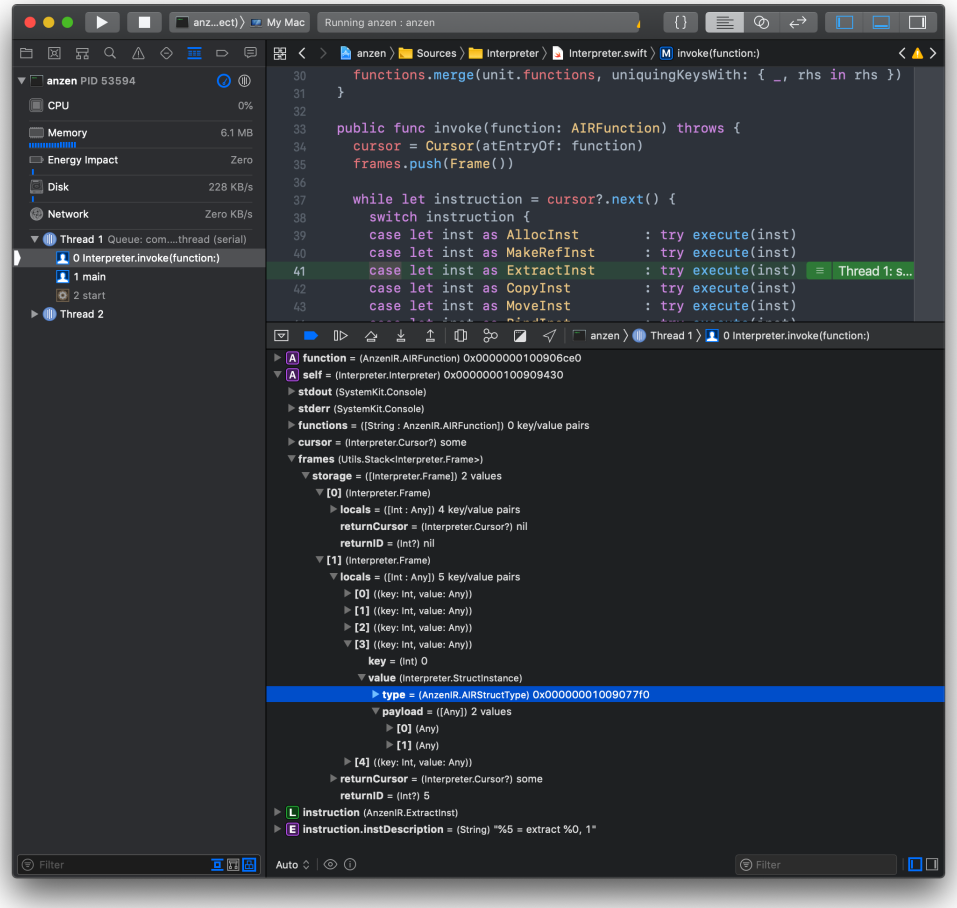


Figure 7.5.4: Inspection of the AIR interpreter's state in Xcode.

Lack of Module Support

Anzen has no support for module imports, meaning that all code should be given in the form of a single source file. This significantly hinders code reusability, in particular in a collaborative environment.

Supporting module imports is also paramount to provide a standard library. A standard library is a collection of types and functions that are commonly used in all sorts of programs. For instance, most modern programming languages offer built-in types to represent lists, sets and tables. Apart from a few built-in types, Anzen does not provide any of these tools, hence requiring all programs to redefine all common data structures.

Several steps have already been done toward the preparation of a module importing system. ASTs can be annotated with type definitions defined externally. In fact built-in types are currently "imported" from a built-in module definition in the current implementation.

7.6 Summary

The chapter introduces Anzen, a general purpose programming language based on the $\mathcal{A}$-calculus' semantics. Anzen combines the three assignment operators we have studied in the previous chapters with high-level programming concepts. Anzen's compiler transpiles programs into an intermediate language called AIR. AIR is an assembly-like language reminiscent to LLVM IR, that uses SSA variables to represent references. This IR's operational semantics almost shares a one-to-one correspondance with that of the $\mathcal{A}$-calculus, as temporary registers (i.e. SSA variables) actually corresponds to the memory location computed by the formal semantics.

We have presented Anzen and its features, described its intermediate representation and discussed its compiler's implementation.

Chapter 8

Conclusion

This final chapter concludes with a summary of the work that has been presented in this thesis, and outlines future research and development on the $\mathcal{A}$-calculus and Anzen.

8.1 Summary of our Contributions

Although decades of work in research and academia have yielded hundreds of impressive tools to model, test and analyze code, programming languages are still the first weapon in a developer's arsenal to write correct code. Good abstractions, clear syntaxes and intuitive semantics are at least as important as good tooling for software development. Thanks to the continuing growth in computing power, modern compilers are now able to process extremely sophisticated languages, that constantly raises the abstraction level closer to the human and farther from the machine. However, while these advances have undeniably contributed to make code simpler to write and clearer to read, relics of the underlying model still transpire in most languages' semantics, revealing cracks on the surface of the abstractions. This problem is particularly pervasive in imperative systems, where the relation between variables and values stored in memory is often confusing. Although it is tempting to see them as interchangeable notions, values are semantic objects that live in memory, whereas the variables are syntactic tools to describe operations on them. Furthermore, there is not necessarily a one-to-one relationship between both. Consequently, foreseeing the impact of a modification described over a variable can prove difficult. It is therefore of paramount importance that programming languages be defined with a syntax and semantics that leave no doubt about the effect of an instruction. Unfortunately, most mainstream programming languages hardly satisfy this need, because their assignment semantics is set at a lower abstraction level than the other concepts they have to offer.

This thesis revisits memory assignment semantics in the context of imperative programming languages, with the objective to provide a universal framework to reason about imperative systems, that can be used to model existing programming languages and design new ones. Its contributions are summarized below:

A computational model for imperative languages Our main contribution is the $\mathcal{A}$-calculus (pronounced *assignment calculus*), a formal model that aims at unifying imperative assignment semantics. Unlike other calculi formalizing assignment, the $\mathcal{A}$-calculus does not attempt to abstract memory away, but rather puts a more precise definition of the relationship between variables and memory. This is achieved by the means of three primitives representing the assignment semantics commonly found in imperative languages. One describes aliasing, another describes cloning and the last relates to assignments that preserve uniqueness.

A formalization of memory errors Using the the $\mathcal{A}$-calculus' operational semantics, common memory safety problems are formally described, namely uninitialized memory errors, use after free errors, doubly freed pointers and memory leaks. Two approaches to detect them are proposed. One is based on a post-mortem examination of failed executions, whereas the other offers to instrument the semantics to model errors explicitly.

A static type system for memory safety This research also explores aliasing control mechanisms for memory and data safety in the context of the $\mathcal{A}$-calculus. We first propose a type system to support static garbage collection, based on ownership and uniqueness to identify safe deallocation points. A second type system is suggested to bring immutability, while relaxing the usual single writer constraints enforced in related approaches.

A programming language Finally, this thesis introduces Anzen, a general purpose programming languages that aims at validating the practically of the $\mathcal{A}$-calculus as an actual programming language. Anzen combines the $\mathcal{A}$-calculus' semantics with high-level programming concepts, such as higher-order functions and generic types, and enforces aliasing control mechanisms at runtime. A compiler and interpreter are presented. The former translates Anzen sources into an intermediate representation that is as close as possible to the $\mathcal{A}$-calculus, while the latter executes it.

8.2 Critique

The $\mathcal{A}$-calculus aims at being a minimal yet universal framework to formally define assignment semantics. Consequently, much of effort has been put to remove

all non essential features. For instance, the language originally supported higher-order functions. However, their formalization was bringing much complexity to handle closures, which were overlapping with composite types. While their removal ultimately proved beneficial in reducing the language's size, the model remains arguably complex, in particular because of its treatment of first-class functions. With some hindsight, opting for a computational model derived from the λ-calculus to describe imperative languages is a debatable choice. Most of the λ-calculus' elegance relies on the simplicity of β-conversion, from which our model has to completely walk away. Nonetheless, the renewed interest into functional programming in mainstream languages convinces us that first-class functions are worth the complexity.

Memory and data safety is very difficult to achieve in the $\mathcal{A}$-calculus, because the model does not naturally distinguish between aliases and references. Other statically typed languages (e.g. Rust, C++, Swift) usually fix this as a flow-insensitive information. Our type system attempts to distance itself from this approach as much as possible, but at the cost of much annotation overhead. Forcing constraints onto an a priori more permissive system is a questionable approach, because reasons for such constraints have to be explained and understood.

Although this thesis describes a static approach to achieve memory safety, our implementation of Anzen performs type checking at runtime, which therefore deviates from the formal definitions that were presented in the present document. While both type systems share a lot of similarities, all mechanisms aimed to statically track type capabilities beyond function boundaries and conditional terms are eluded in our implementation.

Finally, Anzen's compiler is a quite complex machine. In particular, bringing generic types and inference in Anzen has proven to be a formidable challenge. In retrospect, settling for either a simpler language or a less powerful type inference engine would have taken far less time, and given more opportunity to conduct practical evaluations.

8.3 Future Directions

The research presented in this thesis suggests a number of appealing future directions. We quickly discuss some of them below.

Concurrency The $\mathcal{A}$-calculus only models single-threaded programs, whereas real programming languages usually offer primitives for concurrency.

Concurrency has been a topic of interest for many years and have been extensively researched. A number of language calculi have been introduced

for stuying various models of concurrency, including highly influencial proposals such as CSP [77] and the π-calculus [122]. More recent efforts have been directed towards a unification of these models [1]. Other approaches have proposed to extend existing object calculi, such as Featherweight Java, with support for concurrency (e.g. [113, 36]). A natural future direction of our research is to determine which of these approaches would be best suited for extending the $\mathcal{A}$-calculus with concurrency.

Refine the Type System The type system we have presented for the $\mathcal{A}$-calculus is opinionated, whereas the language aims to unify imperative assignment semantics. In fact, most type constraints we proposed relate to the constraints of static garbage collection, Consequently, one of the future directions we would like to take is to continue studying how aliasing and memory control mechanisms can be applied to the $\mathcal{A}$-calculus. Of particular interest is to identify a relationship between language constructs and safety properties. For instance, we showed in Section 6.4 that constructors and destructors were needed to support static garbage collection.

Explicit memory management invites a number of challenges, because it expands the number of possibly unsafe situations. As it stands our type system guarantees that a reference cannot be simply allocated but uninitialized, because allocation and initialization are viewed as a single operation. This property implies that owning references are necessarily safe to dereference. However, it also entails definite initialization, which fails to model how certain languages operate (e.g. C), and is one of the main obstacles to the support of records, as discussed in Section 6.4. Definite initialization could be represented as an additional flow-sensitive capability instead, required to type check dereferencing. A similar approach is taken in the Cyclone language [84].

Dynamic garbage collection is a feature present in most contemporary programming languages, including Anzen. While assuming deallocations are properly handled at runtime would certainly simplify our type system, as one could elude lifetime annotations and most of the restrictions on move operations, other statically verifiable properties could be valuable. For example, Region Types [26] could be used to detect and/or resolve circular references, and prevent strong reference cycles.

Lastly, the type system could be enhanced with additional features in numerous ways. One obvious extension candidate is immutability, which we suggested in Section 6.5. Other directions include the vast body of research dedicated to aliasing control mechanisms [39].

Model Checking One objective of the $\mathcal{A}$-calculus is to model existing programming languages, by translating programs into a simpler and formally defined form. From there, an interesting enterprise is to leverage model checking techniques to verify properties. Ideally, the translation from a high-level language to the $\mathcal{A}$-calculus should be automated, and fed to a model checker without manual intervention.

Although state space explosion is certainly a limiting factor to such an approach, program annotations and static analysis could be used to create predicate abstractions [11] and reduce the search space.

Anzen We already mentioned few of the limitations of the Anzen's compiler in Section 7.5. One relates to the extremely poor performances of its interpreter. An obvious future work is the implementation of standard code optimizations, such as constant folding and propagation, common subexpression elimination or data store elimination [70, Chapter 9]. Discussions with other compiler implementors suggest translating Anzen or its IR into another actual programming language, in order to rely on the latter's compiler for optimizations. A promising lead in that direction is to generate LLVM IR [93] code. LLVM IR is another SSA language, that also provides virtual registers to handle mutable states, and features powerful optimization techniques (e.g. iterated dominance frontier) to remove unnecessary memory traffic.

Anzen lacks application reports to assess its practicality to write actual programs. A standard library and/or a support for modules are probably required to conduct such an evaluation.

8.4 The Holy War of Programming Languages

No software developer is stranger to the Holy War of Programming Languages. Everyone has their favorite, and even the most mature of us cannot, at least occasionally, look down on others' choices. I am no exception. I started this thesis convinced Python was the best language in the world. I grew tired of runtime errors and turned myself to C++. I grew tired of memory errors and turned myself to Swift. I grew tired of confusing assignment semantics and decided to write my own. A few years later I succeeded, but cannot help to lament at the many, many areas where Anzen falls short compared to others. This journey convinced me that, in some respect, all languages are bad. They have unsound type systems, leaky abstractions, horrendous syntax or even worse semantics. Therefore there is no best language, there are only ones that are "less bad".

In spite of this grim and cynical view, I cannot deny that languages are getting better. More code is written every day than the one before, for applications that are getting more and more complex. Evidently, less bad is good enough. This means that the quest for the holy programming language, savior of us all, is probably futile after all. With some hindsight, this mimics the state affairs in nearly all scientific domains, in the sense that theories of everything are rarer than a unicorn. Instead, one should pursue with love and passion the discovery of small increments, which slowly but surely will help make sense of seemingly intractable predicaments.

I certainly love my assignment operators, and by writing this thesis I hope I have convinced the reader they constitue one of these small increment, making Anzen less bad than other languages, in default of making it good.

Bibliography

[1] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: a calculus for parallel computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 18–32, 2016.

[2] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 78–91, 2005.

[3] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. Plaid: a permission-based programming language. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 183–184, 2011.

[4] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 1–25, 2004.

[5] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.*, pages 311–330, 2002.

[6] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[7] Nada Amin and Ross Tate. Java and scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 838–848, 2016.

[8] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[9] Apple. Swift compiler. `https://github.com/apple/swift`, 2019.

[10] Apple. Xcode. `https://developer.apple.com/support/xcode/`, 2019.

[11] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213, 2001.

[12] Jiri Barnat, Lubos Brim, and Petr Rockai. Towards LTL model checking of unmodified thread-based C & C++ programs. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, pages 252–266, 2012.

[13] Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. Annotations for (more) precise points-to analysis. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming, 2007, Berlin, Germany*, January 2007.

[14] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality. *CoRR*, abs/1901.10220, 2019.

[15] Jan A. Bergstra, T. B. Dinesh, John Field, and Jan Heering. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):639–684, 1997.

[16] Yves Bertot. A short presentation of coq. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 12–16, 2008.

[17] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Logical Methods in Computer Science*, 14(3), 2018.

[18] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 195–219, 2009.

[19] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 257–281, 2014.

[20] Gavin M Bierman, MJ Parkinson, and AM Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[21] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 25–37, 1998.

[22] Bruno Blanchet. Escape analysis for java$^{\text{tm}}$: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[23] Aniruddha Bohra and Eran Gabber. Are mallocs free of fragmentation? In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 105–117, 2001.

[24] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, Boston, USA, 2001.

[25] Chandrasekhar Boyapati and Martin C. Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, pages 56–69, 2001.

[26] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 324–337, 2003.

[27] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[28] John Boyland. Checking interference with fractional permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 55–72, 2003.

[29] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pages 2–27, 2001.

[30] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, 2000.

[31] Nicholas Robert Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 441–460, 2007.

[32] Nicholas Robert Cameron, James Noble, and Tobias Wrigstad. Tribal ownership. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 618–633, 2010.

[33] G. Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., 2013.

[34] Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *Electronic Notes in Theoretical Computer Science*, 322:87–102, 2016.

[35] Elias Castegren. *Capability-Based Type Systems for Concurrency Control*. PhD thesis, Uppsala University, Sweden, 2018.

[36] Elias Castegren and Tobias Wrigstad. Oolong: an extensible concurrent object calculus. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1022–1029, 2018.

[37] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 347–364, 1994.

[38] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*, pages 378–387, 2005.

[39] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[40] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pages 176–200, 2003.

[41] David G. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[42] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 48–64, 1998.

[43] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 305–349, 1996.

[44] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, 2011.

[45] Christopher Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.

[46] Erik Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 233–244, 1991.

[47] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[48] Niels H. M. Aan de Brugh, Viet Yen Nguyen, and Theo C. Ruys. Moonwalker: Verification of .net programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 170–173, 2009.

[49] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6:1–6:31, 2009.

[50] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 465–490, 2004.

[51] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Transactions on Programming Languages and Systems*, 33(6):20:1–20:62, 2011.

[52] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[53] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy bounds with accurate checking. In *23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Dallas, TX, USA, November 27-30, 2012*, pages 195–200, 2012.

[54] Renshuang Ding and Meihua Wang. Design and implementation of web crawler based on coroutine model. In *Cloud Computing and Security - 4th*

International Conference, ICCCS 2018, Haikou, China, June 8-10, 2018, Revised Selected Papers, Part I, pages 427–435, 2018.

[55] Kevin Donnelly, J. J. Hallett, and Assaf J. Kfoury. Formal semantics of weak references. In *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, pages 126–137, 2006.

[56] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS 2009*, pages 42–47, New York, NY, USA, 2009. ACM.

[57] Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, University of California, Santa Barbara, CA, USA, 1999.

[58] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 132–142, 2016.

[59] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theor. Comput. Sci.*, 69(3):243–287, 1989.

[60] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, 2008.

[61] Marcus S. Fisher. *Software verification and validation - an engineering and scientific approach*. Springer, 2007.

[62] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 171–183, 1998.

[63] Martin Fowler and Kendall Scott. *UML distilled - a brief guide to the Standard Object Modeling Language (2. ed.)*. notThenot Addison-Wesley object technology series. Addison-Wesley-Longman, 2000.

[64] Nicu G. Fruja. The correctness of the definite assignment analysis in c#. *Journal of Object Technology*, 3(9):29–52, 2004.

[65] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[66] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, 2014.

[67] Paola Giannini, Marco Servetto, and Elena Zucca. A syntactic model of mutation and aliasing. In *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, DCM/ITRS 2018, and Ninth Workshop on Intersection Types and Related Systems Oxford, UK, 8th July 2018.*, pages 39–55, 2018.

[68] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[69] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 21–40, 2012.

[70] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer, New York, NY, USA, 2012.

[71] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 265–276, 2001.

[72] Philipp Haller and Alexander Loiko. Lacasa: lightweight affinity and object capabilities in scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 272–291, 2016.

[73] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 354–378, 2010.

[74] Mark Harman and Peter W. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*, pages 1–23, 2018.

[75] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

[76] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*, pages 59–69, 2016.

[77] C. A. R. Hoare. Process algebra: A unifying approach. In *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, pages 36–60, 2004.

[78] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard C. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[79] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[80] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[81] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[82] Intel. Intel® 64 and ia-32 architectures developer's manual. `https://software.intel.com/en-us/articles/intel-sdm`, 2019.

[83] Jan Martin Jansen. Programming in the λ-calculus: From church to scott and back. In *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, pages 168–180, 2013.

[84] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 275–288, 2002.

[85] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681, 2013.

[86] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Asp. Comput.*, 27(3):475–497, 2015.

[87] Maria Jump and Kathryn S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 31–38, 2007.

[88] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[89] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[90] Moritz Kleine, Björn Bartels, Thomas Göthel, Steffen Helke, and Dirk Prenzel. *LLVM2CSP*: Extracting CSP models from concurrent programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 500–505, 2011.

[91] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, Y. Thierry-Mieg, K. Wolf, J. van de Pol, C. Rohr, M. Heiner, G. Tran, and J. Srba. Complete Results for the 2016 Edition of the Model Checking Contest. `http://mcc.lip6.fr/2016/results.php`, 2016.

[92] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 7:1–7:6, 2015.

[93] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[94] K. Rustan M. Leino. Applications of extended static checking. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, pages 185–193, 2001.

[95] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[96] Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, pages 21–26, 2015.

[97] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, pages 89–98, 2016.

[98] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1635–1648, 2018.

[99] Eve MacGregor, Yvonne Hsieh, and Philippe Kruchten. Cultural patterns in software process mishaps: incidents in global projects. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[100] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Robert Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 11–19, 2012.

[101] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer System and Sciences*, 17(3):348–375, 1978.

[102] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[103] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, volume 263, 1999.

[104] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 557–570, 2012.

[105] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 245–258, 2009.

[106] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 213–228, 2002.

[107] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100, 2007.

[108] Oscar Nierstrasz. A survey of object-oriented concepts. In *Object-Oriented Concepts, Databases, and Applications*, pages 3–21. ACM, New York, NY, USA, 1989.

[109] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[110] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, pages 158–185, 1998.

[111] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 43–56, 1993.

[112] Eugenio G. Omodeo and Alberto Policriti, editors. *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, volume 10 of *Outstanding Contributions to Logic*. Springer, 2016.

[113] Johan Östlund and Tobias Wrigstad. Welterweight java. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, pages 97–116, 2010.

[114] Johan Östlund and Tobias Wrigstad. Multiple aggregate entry points for ownership types. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 156–180, 2012.

[115] Alex Potanin, Monique Damitio, and James Noble. Are your incoming aliases really necessary? counting the cost of object ownership. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 742–751, 2013.

[116] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 233–269. Springer Berlin Heidelberg, 2013.

[117] Dimitri Racordon and Didier Buchs. A practical type system for safe aliasing. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 133–146, 2018.

[118] Dimitri Racordon and Didier Buchs. Implementing a language with explicit assignment semantics. In *11th International Workshop on Virtual Machines and Intermediate Languages, VMIL 2019, Athens, Grece, October 22, 2019, Proceedings*, 2019.

[119] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, 2017.

[120] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[121] Lukas Rytz. *A Practical Effect System for Scala*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2014.

[122] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[123] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for threadsanitizer. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 110–114, 2011.

[124] Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 208–228, 2015.

[125] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-acsl, a runtime verification tool for safety and security of C programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, pages 164–173, 2017.

[126] Ian Sommerville. *Software engineering, 8th Edition*. International computer science series. Addison-Wesley, 2007.

[127] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[128] Bjarne Stroustrup. *A Tour of C++*. C++ In-Depth Series. Pearson Education, 2018.

[129] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305, 2016.

[130] Synopsys. Coverity. `https://scan.coverity.com`, 2019.

[131] OOO Program Verification Systems. Pvs-studio analyzer. `https://www.viva64.com/en/pvs-studio/`, 2019.

[132] Phit-Huan Tan, Choo-Yee Ting, and Siew-Woei Ling. Learning difficulties in programming courses: Undergraduates' perspective and perception. In *2009 International Conference on Computer Technology and Development*, volume 1, pages 42–46, Nov 2009.

[133] LLVM Team et al. clang: a c language family frontend for llvm. `https://clang.llvm.org`, 2007.

[134] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[135] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011.

[136] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 200–212, 2009.

[137] Alan M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937.

[138] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, pages 86–106, 2012.

[139] Jan Vitek and Boris Bokowski. Confined types in java. *Software: Practice and Experience*, 31(6):507–532, 2001.

[140] Philip Wadler. Linear types can change the world! In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561, 1990.

[141] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*, pages 44–51, 2018.

[142] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings*, pages 1–42, 1992.

[143] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings*, pages 1–116, 1995.

[144] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 459–483, 2011.

[145] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, KTH, School of Information and Communication Technology (ICT), Stockholm, Sweden, 2006.

[146] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 445–469, 2009.

[147] Hongseok Yang and Uday Reddy. Imperative lambda calculus revisited. Technical report, University of Illinois at Urbana-Champaign, Aug 1997.

Appendix A

Formal Companion

This appendix is intended to be a companion to the reading of the formal parts of this thesis. Appendix A.1 lists all symbols and notations used in mathematical formulae. Appendix A.2 gathers all inference rules related to the dynamic and static semantics of the $\mathcal{A}$-calculus.

A.1 List of Symbols and Notations

The following list describes most of the symbols used in mathematical formulae. A comprehensive introduction to these notations is given in Chapter 3.

General Symbols and Notations

$\mathbb{N}$	The set of natural numbers.
$\mathbb{Z}$	The set of integer numbers.
$\mathcal{P}(A)$	The set of all subsets (a.k.a. the powerset) of A.
$\{a \dots b\}$	The set $\{i \mid i \in \mathbb{N} \wedge a \leq i \leq b\}$
$\{s_a \dots s_b\}$	The set $\{s_i \mid a \leq i \leq b\}$
$\| A \|$	The cardinal of a set A.
Σ^*	The set of words over S.
$\Sigma^\#$	The set of words over S without any letter repetition.
ϵ	The empty word.
$\overline{w}$	A word.

$\overline{w}_1\overline{w}_2$	The concatenation of two words $\overline{w}_1$ and $\overline{w}_2$.
$\| \overline{w} \|$	The cardinal (a.k.a. length) of a word $\overline{w}$.
$\| \overline{w} \|_s$	The number of occurences of s in a word $\overline{w}$.
$\overline{w}_i$	The i-th letter of a word $\overline{w}$.
$\overline{w}_{i,j}$	The substring of a word $\overline{w}$ from the i-th to the j-th letter.
$\overline{w}[s]$	The position of a letter s in a word without repetition $\overline{w}$.
dom	The domain of a function, record or table.
codom	The codomain of a function, record or table.
$A \to B$	A total function from A to B.
$A \nrightarrow B$	A partial function from some set $A' \subseteq A$ to B.
$t[^x/_u]$	The substitution of the variable x by u in a term t.
$t[/\sigma]$	The application of the substitutions of a table σ in a term t.

Notations for Composite and Table Types

$\langle l_1 : D_1 \dots l_n : D_n \rangle$	A composite type with labels $l_1 \dots l_n$ and domains $D_1 \dots D_n$.
$\langle l_1 \mapsto v_1 \dots l_n : v_n \rangle$	A record mapping l_i onto v_i.
$\{l_1 \mapsto v_1 \dots l_n : v_n\}$	A table mapping l_i onto v_i.
$r.l$	The field labeled l of a record or table r.
$r[l \mapsto v]$	The update of a field $r.l$ to v.
$r[l \to v \mid p(l)]$	The update of all fields $r.l$ to v for all labels such that $p(l)$.
$r[l_1 \dots l_n \to v]$	The update of all fields $r.l$ to v for all labels $\{l_1 \dots l_n\}$.
$a[b.l \mapsto v]$	The update of a field's field $a.b.l$ to v.
$t\|_l$	The removal of the label l from the domain of a table t.
$t_1 \uplus t_2$	The merging of the table t_2 into t_1.

Symbols and Notations Specific to the $\mathcal{A}$-Calculus

$\mathcal{AC}$	The set of terms of terms of the $\mathcal{A}$-calculus.
t	A term or expression of the $\mathcal{A}$-calculus $t \in \mathcal{AC}$.
$\mathbb{A}$	The set of atomic literals or values.
a	An atomic literal or value $a \in \mathbb{A}$.
$\mathbb{X}$	The set of variable identifiers.
x, y, z	Variable identifiers $x, y, z \in \mathbb{X}$.
f, g	Usually variable identifiers $f, g \in \mathbb{X}$ denoting functions.
$\mathfrak{R}$	The return identifier.
$\mathbb{O}$	The set of assignment operators.
$\diamond$	An assignment operator $\diamond \in \mathbb{O}$.
C	An evaluation context.
$C.\pi$	The pointer table of an evaluation context C.
$C.\mu$	The memory table of an evaluation context C.
$\widetilde{C}$	A frame-aware evaluation context.
$\rightsquigarrow$	The local memory accessibility relation.
$\rightsquigarrow *$	The absolute memory accessibility relation.
$\mathbb{V}$	The set of semantic values.
v	A semantic value $v \in \mathbb{V}$.
$\perp$	The uninitialized memory value $\perp \in \mathbb{V}$.
$\square$	The moved memory value $\square \in \mathbb{V}$.
l, m	Memory locations $l, m \in \mathbb{N}$.
$\mathcal{X}$	The set of memory exceptions.
$\xi_\perp$	The uninitialized memory error $\xi_\perp \in \mathcal{X}$.
$\xi_\square$	The moved memory error $\xi_\square \in \mathcal{X}$.

ξ_{uaf}	The use after free error $\xi_{uaf} \in \mathcal{X}$.
ξ_{df}	The doubly freed memory error $\xi_{df} \in \mathcal{X}$.
ξ_{lk}	The memory leak error $\xi_{lk} \in \mathcal{X}$.
$\mathbb{Q}_R$	The set of reference qualifiers.
`brw`	The borrwed typed qualfier $\texttt{brw} \in \mathbb{Q}_R$.
`own`	The owned typed qualfier $\texttt{own} \in \mathbb{Q}_R$.
$\sqsubset$	The scope belonging relation.
Σ	A scope context $\Sigma : \mathbb{X} \nrightarrow \mathbb{N}$.
$\mathbb{C}$	The set of type capabilities.
$\mathbf{o}$	The ownership type capability.
$(\mathbf{b}, i)$	The borrowed type capability.
$\circ$	The absence of any type capability.
K	A type capability context $K : \mathbb{X} \nrightarrow \mathbb{C} \cup \circ$.
$x!K$	The uniqueness of x in the type capability context K.

A.2 $\mathcal{A}$-Calculus' Semantics

This section gathers the inference rules related to the $\mathcal{A}$-calculus' operational semantics.

Literals

E-Atom
$$\frac{a \in \mathbb{A} \quad \exists l \notin \mathrm{dom}(C.\mu)}{C \vdash a \Downarrow l, C[\mu.l \mapsto a]}$$

E-Fun
$$\frac{\exists l \notin \mathrm{dom}(C.\mu)}{C \vdash \lambda\overline{x} \triangleright t \Downarrow l, C[\mu.l \mapsto \lambda\overline{x} \triangleright t]}$$

Variables

E-Let
$$\frac{C[\pi.x \mapsto 0] \vdash t \Downarrow l, C'}{C \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow l, C'[\pi \mapsto \mathrm{unset}(x, C'.\pi, C.\pi)]}$$

E-Var
$$\frac{x \in \mathrm{dom}(C.\pi)}{C \vdash x \Downarrow C.\pi.x, C}$$

Records

ER-Inst
$$\frac{\exists l \notin \mathrm{dom}(C.\mu) \quad r = \langle \overline{x}_i \mapsto 0 | 1 \leq i \leq \| \overline{x} \| \rangle}{C \vdash \texttt{new}\ \overline{x} \Downarrow_R l, C[\mu.l \mapsto r]}$$

ER-Field-New
$$\frac{C \vdash t \Downarrow_R l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad r = C'.\mu.l \quad \mathrm{rec}(r) \quad x \in \mathrm{dom}(r) \quad m \notin \mathrm{dom}(C'.\mu)}{C \vdash \texttt{alloc}\ t.x \Downarrow_R m, C[\mu.l.x \mapsto m, \mu.m \mapsto \bot]}$$

ER-Field
$$\frac{C \vdash t \Downarrow_R l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad r = C'.\mu.l \quad \mathrm{rec}(r) \quad x \in \mathrm{dom}(r)}{C \vdash t.x \Downarrow r.x, C'}$$

Memory management

E-New
$$\frac{x \in \mathrm{dom}(C.\pi) \quad \exists l \notin \mathrm{dom}(C.\mu)}{C \vdash \texttt{alloc}\ x \Downarrow l, C[\pi.x \mapsto l, \mu.l \mapsto \bot]}$$

E-Del
$$\frac{C \vdash t \Downarrow l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad l \neq 0}{C \vdash \texttt{del}\ t \Downarrow l, C'[\mu \mapsto C'.\mu|_l]}$$

Assignments

E-Alias

$$\frac{C \vdash t \Downarrow l, C'}{C \vdash x \,\&\text{-}\, t \Downarrow l, C'[\pi.x \mapsto l]}$$

E-Mut

$$\frac{C \vdash t \Downarrow m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad l = C'.\pi.x \quad l \neq 0}{C \vdash x := t \Downarrow l, C'[\mu.l \mapsto v]}$$

ER-Mut

$$\frac{C \vdash t \Downarrow m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad u, C'' = \mathrm{copy}(v, C') \quad l = C''.\pi.x \quad l \neq 0}{C \vdash x := t \Downarrow l, C''[\mu.l \mapsto u]}$$

E-Move

$$\frac{C \vdash t \Downarrow m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad l = C'.\pi.x \quad l \neq 0}{C \vdash x \leftarrow t \Downarrow l, C'[\mu.m \mapsto \square][\mu.l \mapsto v]}$$

Field assignments

ER-Field-Alias

$$\frac{C \vdash t_2 \Downarrow_R m, C' \quad C' \vdash t_1 \Downarrow_R l, C'' \quad l \in \mathrm{dom}(C''.\mu) \quad r = C''.\mu.l \quad \mathrm{rec}(r) \quad x \in \mathrm{dom}(r)}{C \vdash t_1.x \,\&\text{-}\, t_2 \Downarrow_R m, C''[\mu.l.x \mapsto m]}$$

ER-Field-Mut

$$\frac{C \vdash t_2 \Downarrow_R m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad u, C_u = \mathrm{copy}(v, C') \quad C_u \vdash t_1.x \Downarrow_R l, C'' \quad C''.\mu.l \neq 0}{C \vdash t_1.x \,\&\text{-}\, t_2 \Downarrow_R l, C''[\mu.l \mapsto u]}$$

ER-Field-Move

$$\frac{C \vdash t_2 \Downarrow_R m, C' \quad m \in \mathrm{dom}(C'.\mu) \quad v = C'.\mu.m \quad v \notin \{\bot, \square\} \quad C' \vdash t_1.x \Downarrow_R l, C'' \quad C''.\mu.l \neq 0}{C \vdash t_1.x \leftarrow t_2 \Downarrow_R l, C''[\mu.l \mapsto v, \mu.m \mapsto \square]}$$

Sequences

E-Seq

$$\frac{C \vdash t_1 \Downarrow l_1, C_1 \quad C_1 \vdash t_2 \Downarrow l_2, C_2}{C \vdash t_1; t_2 \Downarrow l_2, C_2}$$

Conditional terms

EC-Cond-T

$$\frac{\begin{array}{c} C \vdash t_1 \Downarrow_C l_1, C' \\ l_1 \in \mathrm{dom}(C'.\mu) \quad C'.\mu.l_1 \notin \{\bot, \square\} \quad C'.\mu.l_1 = \texttt{true} \quad C' \vdash t_2 \Downarrow_C l_2, C'' \end{array}}{C \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_C l_2, C''}$$

EC-Cond-F

$$\frac{\begin{array}{c} C \vdash t_1 \Downarrow_C l_1, C' \\ l_1 \in \mathrm{dom}(C'.\mu) \quad C'.\mu.l_1 \notin \{\bot, \square\} \quad C'.\mu.l_1 = \texttt{false} \quad C' \vdash t_3 \Downarrow_C l_3, C'' \end{array}}{C \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow_C l_3, C''}$$

Functions

E-Callee

$$\frac{C \vdash t_c \Downarrow l, C' \quad l \in \mathrm{dom}(C'.\mu) \quad C'.\mu.l \notin \{\bot, \square\} \quad C'.\mu.l = \lambda\overline{x} \triangleright t}{C \vdash t_c \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C'}$$

E-Args-0

$$\frac{}{C \vdash \epsilon \Downarrow^{\text{args}} 0, C}$$

E-Args-Alias

$$\frac{C[\pi.x \mapsto 0] \vdash x \;\&\text{-}\; t \Downarrow l_x, C_x \quad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x \;\&\text{-}\; t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

E-Args-MM

$$\frac{\diamond \in \{:=, \leftarrow\} \quad \exists l \notin \mathrm{dom}(C.\mu) \quad C[\pi.x \mapsto l, \mu.l \mapsto \bot] \vdash x \diamond t \Downarrow l_x, C_x \quad C_x \vdash \overline{u} \Downarrow^{\text{args}} 0, C'}{C \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} 0, C'}$$

E-Ret-Alias

$$\frac{C[\pi.\mathfrak{R} \mapsto 0] \vdash \mathfrak{R} \;\&\text{-}\; t \Downarrow l, C'}{C \vdash \texttt{ret} \diamond t \Downarrow l, C'}$$

E-Ret-MM

$$\frac{\diamond \in \{:=, \leftarrow\} \quad \exists l \notin \mathrm{dom}(C.\mu) \quad C[\pi.\mathfrak{R} \mapsto l, \mu.l \mapsto \bot] \vdash \mathfrak{R} \diamond t \Downarrow l, C'}{C \vdash \texttt{ret} \diamond t \Downarrow l, C'}$$

E-Call

$$\frac{C \vdash f \Downarrow^{\text{callee}} \lambda\overline{x} \triangleright t, C_1 \quad \lambda\overline{x}' \triangleright t' = \mathrm{rename}_{C_1}(\lambda\overline{x} \triangleright t) \quad C_1 \vdash \overline{u}[/\{\forall_{i=1}^{\|\overline{x}\|}, \overline{x}_i \mapsto \overline{x}'_i\}] \Downarrow^{\text{args}} 0, C_2 \quad C_2 \vdash t' \Downarrow l, C'}{C \vdash f(\overline{u}) \Downarrow l, C'[\pi \mapsto \mathrm{unset}(\mathfrak{R}, C_2.\pi, C.\pi)]}$$

A.3 $\mathcal{A}$-Calculus' Type System

This section gathers the inference rules related to the $\mathcal{A}$-calculus' type system.

A.3.1 Flow-Insensitive Typing

Literals

S-Atom
$$\frac{a \in \mathbb{A}}{\Sigma, i \vdash a \sqsubset -\infty}$$

S-Fun
$$\frac{(\lambda \overline{x} \triangleright t) : \tau \quad \Sigma' = \text{funscope}(\tau) \quad \Sigma', \min(\{\Sigma'.x \mid x \in \overline{x}\}) - 1 \vdash t \sqsubset j}{\Sigma, i \vdash \lambda \overline{x} \triangleright t \sqsubset -\infty}$$

Variables

S-Let
$$\frac{\Sigma[x \mapsto i - 1], i - 1 \vdash t \sqsubset j}{\Sigma, i \vdash \texttt{let}\ x\ \texttt{in}\ t \sqsubset j}$$

S-Var
$$\frac{x \in \text{dom}(\Sigma)}{\Sigma, i \vdash x \sqsubset \Sigma.x}$$

Assignments

S-Alias
$$\frac{\Sigma, i \vdash x \sqsubset j' \quad \Sigma, i \vdash u \sqsubset j \quad j < j'}{\Sigma, i \vdash x\ \&\text{-}\ u \sqsubset j}$$

S-Mut
$$\frac{\Sigma, i \vdash x \sqsubset j \quad \Sigma, i \vdash u \sqsubset j'}{\Sigma, i \vdash x := u \sqsubset j}$$

S-Move
$$\frac{\Sigma, i \vdash x \sqsubset j \quad \Sigma, i \vdash u \sqsubset j'}{\Sigma, i \vdash x \leftarrow u \sqsubset j}$$

Sequences

S-Seq
$$\frac{\Sigma, i \vdash t_1 \sqsubset j_1 \quad \Sigma, i \vdash t_1 \sqsubset j_2}{\Sigma, i \vdash t_1\ ;\ t_2 \sqsubset j_2}$$

Conditional terms

S-Cond
$$\frac{\Sigma, i \vdash t_1 \sqsubset j_1 \quad \Sigma, i \vdash t_2 \sqsubset j_2 \quad \Sigma, i \vdash t_3 \sqsubset j_3}{\Sigma, i \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \sqsubset \min(j_2, j_3)}$$

Functions

S-Ret
$$\frac{\Sigma, i \vdash \Re \diamond t \sqsubset j}{\Sigma, i \vdash \texttt{ret}\ \diamond t \sqsubset j}$$

S-Call
$$\frac{f : \tau \quad \forall (x \diamond t) \in \overline{u}, \Sigma, i \vdash t \sqsubset j_x}{\Sigma, i \vdash f(\overline{u}) \sqsubset \min(\{j_x \mid x \in \tau.bds\} \cup \{i - 1\})}$$

A.3.2 Flow-Sensitive Typing

Literals

$$\textsc{K-Atom}\ \frac{a \in \mathbb{A}}{K \vdash a \Downarrow \mathbf{o}, K} \qquad \textsc{K-Fun}\ \frac{}{K \vdash \lambda \overline{x} \triangleright t \Downarrow \mathbf{o}, K}$$

Variables

$$\textsc{K-Let}\ \frac{K[x \mapsto \ominus] \vdash t \Downarrow c, K'}{K \vdash \texttt{let}\ x\ \texttt{in}\ t \Downarrow c, K'|_x} \qquad \textsc{K-Var}\ \frac{x \in \mathrm{dom}(K)}{K \vdash x \Downarrow K.x, K}$$

Assignments

K-Alias-A
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x \neq \mathbf{o} \vee x!K'}{K \vdash x \,\&\text{-}\, y \Downarrow \mathbf{o}, K'[x \leftarrow (\mathbf{b}, \{y\})]}$$

K-Alias-B
$$\frac{K \vdash t \Downarrow c, K' \quad c = (\mathbf{b}, O) \quad x \notin O \quad K'.x \neq \mathbf{o} \vee x!K'}{K \vdash x \,\&\text{-}\, t \Downarrow \mathbf{o}, K'[x \leftarrow (\mathbf{b}, O)]}$$

K-Mut-A
$$\frac{K \vdash t \Downarrow c, K' \quad c \neq \ominus \quad K'.x = \ominus}{K \vdash x := t \Downarrow \mathbf{o}, K'[x \mapsto \mathbf{o}]}$$

K-Mut-B
$$\frac{K \vdash t \Downarrow c, K' \quad c \neq \ominus \quad K'.x \neq \ominus}{K \vdash x := t \Downarrow K'.x, K'}$$

K-Move-A
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x = \ominus}{K \vdash x \leftarrow y \Downarrow \mathbf{o}, K'[x \leftarrow \mathbf{o}, y \mapsto \ominus]}$$

K-Move-B
$$\frac{K \vdash y \Downarrow c, K' \quad y \in \mathbb{X} \quad y!K' \quad x \neq y \quad K'.x \neq \ominus}{K \vdash x \leftarrow y \Downarrow K'.x, K'[y \mapsto \ominus]}$$

K-Move-C
$$\frac{K \vdash t \Downarrow c, K' \quad t \notin \mathbb{X} \quad c = \mathbf{o} \quad K'.x = \ominus}{K \vdash x \leftarrow t \Downarrow \mathbf{o}, K'[x \leftarrow \mathbf{o}]}$$

K-Move-D
$$\frac{K \vdash t \Downarrow c, K' \quad t \notin \mathbb{X} \quad c = \mathbf{o} \quad K'.x \neq \ominus}{K \vdash x \leftarrow t \Downarrow K'.x, K'}$$

Sequences

K-Seq
$$\frac{K \vdash t_1 \Downarrow c_1, K_1 \quad K_1 \vdash t_2 \Downarrow c_2, K_2}{K \vdash t_1; t_2 \Downarrow c_2, K_2}$$

Conditional terms

K-Cond

$$\frac{K \vdash t_1 \Downarrow c_1, K_1 \quad K_1 \vdash t_2 \Downarrow c_t, K_t \quad K_1 \vdash t_3 \Downarrow c_e, K_e \quad K' = \text{merge}(K_t, K_e)}{K \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow \ominus, K'}$$

Functions

K-Args-0

$$\frac{}{K, A \vdash \epsilon \Downarrow^{\text{args}} K, A}$$

K-Args-N

$$\frac{y \notin \text{dom}(K) \quad K[y \mapsto \ominus] \vdash y \diamond t \Downarrow c_x, K_x \quad K_x, A[x \mapsto c_x] \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K, A \vdash (x \diamond t)\overline{u} \Downarrow^{\text{args}} K', A'}$$

K-Ret

$$\frac{K \vdash \mathfrak{R} \diamond t \Downarrow c, K'}{K \vdash \texttt{ret}\ \diamond t \Downarrow c, K'}$$

K-Call-A

$$\frac{f : \tau \quad \tau.codom.qual = \texttt{own} \quad K \vdash f \Downarrow c_f, K_f \quad c_f \neq \ominus \quad K_f, \varnothing \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K \vdash f(\overline{u}) \Downarrow \mathbf{o}, K'}$$

K-Call-B

$$\frac{f : \tau \quad \tau.codom.qual = \texttt{brw} \quad K \vdash f \Downarrow c_f, K_f \quad c_f \neq \ominus \quad K_f, \varnothing \vdash \overline{u} \Downarrow^{\text{args}} K', A'}{K \vdash f(\overline{u}) \Downarrow (\mathbf{b}, \text{owners}(A')), K'}$$

Appendix B

Anzen

This appendix describes Anzen and AIR's concrete syntax.

B.1 Anzen's Concrete Syntax

Below is given Anzen's concrete syntax, in the form of EBNF rules.

⟨*program*⟩ ::= ⟨*stmt-list*⟩

⟨*stmt-list*⟩ ::= `{` ⟨*stmt*⟩ ⟨*stmt-sep*⟩ `}`

⟨*stmt-sep*⟩ ::= '`;`'
  | NEWLINE

⟨*stmt*⟩ ::= ⟨*scope*⟩
  | ⟨*prop-decl*⟩
  | ⟨*fun-decl*⟩
  | ⟨*struct-decl*⟩
  | ⟨*while-stmt*⟩
  | ⟨*ret-stmt*⟩
  | ⟨*bind-stmt*⟩
  | ⟨*expr*⟩

⟨*scope*⟩ ::= '`{`' ⟨*stmt-list*⟩ '`}`'

⟨*prop-decl*⟩ ::= ('`let`' | '`var`') ⟨*ident*⟩ `[` '`:`' ⟨*type-sign*⟩ `]` `[` ⟨*bind-op*⟩ ⟨*expr*⟩ `]`

⟨*fun-decl*⟩ ::= '`fun`' (⟨*ident*⟩ | ⟨*expr-op*⟩) `[` '`<`' ⟨*phldr-list*⟩ '`>`' `]` ⟨*fun-type*⟩ ⟨*scope*⟩

⟨*struct-decl*⟩ ::= '`struct`' `<ident>` `"["` '`<`' ⟨*phldr-list*⟩ '`>`' `]` '`{`' ⟨*mbr-list*⟩ '`}`'

⟨*mbr-list*⟩ ::= { ⟨*mdr*⟩ ⟨*stmt-sep*⟩ }

⟨*struct-mbr*⟩ ::= ⟨*field*⟩
  | ⟨*mthd*⟩
  | ⟨*ctor*⟩
  | ⟨*dtor*⟩

⟨*field*⟩ ::= ('let' | 'var') ⟨*ident*⟩ ':' ⟨*type-sign*⟩

⟨*mthd*⟩ ::= ['mutating'] ⟨*fun-decl*⟩

⟨*ctor*⟩ ::= 'new' '<' ⟨*phldr-list*⟩ '>' ⟨*fun-type*⟩ ⟨*scope*⟩

⟨*dtor*⟩ ::= 'del' '(' ')' ⟨*scope*⟩

⟨*while-stmt*⟩ ::= 'while' ⟨*expr*⟩ ⟨*scope*⟩

⟨*ret-stmt*⟩ ::= 'return' [⟨*bind-op*⟩ ⟨*expr*⟩]

⟨*bind-stmt*⟩ ::= ⟨*expr*⟩ ⟨*bind-op*⟩ ⟨*expr*⟩

⟨*type-sign*⟩ ::= ⟨*qual-type*⟩
  | '(' ⟨*qual-type*⟩ ')'

⟨*qual-type*⟩ ::= ⟨*qual-list*⟩ ⟨*unqual-type*⟩
  | ⟨*unqual-type*⟩

⟨*qual-list*⟩ ::= ⟨*qual*⟩ { ⟨*qual*⟩ }

⟨*qual*⟩ ::= '@cst' | '@mut' | '@brw' | '@own'

⟨*unqual-type*⟩ ::= ⟨*ident*⟩ ['<' ⟨*spec-list*⟩ '>']
  | ⟨*fun-type*⟩

⟨*fun-type*⟩ ::= '(' [⟨*param-list*⟩] ')' '->' ⟨*type-sign*⟩

⟨*param-list*⟩ ::= ⟨*param*⟩ { ',' ⟨*param*⟩ } [',']

⟨*param*⟩ ::= ⟨*ident*⟩ ':' ⟨*type-sign*⟩

⟨*phldr-list*⟩ ::= ⟨*ident*⟩ { ',' ⟨*ident*⟩ } [',']

⟨*spec-list*⟩ ::= ⟨*spec*⟩ { ',' ⟨*spec*⟩ } [',']

⟨*spec*⟩ ::= ⟨*ident*⟩ '=' ⟨*unqual-type*⟩

⟨*expr*⟩ ::= ⟨*atom*⟩ ⟨*post-expr*⟩

⟨*post-expr*⟩ ::= 'as' ⟨*unqual-type*⟩
  | ⟨*binary-op*⟩ ⟨*expr*⟩
  | '(' ⟨*expr*⟩ ')'
  | '.' ⟨*expr*⟩

⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')'
  | ⟨*unary-op*⟩ ⟨*expr*⟩
  | ⟨*ident*⟩
  | ⟨*if-expr*⟩
  | ⟨*fun-expr*⟩
  | ⟨*literal*⟩

⟨*ident*⟩ ::= (CHARACTER | '_') { ALPHANUM | '_' }

⟨*if-expr*⟩ ::= 'if' ⟨*expr*⟩ ⟨*scope*⟩ ['else' ⟨*scope*⟩]

⟨*fun-expr*⟩ ::= 'fun' ⟨*fun-type*⟩ ⟨*scope*⟩

⟨*bind-op*⟩ ::= '<-' | ':=' | '&-'

⟨*expr-op*⟩ ::= ⟨*binary-op*⟩
  | ⟨*unary-op*⟩

⟨*binary-op*⟩ ::= '+' | '-' | '*' | '/' | '<' | '<=' | '==' | '!=' | '>=' | '>'

⟨*unary-op*⟩ ::= '+' | '-' | '!'

⟨*literal*⟩ ::= ⟨*int*⟩
  | ⟨*float*⟩
  | ⟨*str*⟩
  | ⟨*bool*⟩

⟨*int*⟩ ::= (DIGIT - '0') { DIGIT }

⟨*float*⟩ ::= (⟨*int*⟩ | '0') '.' ⟨*int*⟩

⟨*str*⟩ ::= '"' { * - '"' } '"'

⟨*bool*⟩ ::= 'true' | 'false'

B.2 AIR's Concrete Syntax

Below is given AIR's concrete syntax, in the form of EBNF rules.

⟨*program*⟩ ::= ⟨*struct-decls*⟩ ⟨*fun-decls*⟩

⟨*struct-decls*⟩ ::= { ⟨*struct-decl*⟩ NEWLINE }

⟨*struct-decl*⟩ ::= 'struct' ⟨*ident*⟩ ⟨*struct-body*⟩

⟨*struct-body*⟩ ::= '{' ⟨*type-sign*⟩ { ',' ⟨*type-sign*⟩ } '}'

⟨*fun-decls*⟩ ::= { ⟨*fun-decl*⟩ NEWLINE }

⟨*fun-decl*⟩ ::= 'fun' ⟨*ident*⟩ ⟨*fun-type*⟩ ⟨*fun-body*⟩

⟨*fun-body*⟩ ::= '{' { ⟨*block*⟩ } '}'

⟨*block*⟩ ::= ⟨*label*⟩ NEWLINE { ⟨*inst*⟩ NEWLINE }

⟨*inst*⟩ ::= ⟨*alloc-inst*⟩
| ⟨*apply-inst*⟩
| ⟨*bind-inst*⟩
| ⟨*branch-inst*⟩
| ⟨*copy-inst*⟩
| ⟨*drop-inst*⟩
| ⟨*extract-inst*⟩
| ⟨*jump-inst*⟩
| ⟨*mkref-inst*⟩
| ⟨*move-inst*⟩
| ⟨*papply-inst*⟩
| ⟨*ret-inst*⟩
| ⟨*ucast-inst*⟩

⟨*alloc-inst*⟩ ::= ⟨*register*⟩ '=' 'alloc' ⟨*type-sign*⟩

⟨*apply-inst*⟩ ::= ⟨*register*⟩ '=' 'apply' ⟨*ref*⟩ { ⟨*arg-list*⟩ }

⟨*arg-list*⟩ ::= ⟨*value*⟩ { ',' ⟨*value*⟩ }

⟨*bind-inst*⟩ ::= 'bind' ⟨*value*⟩ ',' ⟨*register*⟩

⟨*branch-inst*⟩ ::= 'branch' ⟨*value*⟩ ',' ⟨*label*⟩ ',' ⟨*label*⟩

⟨*copy-inst*⟩ ::= 'copy' ⟨*value*⟩ ',' ⟨*register*⟩

⟨*drop-inst*⟩ ::= 'drop' ⟨*register*⟩

⟨*extract-inst*⟩ ::= ⟨*register*⟩ '=' 'extract' ⟨*register*⟩ ',' ⟨*int*⟩

⟨*jump-inst*⟩ ::= 'jump' ⟨*label*⟩

⟨*makeref-inst*⟩ ::= ⟨*register*⟩ '=' 'make_ref' ⟨*type-sign*⟩

⟨*move-inst*⟩ ::= 'copy' ⟨*value*⟩ ',' ⟨*register*⟩

⟨*papply-inst*⟩ ::= 'partial_apply' ⟨*ref*⟩ { ⟨*arg-list*⟩ }

⟨*ret-inst*⟩ ::= 'ret' [⟨*ref*⟩]

⟨*ucast-inst*⟩ ::= ⟨*register*⟩ '=' 'ucast' ⟨*value*⟩ ',' ⟨*type-sign*⟩

⟨*label*⟩ ::= '#' ⟨*ident*⟩

⟨*value*⟩ ::= ⟨*ref*⟩
| ⟨*literal*⟩

⟨*ref*⟩ ::= ⟨*register*⟩
| ⟨*fun-ptr*⟩

⟨*register*⟩ ::= '%' ⟨*int*⟩

⟨*fun-ptr*⟩ ::= '@' ⟨*ident*⟩

⟨*type-sign*⟩ ::= ⟨*ident*⟩
| ⟨*fun-type*⟩

⟨*fun-type*⟩ ::= '(' [⟨*param-list*⟩] ')' '->' ⟨*type-sign*⟩

⟨*param-list*⟩ ::= ⟨*param*⟩ { ',' ⟨*param*⟩ }

⟨*param*⟩ ::= ⟨*ident*⟩ ':' ⟨*type-sign*⟩

⟨*ident*⟩ ::= (CHARACTER | '_') { ALPHANUM | '_' }

⟨*literal*⟩ ::= ⟨*int*⟩
| ⟨*float*⟩
| ⟨*str*⟩
| ⟨*bool*⟩

⟨*int*⟩ ::= (DIGIT - '0') { DIGIT }

⟨*float*⟩ ::= (⟨*int*⟩ | '0') '.' ⟨*int*⟩

⟨*str*⟩ ::= '"' { * - '"' } '"'

⟨*bool*⟩ ::= 'true' | 'false'

Appendix C

Full Proof of Property 6.3.2

Let K be a well-formed capability context. If $K \vdash t \Downarrow c, K'$, then K' is well-formed.

Proof. The proof is by induction over the typing judgements.

- Atoms
  - Case K-LET
    1. The proof is trivial, by the fact that K is not modified.
- Functions
  - Case K-FUN
    1. The proof is trivial, by the fact that K is not modified.
- Variables' dereferencing
  - Case K-VAR
    1. The proof is trivial, by the fact that K is not modified.
- Variable declarations
  - Case K-LET
    1. $K[x \mapsto \ominus]$ is well-formed.
    2. By 1 and the induction hypothesis, $K[x \mapsto \ominus] \vdash t \Downarrow c, K'$ implies that K' is well-formed.
- Aliasing assignments
  - Case K-ALIAS-A
    1. By the induction hypothesis, $K \vdash y \Downarrow c, K'$ implies that K' is well-formed.

2. By definition, $y!K'$ implies that $K'.y = \mathbf{o}$.
3. $\nexists z \in \mathrm{dom}(K'), K'.z = (\mathbf{b}, O) \wedge x \in O$.
   (a) either by definition if $x!K'$,
   (b) or by contradiction, as $K'.x \neq \mathbf{o} \wedge \exists z, K'.z = (\mathbf{b}, O) \wedge x \in O$ implies that K' is not well-formed.
4. By 1, 2, and 3, $K'[x \mapsto (\mathbf{b}, \{y\})]$ is well-formed.

– Case K-Alias-B

1. By the induction hypothesis, $K \vdash t \Downarrow c, K'$ implies that K' is well-formed.
2. $\nexists z \in \mathrm{dom}(K'), K'.z = (\mathbf{b}, O) \wedge x \in O$.
   (a) either by definition if $x!K'$,
   (b) or by contradiction, as $K'.x \neq \mathbf{o} \wedge \exists z, K'.z = (\mathbf{b}, O) \wedge x \in O$ implies that K' is not well-formed.
3. By 1 and 2, $K'[x \mapsto (\mathbf{b}, \{y\})]$ is well-formed.

- Mutation assignments
  – Case K-Mut-A
    1. By the induction hypothesis, $K \vdash t \Downarrow c, K'$ implies that K' is well-formed.
    2. By 1, $K'[x \mapsto \mathbf{o}]$ is well-formed.
  – Case K-Mut-B
    1. By the induction hypothesis, $K \vdash t \Downarrow c, K'$ implies that K' is well-formed.
- Move assignments
  – Case K-Move-A
    1. By the induction hypothesis, $K \vdash y \Downarrow c, K'$ implies that K' is well-formed.
    2. By definition, $y!K'$ implies that $\nexists z \in \mathrm{dom}(K'), K'.z = (\mathbf{b}, O) \wedge y \in O$.
    3. By 1 and 2, $K'[x \mapsto \mathbf{o}, y \mapsto \ominus]$ is well-formed.
  – Case K-Move-B
    1. By the induction hypothesis, $K \vdash y \Downarrow c, K'$ implies that K' is well-formed.

2. By definition, $y!K'$ implies that $\nexists z \in \mathrm{dom}(K'), K'.z = (\mathbf{b}, O) \wedge y \in O$.
3. By 1 and 2, $K'[x \mapsto \mathbf{o}, y \mapsto \ominus]$ is well-formed.

– Case K-Move-C

1. By the induction hypothesis, $K \vdash y \Downarrow c, K'$ implies that K' is well-formed.
2. By 1, $K'[x \mapsto \mathbf{o}]$ is well-formed.

– Case K-Move-D

1. By the induction hypothesis, $K \vdash y \Downarrow c, K'$ implies that K' is well-formed.

- Function applications

1. If K is well-formed and $K, A \vdash \overline{u} \Downarrow^{\text{args}} K', A'$, then K' is well-formed.
   (a) The proof is trivial for the case K-Args-0
   (b) Case K-Args-N
      i. $y \notin \mathrm{dom}(K)$ implies that $K[y \mapsto \ominus]$ is well-formed.
      ii. By the induction hypothesis, $K[y \mapsto \ominus] \vdash y \diamond t \Downarrow c_x, K_x$ implies that K_x is well-formed.
      iii. By 1.b.ii and and the induction hypothesis, $K_x, A[x \mapsto c_x] \vdash \overline{u} \Downarrow^{\text{args}} K', A'$ implies that K' is well-formed.
2. Cases K-Call-A and K-Call-B
   (a) By the induction hypothesis, $K \vdash f \Downarrow c_f, K_f$ implies that K_f is well-formed.
   (b) By 1 and the induction hypothesis, $K_f, \varnothing \vdash \overline{u} \Downarrow^{\text{args}} K', A'$ implies that K' is well-formed.

- Return statements

– Case K-Ret

1. By the induction hypothesis, $K \vdash \mathfrak{R} \diamond t \Downarrow c, K'$ implies that K' is well-formed.

- Sequences

– Case K-Seq

1. By the induction hypothesis, $K \vdash t_1 \Downarrow c_1, K_1$ implies that K_1 is well-formed.

2. By 1 and the induction hypothesis, $K_1 \vdash t_2 \Downarrow c_2, K_2$ implies that K_2 is well-formed.

- Conditional statements
  - Case K-COND
    1. By the induction hypothesis, $K \vdash t_1 \Downarrow c_1, K_1$ implies that K_1 is well-formed.
    2. By 1 and the induction hypothesis, $K_1 \vdash t_2 \Downarrow c_t, K_t$ implies that K_t is well-formed.
    3. By 1 and the induction hypothesis, $K_1 \vdash t_3 \Downarrow c_e, K_e$ implies that K_e is well-formed.
  - $K' = \text{merge}(K_t, K_e)$ implies that K' is well-formed.

□