Amortised Resource Analysis for Object-Oriented Programs

Dulma Rodriguez

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

Eingereicht am 28. Juni 2012

Amortised Resource Analysis for Object-Oriented Programs

Dulma Rodriguez

Erstgutachter:	Prof. Martin Hofmann, PhD Ludwig Maximilians-Universität München
Zweitgutachter:	Prof. Dr. C.-H. Luke Ong University of Oxford
Abgabedatum:	28. Juni 2012
Disputationsdatum:	5. Oktober 2012

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, §8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Dulma Rodriguez
München, 28. Juni 2012

Abstract

As software systems rise in size and complexity, the need for verifying some of their properties increases. One important property to be verified is the resource usage, i.e. how many resources the program will need for its execution, where resources include execution time, memory, power, etc. Resource usage analysis is important in many areas, in particular embedded systems and cloud computing. Thus, resource analysis has been widely researched and some different approaches to this have been proposed based in particular on recurrence solving, abstract interpretation and amortised analysis.

In the amortised analysis technique, a nonnegative number, called *potential*, is assigned to a data structure. The *amortised cost* of operations is then defined by its actual cost plus the difference in potential of the data structure before and after performing the operation. Amortised analysis has been used for automatic resource analysis of functional and object-oriented programs. The potentials are defined using refined types and typing rules then ensure that potential and actual resource usage is accounted for correctly. The automatic inference of the potential functions can then be achieved by type inference.

In the case of functional programs, the structure of the types is known. Thus, type inference can be reduced to solving linear arithmetic constraints. For object-oriented programs, however, the refined types are more complicated because of the general nature of objects: they can be used to define any data structure. Thus, the type inference must discover not only the potential functions for the data structure but also the data structures themselves. Other features of object-oriented programs that complicate the analysis are aliasing and imperative update. Hofmann and Jost presented in 2006 a type system for amortised heap-space analysis of object-oriented programs, called Resource Aware JAva (RAJA). However, they left the problem of type inference open.

In this thesis we present a type inference algorithm for the RAJA system. We were able to reduce the type inference problem to the novel problem of satisfiability of arithmetic constraints over infinite trees and we developed a heuristic algorithm for satisfiability of these constraints. We proved the soundness of the type inference algorithm and developed an OCaml implementation and experimental evaluation that shows that we can compute

linear upper-bounds to the heap-space requirements of many programs, including sorting algorithms for lists such as insertion sort and merge sort and also programs that contain different interacting objects that describe real-life scenarios like a bank account.

Another contribution of this thesis is a type checking algorithm for the RAJA system that is useful for verifying the types discovered by the type inference by using the *proof carrying code* technology.

Zusammenfassung

Eine wichtige Eigenschaft von Software ist der Ressourcenverbrauch; also der Verbrauch von Ausführungszeit, Speicher, und anderen quantifizierbaren Größen. Speziell bei eingebetteten Systemen und in Situationen wo Rechenleistung abgerechnet wird ist die automatische Vorhersage des Ressourcenverbrauchs sehr wünschenswert. Es wurden daher verschiedene Ansätze zur Lösung dieses Problems entwickelt, insbesondere Rekurrenzgleichungen, abstrakte Interpretation und die amortisierte Analyse, mit der sich auch die vorliegende Arbeit befasst.

In der amortisierten Analyse wird eine nichtnegative Zahl, genannt *Potenzial*, einer Datenstruktur zugewiesen. Die *amortisierten Kosten* von Operationen ergeben sich aus den tatsächlichen Kosten zuzüglich der Differenz im Potenzial der Datenstruktur vor und nach der Durchführung der Operation. Indem man die Potenzialfunktionen als (verfeinerte) Typen auffasst, wird die Aufgabe der automatischen Findung der Potentiale zu einem Typinferenzproblem.

Bei funktionalen Programmen ist die Struktur der Typen bekannt, deswegen kann die Typinferenz auf die Lösung linearer arithmetischer Ungleichungen reduziert werden. Aber da Objekte beliebige Datenstrukturen definieren können, sind die verfeinerte Typen für objekt-orientierte Programme viel komplizierter. Aus diesem Grund muss die Typinferenz nicht nur die Potenzialfunktionen entdecken sondern auch die codierten Datenstrukturen an sich. Die Analyse wird durch weitere Eigenschaften von objekt-orientierten Programme erschwert wie das Aliasing und imperatives Update. Hofmann und Jost haben 2006 ein Typsystem für die amortisierte Heap-Space-Analyse von objekt-orientierten Programmen vorgestellt; genannt Resource Aware JAva. Allerdings blieb das Problem der Typinferenz offen.

In dieser Arbeit präsentieren wir ein Algorithmus zur Typinferenz für das System RAJA. Wir reduzieren die Typinferenz auf das neuartige Problem der Erfüllbarkeit von arithmetischen Ungleichungen über unendliche Bäume. Darüber hinaus entwickeln wir einen heuristischen Algorithmus für die Erfüllbarkeit dieser Ungleichungen. Zusätzlich haben wir die Korrektheit des Algorithmus bewiesen. Wir haben den Algorithmus in OCaml implementiert. Experimente belegen, dass wir lineare obere Schranken an

den Heap-Space-Verbrauch vieler Programme berechnen können, z.B. für Sortieralgorithmen für Listen wie Sortieren durch Einfügen oder Mischen, sowie Programme die verschiedene interagierende Objekte enthalten, die reale Szenarien beschreiben wie z.B. ein Bankkonto.

Ein weiterer Beitrag dieser Arbeit ist ein Type-Checking Algorithmus. Damit ist eine Uberprüfung der Typen möglich, die die Typinferenz liefert, z.B. mit Hilfe der *proof carrying code* Technologie.

Acknowledgements

This thesis would not have been possible without the dedication and support of my supervisor Martin Hofmann. I thank him for suggesting this difficult research topic and believing in my ability to master it.

I also thank Andreas Abel for many interesting discussions on type systems and fixpoints. I thank Steffen Jost for kindly giving me access to all his drafts related to the RAJA system and for many insightful discussions.

I thank Luke Ong for inviting me to Oxford to discuss my work and for his many useful questions about RAJA and its relation to other research; these encouraged me to seek a deeper understanding of various issues.

Andrew Gordon's Marktoberdorf 2009 lecture inspired me to relate the type inference for RAJA to that of refinement types.

I thank Ramyaa for various insightful discussions on the relation of constraints over infinite trees to recurrence relations. I thank my colleagues for listening to numerous talks on my research and providing useful feedback; in particular Jan Hoffmann, Javier Esparza, Helmut Seidl, Lennart Beringer, Robert Grabowski, Markus Latte, Ulrich Schöpp and Vivek Nigam. I also thank Max Jakob for technical support, in particular for setting up the RAJA demo web-page.

Special thanks to Martin Churchill for listening patiently to my research problems and providing me with useful advice, and for proofreading chapters of this thesis. Many thanks also to my family for their constant love and support.

I acknowledge financial support from the DFG Graduiertenkolleg 1480 (PUMA) and the EU integrated project MOBIUS IST 15905.

Contents

List of Figures

Chapter 1

Introduction

This thesis studies the problem of analysing the resource consumption of programs automatically. In particular, it focuses on the analysis of the heap-space requirements of object-oriented programs.

Our system takes a Java-like program and computes statically a linear bound to its heap space requirements, as a function of the size of its input. The analysis is fully automatic, i.e. it does not require any user input. Whenever the analysis returns a result, it is guaranteed to be correct. Moreover, the system can analyse many interesting programs.

1.1 Motivation

The study of resource analysis is very relevant, in particular for real-time and embedded systems, which need to respond to externally generated input stimuli within a finite and specified period (a deadline). Failing to meet the deadline can result in loss of life, damage to environment or economic loss. Hence, for those systems, it is very important to obtain an upper bound on the execution time of the programs to ensure that they will meet their deadlines even in the worst case. But the analysis of other resources is relevant as well: an application that needs a particular resource for its execution and can not obtain it, will also fail to meet its deadline. For instance, it is important to ensure that applications will not run out of memory. The probability of this event would be rather insignificant if those systems had unlimited resources available, but embedded systems often run in small devices with constrained resources.

The size and complexity of real-time systems vary from a few hundred lines of assembler or C to more than a million lines of Ada code in the core system of the Space Station Freedom [Rai92]. It has been estimated that 99% of the worldwide production of microprocessors is used in embedded systems [Tur99].

Traditionally, embedded systems have been written in languages such

as C, C++ or Ada. In the last years, Java has also emerged as a suitable language for real-time systems. With Java, developers can target a platform-independent API and migrate their applications to different devices without recompiling them. Further, the object-oriented nature of Java supports well-structured development and software reuse. Errors introduced by the need to manage memory explicitly in languages such as C and C++ are some of the most difficult problems to diagnose. One of the Java programming model's major strengths is that the Java Virtual Machine (JVM) performs memory management, other than the application. However, traditional garbage collectors can introduce long delays at times that are impossible for the application programmer to predict.

The Real-time Specification for Java (RTSJ) [BBG+00] is a set of interfaces and behavioural specifications that was created to address some of the limitations of the Java language that prevent its widespread use in real-time systems.

To support tasks that cannot tolerate garbage collection interruptions, the RTSJ defines *scoped memory areas*. When objects are executing within a scoped memory area, all memory allocations are performed from the scoped memory. When there are no objects active inside an area, the allocated memory is reclaimed. Each scoped memory area is allocated with a maximum size.

To use scoped memory efficiently and safely, the programmers must determine upper bounds on the heap-space requirements of the threads, so that they can specify an accurate maximum size of the scoped memory area. The bounds must be tight enough to enable the efficient use of the limited resources of the embedded device and, more importantly, it must be ensured that the code will be executed without running out of memory.

1.2 Java-like language

Our target language is a subset of Java, that we call FJEU and that is similar to Featherweight Java [IPW99]. It contains object-oriented features such as classes, objects, inheritance and imperative update. It does not contain loops, but only recursion. This is not a major restriction, since loops can be transformed into recursive methods easily. Also, the language does not contain other advanced features of the Java language such as exceptions or threads.

We assume a simple *freelist* based model where we maintain a set of free memory units, the freelist. When creating an object, a heap unit required to store it is taken from the freelist, provided it contains enough units. When deallocating an object, the unit returns to the freelist. We deallocate objects explicitly by means of a free() expression. This freelist-based model is an abstraction from concrete memory models, which we consider convenient for

keeping the theory simple. We believe that extending the analysis to deal with concrete memory models such as the scoped memory from RTSJ would be straightforward.

Using a special free() expression to recover each heap unit that is no longer required is an over-approximation for the recovery that can be performed by the garbage collector. We do not treat in this thesis the problem of automatically inserting free() expressions, since this problem is orthogonal to the problem of resource analysis, which is the main focus of this thesis. There are, however, various works that tackle this problem, which we could integrate in our analysis. For instance, Chin et al. [CNQR05] handle the automatic insertion of free() expressions using an alias type system.

1.3 Method

The need for resource prediction has been widely recognised and there has been considerable progress in the last years. Several approaches to resource analysis have been proposed, such as approaches based on recurrence solving [Weg75, US09, AAG+12], on abstract interpretation [FHL+01, GL02, GMC09] and on amortised analysis [HJ03, HJ06, Cam08, HR09, JLHH10, Atk11, HAH11].

Many of the proposed techniques work under the assumption that control-flow is determined by some numeric parameters such as size of input or linear arithmetic functions thereof. Other dependencies of the control flow are over-approximated by simply taking the maximum over all possible runs. This works well for programs that use arrays that are allocated at the beginning with a given size and processed with for loops with a simple iteration pattern. This is very useful in embedded systems or scientific computing where most programs have such a shape. However, this does not work well with object-oriented programs where resource behaviour depends on the dynamic class tags of objects.

The method of amortised analysis [Tar85, Oka98] is particularly effective in those cases. Therein, data structures are assigned nonnegative numbers, called *potential*. Then, it is possible to compute bounds on the "amortised cost" of an individual operation; that is, its actual resource usage plus the difference in potential of the data structure before and after performing the operation. This makes it possible to take into account the effect that an operation might have on the resource usage of subsequent operations. When amortised analysis is used for automatic resource analysis, we use refined types to define the potentials; typing rules then ensure that potential and actual resource usage is accounted for correctly. Type inference then makes possible an automatic inference of the potential functions.

In amortised resource analysis for functional programs the data structures are known (e.g. lists or trees) and only the potential functions must be

found. Thus, the type inference can be reduced to solving linear arithmetic constraints. In the object-oriented case even the data structures must be discovered by the analysis because objects can be used for just anything be it lists, trees, graphs, etc. As a result, automatic inference becomes considerably more challenging unless one is willing to accept user annotations specifying the way in which objects are to be used, for instance in the form of separation logic annotations [HQLC09, Atk11].

In this thesis we show that it is possible to analyse the resource consumption of object-oriented programs using the amortised analysis technique and type systems without requiring user annotations.

1.4 Contributions

We build upon a system of refinement types for amortised analysis for FJEU programs called Resource Aware JAva (RAJA) which was first described by Hofmann and Jost [HJ06]. This is a powerful type system that can capture the heap-space requirements of many programs.

In this type system, the FJEU classes are refined with *views*, which are infinite trees labelled with real numbers, that can have an arbitrarily complicated structure. This is necessary to model the general structure of objects. Notice that first-order functional programs can be embedded into FJEU programs, if one models the inductive data types as objects using the Composite pattern.

Moreover, the type system takes aliasing into account, which means that the resource analysis based on this system is sound for all programs, even if they contain shared or even cyclic data structures.

Due to the complexity of this type system, the problem of type inference has been open for various years. The main contribution of this thesis is a *type inference algorithm* for a modified version of this system. The type inference comprises three steps:

1. Generating subtyping and arithmetic constraints from a program.

2. Reducing the subtyping constraints to constraints over infinite trees.

3. Solving the constraints over infinite trees.

This thesis also makes the following contributions:

1. We developed *a system of constraint-based types for methods* that represent the method's heap-space requirements. Introducing these polymorphic method types allows for a modular resource analysis, because each method can be analysed independently of its callers. In particular, if the method type is saved after the analysis, that method does

not need to be analysed again if new methods are added to the program, except for the case when the method is redefined in a subclass. This is discussed in detail in Chapter 5.

2. We developed *a type checking algorithm* for an annotated version of the RAJA system, where the methods are given a finite set of concrete RAJA types (not polymorphic types). Those concrete types can be seen as certificates of the resource usage of the program and the type checking algorithm helps to verify that the given resource bounds are correct, independently of the type inference algorithm. We describe this algorithm in Chapter 4.

3. We described the novel *problem of satisfiability of arithmetic constraints over infinite trees*. Although we could not settle the question whether the problem is decidable in general, we proposed a *heuristic algorithm for solving the constraints* when they admit solutions that are regular trees, and proved its soundness. This is described in Chapter 6. Notice that, because we can solve only a restricted class of constraints, our type inference algorithm can only compute linear bounds for programs and can not analyse programs whose heap-space consumption is a non-linear function on the size of its arguments.

4. We described *an algorithm for eliminating variables from constraints over infinite trees*, while preserving the satisfiability of the constraints. The algorithm cannot eliminate all variables, but it can eliminate most variables, which is useful for improving the efficiency of the analysis. This is also described in Chapter 6.

5. We validated the usability of our approach with *an implementation and experimental evaluation*. The experiments validate both the type system and the heuristic algorithm for solving the constraints over infinite trees. We were able to compute precise linear bounds on many programs, including copying and sorting linked-lists, appending lists, creating doubly-linked lists and converting a list of strings to a list of bank accounts. The experiments also show the efficiency of our analysis, and how the use of the elimination procedure improves the efficiency drastically. The implementation and the analysed FJEU programs are available online[1]; we described them in Chapter 7.

1.5 Outline

This thesis is organised as follows:

Chapter 2 describes the syntax and semantics of the language FJEU, including resource aware semantics. Further, it describes the basic typing

[1] http://raja.tcs.ifi.lmu.de/download

rules for FJEU programs, that are similar to the typing rules for Featherweight Java. It also shows a type reconstruction algorithm that is able to reconstruct the type of variables in let expressions in most cases. The type inference algorithm then assumes that the type of all variables in let expressions is available, which simplifies the type inference.

Chapter 3 presents the type system RAJA, a refinement of FJEU's type system. It gives an informal presentation of the system, and shows its use in various examples. Then, it describes the type system formally, and develops the soundness proof, which is basically the soundness proof for the original RAJA system [HJ06] with some minor modifications. Chapter 2 and 3 do not contain many novel results; the details of the proofs have been included for completeness.

Chapter 4 describes the type system *RAJA with explicit types* that is equivalent to the RAJA system, but that is more suitable for type checking and type inference. A RAJA program with explicit types assigns not only a polymorphic type to each method, but also a (possibly infinite) set of concrete types, which is the set of instances of the polymorphic type. Moreover, some FJEU expressions are annotated with refined types, in contrast to the RAJA system, where the expressions have no annotations. The most important feature of this system, however, is the fact that the typing rules are syntax directed. This allows for automatic type-checking. We show that a RAJA program is well typed iff a RAJA program with explicit types is well-typed.

Chapter 5 develops a type inference algorithm for the system $RAJA^m$, a subset of the system RAJA with explicit types; that is, a system where polymorphic recursion is not allowed. We decided to exclude polymorphic recursion to simplify the type inference algorithm. We define constraint generation rules that are sound and complete with respect to the typing rules of $\mathrm{RAJA}^{\mathrm{m}}$. The constraints are generated on the basis of the call graph of the program, and are solved using the algorithm described in Chapter 6.

Chapter 6 develops a theory of infinite trees and presents the syntax and meaning of the constraints over the trees. Then, it shows various algorithmic problems regarding the constraints, such as satisfiability and optimisation. It describes an algorithm for eliminating variables from constraints, and shows its soundness, completeness and termination, in the cases when the algorithm can eliminate the variables. Moreover, it presents a heuristic algorithm for solving the constraints, and proves its soundness. It also detects a subclass of constraints for which the algorithm is complete. Finally, it shows a reduction from subtyping constraints to constraints over infinite trees.

Chapter 7 describes a prototype implementation of the algorithms described in the thesis. The implementation consists of an interpreter for FJEU programs with embedded heap-space analysis. The tool creates an optimised heap, whose size is defined by the parameters given by the analy-

sis. The soundness of the RAJA system guarantees that the programs will not run out of memory when using a heap of that size. In this chapter we describe implementation details of the interpreter, and of the analyser module in particular, including various optimisations. Then, we present an experimental evaluation of the tool. We describe the analysed FJEU programs and discuss the results of the evaluation.

Chapter 8 gives an overview of the research related to this thesis. In particular, it reviews works on resource analysis, on refinement types and on linear types and types-and-capabilities systems.

Chapter 9 discusses the results of the thesis and shows various directions for further research.

1.6 Relation to published work

Various contributions of this thesis have been presented in international conferences.

- The type system RAJA and its soundness proof have been presented in the European Symposium on Programming 2006 by Hofmann and Jost [HJ06]. (*Chapter 2 and 3*)

- Later, Hofmann, Jost and Rodriguez have presented the complete soundness proof for the RAJA system in a technical report [HJR]. (*Chapter 2 and 3*)

  The main differences between the system presented in [HJ06, HJR] and the system presented in this thesis are now given. We introduced polymorphic types, we modified the definition of subtyping and sharing, and we defined the views in a different way. All these changes allowed for a more efficient type inference algorithm, but had little impact in the soundness proof.

- Hofmann and Rodriguez have presented a type checking algorithm for RAJA in the 18th EACSL Annual Conference on Computer Science Logic [HR09]. This algorithm is similar to the algorithm from Chapter 4. However, in this thesis we present a simpler algorithm that requires more annotations, because we also present type inference and so the users do not need to provide the annotations.

- Hofmann and Rodriguez have presented the problem of satisfiability of constraints over infinite trees, the heuristic algorithm and the elimination procedure in the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning [HR12]. These results can be found in Chapter 6 in more detail. For instance, we present a proof of termination for the elimination procedure, which does not appear in [HR12].

Chapter 2

FJ with Update

2.1 Overview

Our formal model of Java, Featherweight Java with Update (FJEU), was first defined by Hofmann and Jost [HJ06]. It extends Featherweight Java(FJ) [IPW99] with attribute update, conditional, null pointers and explicit deallocation. It is thus similar to Flatt *et al.* Classic Java [FKF98].

The typing rules of FJEU are very similar to those of FJ but the semantics is considerably different since it models a mutable heap. While FJ is purely functional, FJEU is imperative. FJEU contains the basic important features of the Java language such as objects, inheritance, side effects and aliasing. It was defined with the main purpose of analysing the resource consumption of programs. Hence it does not include advanced features such as exceptions or blocks, since they would complicate the proofs significantly and do not offer major challenges to the resource analysis.

Because of its simplicity and similarity to Java, FJEU has been the target language of choice in various works on static analysis of object-oriented programs. In particular, Kersten presented in his Master Thesis [Ker09] a system of sized types for determination of polynomial upper-bounds for heap-space consumption of FJEU programs. Beringer *et al.* provided in [BGH10] a region type system for pointer and string analyses for FJEU programs. Later, Grabowski extended FJEU with strings in [Gra11] and defined a type system for ensuring programming guidelines for preventing code injection.

There has been a number of proposals for small imperative subsets of Java such as Mølhave and Petersen's AFJ [lP05], which extends FJ with an update statement and defines the semantics using generalised labelled transition systems. Biermann *et al.* define MJ, an imperative core of Java that contains constructor methods, block structure and field update in addition to FJ. They model the operational semantics of MJ as transitions between configurations where a configuration is a heap, variable stack, term

and frame stack for modelling the block structure scoping.

An FJEU program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ is a partial finite map from class names to class definitions, which we also refer to as *class table*, where main is the method that will be executed when running $\mathcal{P}$. Here is a simple program in FJEU. We shall see more FJEU programs in the following chapter.

```
class Person  {
  String name;
  String address;

}
class Student extends Person {
  String matriculationnr;
}
class Main {

  Student main() {
    let s = new Student in
    let s = s.name <- ``Dulma Rodriguez'' in
    let s = s.address <- ``Oettingenstr. 67'' in
    let s = s.matriculationnr <- ``14232564'' in
      return s;
  }
}
```

Notation We write $\mathcal{D} :: \Gamma \vdash e : C$ for meaning that $\mathcal{D}$ is the type derivation that shows that e has type C in the context Γ. We write fg for meaning the union of the maps f and g when $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$. We write $f|_{A'}$ where $f : A \to B$ and $A' \subseteq A$ for the function f restricted to the domain A'.

2.2 Syntax and typing

2.2.1 Syntax

Throughout the following sections we will consider a fixed (but arbitrary) class table $\mathscr{C}$ for the ease of notation. The abstract syntax of FJEU is given in Fig. 2.1. The metavariables C, D, E, F, G, H range over class names; a, b range over field names; m ranges over method names; x, y, z ranges over variables and e ranges over expressions. We write $\vec{A}$ as a shorthand for a sequence of field declarations $A_1 \ldots A_n$ and $\vec{M}$ as a shorthand for a sequence of method declarations $M_1 \ldots M_m$. Moreover we write $\vec{x}$ as a shorthand for $x_1, \ldots, x_n$.

$$
\begin{array}{rll}
c ::= & \texttt{class}\ C\ [\texttt{extends}\ D]\ \{\vec{A}; \vec{M}\} & \text{(Class)} \\
A ::= & C\ a & \text{(Attribute)} \\
M ::= & H\ m(E_1\ x_1, \ldots, E_j\ x_j)\{\texttt{return}\ e;\} & \text{(Method)} \\
e ::= & x & \text{(Variable)} \\
\mid & \mathsf{null} & \text{(Constant)} \\
\mid & \texttt{new}\ C & \text{(Construction)} \\
\mid & \texttt{free}(x) & \text{(Destruction)} \\
\mid & (C)x & \text{(Cast)} \\
\mid & x.a_i & \text{(Access)} \\
\mid & x.a_i\texttt{<-}x & \text{(Update)} \\
\mid & x.m(\vec{x}) & \text{(Invocation)} \\
\mid & \texttt{if}\ x\ \texttt{instanceof}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \text{(Conditional)} \\
\mid & \mathsf{let}\ [D]\, x = e_1\ \mathsf{in}\ e_2 & \text{(Let)}
\end{array}
$$

Figure 2.1: The syntax of FJEU.

The class declaration $\texttt{class}\ C\,[\texttt{extends}\ \ D]\,\{\vec{A}; \vec{M}\}$ introduces a class named C with an optional *super-class* D and fields A_1 to A_n and methods M_1 to M_m.

We write $\mathsf{S}(C)$ to denote the super-class D, provided that C has a super-class. We write $\mathsf{A}(C)$ to denote the ordered set of attributes of C, including inherited ones, i.e. $\mathsf{A}(C) := \{A_1, \ldots, A_n\} \cup \mathsf{A}(D)$. As in FJ, we do not allow redeclaration of fields in subclasses. We define $\mathsf{A}(\mathsf{S}(C)) := \emptyset$ in the case that C has no super-class. We write $C.a_i$ to denote the class type of each attribute a_i of class C.

We write $\mathsf{Meth}(C)$ to denote the set of all defined method names of C, including inherited ones, i.e. $\mathsf{Meth}(C) = \{M_1, \ldots, M_m\} \cup \mathsf{Meth}(D)$. The method declaration $H\ m(E_1\ x_1, \ldots, E_j\ x_j)\{\texttt{return}\ e;\}$ defines a method named m with parameters x_i of type E_i and result type H. The *method body* is the expression e. For a method m of class C we write $\mathsf{M}_{\mathsf{body}}(C, m)$ to denote the term that comprises the method body of method m, i.e. $\mathsf{M}_{\mathsf{body}}(C, m) = e$. We write $C.m$ to denote the *method type* of m in class C, i.e. $C.m = E_1, \ldots, E_j \rightarrow H$. If otherwise m is not defined in C, then $\mathsf{M}_{\mathsf{body}}(C, m) = \mathsf{M}_{\mathsf{body}}(D, m)$ and $C.m = D.m$, provided that D is the super class of C. In the case that C has no super-class, we also define $\mathsf{Meth}(\mathsf{S}(C)) := \emptyset$ for the sake of a uniform notation.

Each class has only one implicit constructor, which sets all class attributes to a nil value.

The expression $\texttt{free}(x)$ for explicit object deallocation does not occur in Java, it is rather similar to the expression $\texttt{free}(x)$ from C or $\mathsf{delete}(x)$ from C++. Java has built-in garbage collection to handle memory deallocation. We added explicit deallocation to FJEU nevertheless because we plan to analyse the heap-space requirements of FJEU programs. Our long-term goal

is to introduce the `free`(x) expressions automatically, in the places where the Java garbage collector would free the space for the object x. For doing this we need to predict the behaviour of the garbage collector. However, we do not treat that problem in this thesis since here we focus on resource analysis.

Please note that the rule for field update differs slightly from Java: Evaluating the term $x.a\texttt{<-}y$ has the side effect of updating the field a of object x with the value y. However, the whole term itself does not evaluate to the right-hand value y as in Java, but rather to the left-hand value x. Of course, we can define the Java style update as $\mathsf{let}\ u = (x.a\texttt{<-}y)\ \mathsf{in}\ y$. The reason for choosing the other primitive is that the proof of type soundness becomes slightly easier and also that it is more natural from a resource-oriented point of view since the updated object (x) should still be available whereas the update value (y) should be considered "consumed" unless explicitly duplicated. This will become clearer in Chapter 3.

The expressions are in let-normal form which means that all the intermediate expressions are named and there are no nested expressions such as $x.a\texttt{<-}y.a$. The let-normal form of terms was merely chosen to eliminate boring redundancies from our proofs. In fact, in our implementation of FJEU we allow nested expressions and transform them in let-normal form by a simple preprocessing. Note that we will also write "e_1; e_2" instead of "$\mathsf{let}\ D\,x = e_1\ \mathsf{in}\ e_2$" if x does not appear anywhere else inside the program.

Moreover, we do not only allow a variable to be used more than once in update or invocation expressions. For instance, we do not allow the expression $x.a\texttt{<-}x$ or $x.m(y, x)$. This is not a proper restriction, since we can transform such expressions into let expressions, by creating a copy of the variable to be used twice. For instance, $x.a\texttt{<-}x$ can be transformed to $\mathsf{let}\, y = x\ \mathsf{in}\ x.a\texttt{<-}y$, which is allowed. This syntactic restriction makes our type checking and type inference algorithms slightly simpler.

The type of the variable x in the let-expressions can be declared by the user. However, it can be inferred automatically in most cases by an algorithm that we will show later on. In the following chapters, when we analyse refined types for FJEU programs, we assume that those types are present: they can be either supplied manually by the user or discovered by the type inference algorithm. The reason for this requirement is that the algorithms for type checking and type inference for refinement types become significantly simpler when they do not need to find FJEU types as well.

2.2.2 Typing

We say that C is a subtype of D, and write $C <: D$, when $\mathsf{S}(C) = D$ or $\mathsf{S}(C) <: D$. In other words, $C <: D$ is the reflexive and transitive closure of the subclass relation given by the extends clauses in $\mathscr{C}$. (Fig. 2.2)

Definition 2.2.1 *($C_1 \vee C_2$) The least upper bound $C_1 \vee C_2$ of the classes C_1 and C_2 is defined as the class D with $C_1 <: D$ and $C_2 <: D$ and for all classes E with $C_1 <: E$ and $C_2 <: E$ holds $D <: E$.*

If C_1 and C_2 do not have a common super-class, then $C_1 \vee C_2$ is undefined. This can occur because we do not assume that there is always a class Object with $C <:$ Object for each class C, as is the case in Java and FJ.

An environment Γ is a finite partial mapping from identifiers to class names. The typing judgement of FJEU takes the form $\Gamma \vdash e : C$, read "in the environment Γ, expression e has type C". It is defined as a standard extension of the FJ typing rules (Fig. 2.2), with the difference that our rules are non-syntax-directed. There is one rule for each form of expression and an additional rule for subtyping, in the style of Classic Java. Moreover, we introduce the rule ($\vdash_{\mathsf{F}} \mathit{Duplicate}$) for using variables more than once. The reason for this design choice will become clear in Chapter 3 when we define refined types for FJEU. There, the rule ($\Diamond \mathit{Share}$) extends ($\vdash_{\mathsf{F}} \mathit{Duplicate}$) with refined types. However, with ($\Diamond \mathit{Share}$) a variable can only be duplicated if there are refined types for each occurrence that are in the so called *sharing* relation. This control mechanism for variable duplication is essential for the accurate analysis of resource consumption of programs.

For two environments Γ and Θ we write $\Gamma <: \Theta$ for meaning $\forall x \in \Theta \,.\, \Gamma(x) <: \Theta(x)$.

The expressions null and $\texttt{free}(x)$ can have any type E. While this is the standard typing for null expressions in Java or Classic Java, it is less obvious why $\texttt{free}(x)$ can have any type, when its main purpose is deallocating a cell from the heap. A more sensible type for it would be void, but we did not add the type void to FJEU for reasons of simplicity. The typing of $\texttt{free}(x)$ can nevertheless be justified by its semantics: as we will see later, a $\texttt{free}(x)$ expression evaluates to a null pointer.

The typing judgement for method declarations has the form $\vdash m : C.m\,\mathsf{ok}$, read "the method declaration m is well-typed with type $C.m$". A method is well-typed if its body can be typed with its result type, where the free variables are the parameters of the method with their declared types plus the variable this with type C.

Finally, an FJEU program is well-typed if the methods of all classes are well-typed.

Definition 2.2.2 (Well-typed FJEU-program)
An FJEU-program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ is well-typed *if the following condition is satisfied: $\forall C \in \mathscr{C}, m \in \mathsf{Meth}(C)\,.\ \vdash m : C.m\ \mathsf{ok}$.*

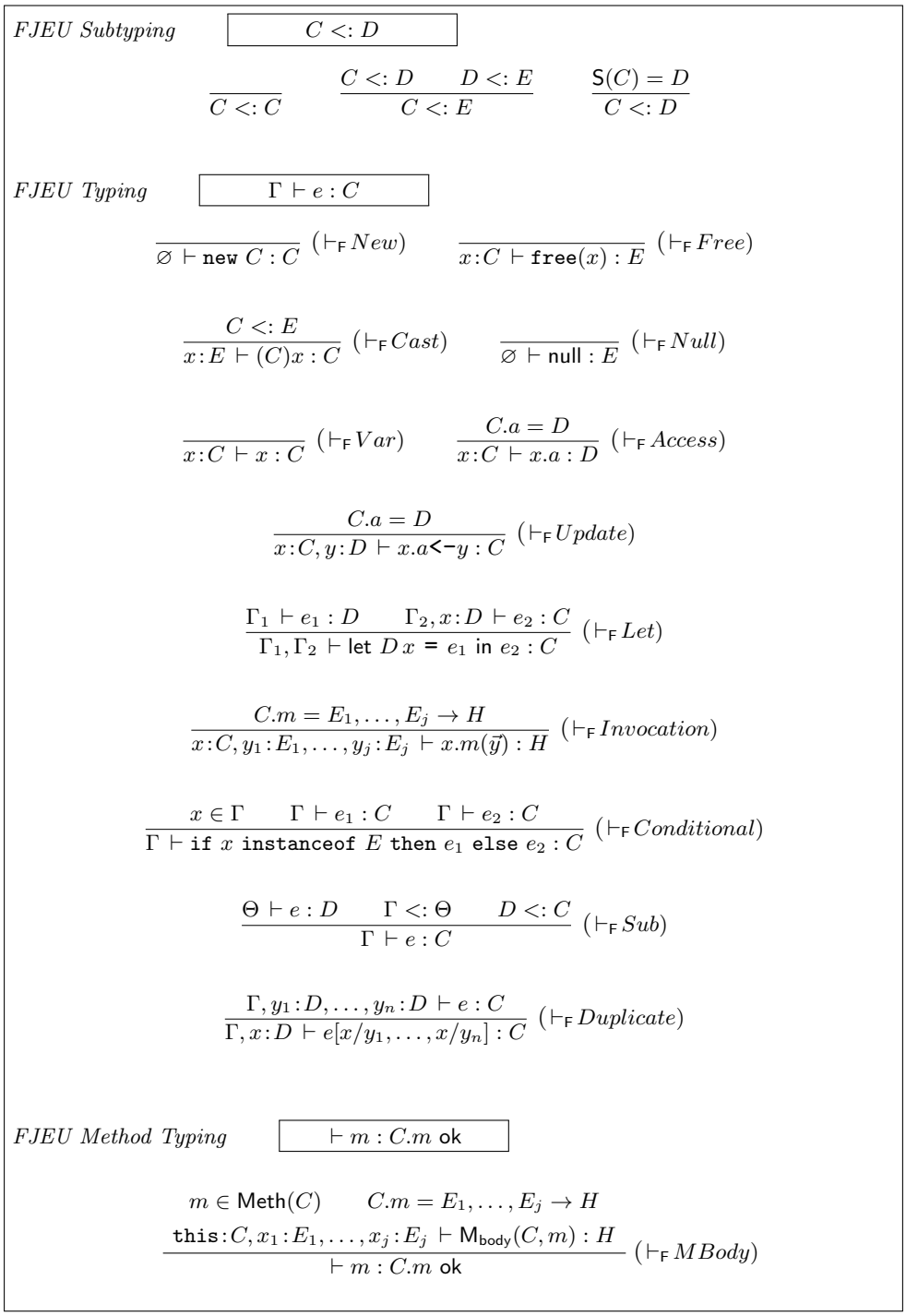


Figure 2.2: FJEU subtyping, FJEU typing and FJEU method typing.

2.2.3 Type reconstruction

In this section we show an (incomplete) algorithm for reconstructing the type of the variable x in let-expressions when it has been omitted by the user. In Fig. 2.3 we define the judgement $\Gamma \vdash e^- : (e, C)$, read "in the environment Γ, the expression e^- with missing type annotations has type C and can be annotated as e". We also define the related judgement $\Gamma \vdash e^- : \mathsf{any\ type}$, read "in the environment Γ, the expression e^- can have any type". This judgement models the cases where we are not able to infer a type, e.g. for null and $\texttt{free}(x)$ expressions, but also for conditional expressions in some cases.

The main goal of the algorithm is to reconstruct the type for x in expressions $\mathsf{let}\ x = e_1^-\ \mathsf{in}\ e_2^-$. This is possible if we can infer a type for e_1^-. There are five nested cases for handling let-expressions.

1. There is no type for x.
   (a) It is possible to infer a type for e_1^-. This case is dealt with by the rule ($\vdash_{\mathsf{R}} Let$) where an annotated expression is returned.
   (b) It is not possible to infer a type for e_1^-.
      i. x does not appear in e_2^-. Then, an arbitrary but fixed type D is given to x in the rule ($\vdash_{\mathsf{R}} Let\ Fail$).
      ii. x appears in e_2^-. There is no rule for this case, so the algorithm fails. This means that in this case the user needs to provide a type annotation for x.
2. There is a type for x.
   (a) It is possible to infer a type for e_1^-. Then we check that the inferred type is a subtype of the declared type in the rule ($\vdash_{\mathsf{R}}$ *Let Ann*).
   (b) It is not possible to infer a type for e_1^-. Then an annotated expression is returned with the declared type in the rule ($\vdash_{\mathsf{R}}$ *Let Fail Ann*).

For the conditional expression $\mathsf{if}\ x\ \mathsf{instanceof}\ C\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ it is possible to infer a type, if a type can be inferred for e_1 or e_2. If a type can be inferred for only one of them, then that will be the type of the whole expression. If, otherwise, a type can be inferred for both expressions, the type returned will be the least upper bound of those types. Finally, if no type can be inferred for e_1 or e_2, then we can not infer a type for the conditional expression.

The remaining rules are closely related to the typing rules. Since the type inference rules should be syntax-directed, the rules ($\vdash_{\mathsf{F}} Sub$) and ($\vdash_{\mathsf{F}}$ *Duplicate*) have been integrated in the other rules. We obtain the following soundness result:

$$\frac{}{\Gamma \vdash \mathsf{null} : \mathsf{any\ type}}\ (\vdash_{\mathsf{R}} Null) \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{free}(x) : \mathsf{any\ type}}\ (\vdash_{\mathsf{R}} Free)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash e_1^- : \mathsf{any\ type} \qquad \Gamma \vdash e_2^- : \mathsf{any\ type}}{\Gamma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2^- : \mathsf{any\ type})}\ (\vdash_{\mathsf{R}} Cond\ Fail)$$

$$\frac{\Gamma(x) = C}{\Gamma \vdash x : (x, C)}\ (\vdash_{\mathsf{R}} Var) \qquad \frac{\Gamma(x) = E \qquad C <: E}{\Gamma \vdash (C)x : ((C)x, C)}\ (\vdash_{\mathsf{R}} Cast)$$

$$\frac{}{\Gamma \vdash \mathtt{new}\ C : (\mathtt{new}\ C, C)}\ (\vdash_{\mathsf{R}} New) \qquad \frac{\Gamma(x) = C \qquad C.a = D}{\Gamma \vdash x.a : (x.a, D)}\ (\vdash_{\mathsf{R}} Access)$$

$$\frac{\Gamma(x) = C \qquad C.a = D \qquad \Gamma(y) <: D}{\Gamma \vdash x.a\mathtt{<-}y : (x.a\mathtt{<-}y, C)}\ (\vdash_{\mathsf{R}} Update)$$

$$\frac{\Gamma(x) = C \qquad C.m = \vec{E} \to H \quad \Gamma(y_i) <: E_i}{\Gamma \vdash x.m(\vec{y}) : (x.m(\vec{y}), H)}\ (\vdash_{\mathsf{R}} Invocation)$$

$$\frac{\Gamma \vdash e_1^- : (e_1, D) \qquad \Gamma, x{:}D \vdash e_2^- : (e_2, C)}{\Gamma \vdash \mathsf{let}\ \ x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2^- : (\mathsf{let}\ \ D\,x\ \mathsf{=}\ e_1\ \mathsf{in}\ e_2, C)}\ (\vdash_{\mathsf{R}} Let)$$

$$\frac{\Gamma \vdash e_1^- : \mathsf{any\ type} \quad x \notin \mathsf{Vars}(e_2) \quad D \in \mathcal{C} \quad \Gamma, x{:}D \vdash e_2^- : (e_2, C)}{\Gamma \vdash \mathsf{let}\ \ x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2^- : (\mathsf{let}\ \ D\,x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2, C)}\ (\vdash_{\mathsf{R}} Let\ Fail)$$

$$\frac{\Gamma \vdash e_1^- : \mathsf{any\ type} \qquad \Gamma, x{:}D \vdash e_2^- : (e_2, C)}{\Gamma \vdash \mathsf{let}\ \ D\,x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2^- : (\mathsf{let}\ \ D\,x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2, C)}\ (\vdash_{\mathsf{R}} Let\ Fail\ Ann)$$

$$\frac{\Gamma \vdash e_1^- : (e_1, E) \qquad \Gamma, x{:}D \vdash e_2^- : (e_2, C) \qquad E <: D}{\Gamma \vdash \mathsf{let}\ D\,x\ \mathsf{=}\ e_1^-\ \mathsf{in}\ e_2^- : (\mathsf{let}\ D\,x\ \mathsf{=}\ e_1\ \mathsf{in}\ e_2, C)}\ (\vdash_{\mathsf{R}} Let\ Ann)$$

$$\frac{x \in \Gamma \quad \Gamma \vdash e_1^- : (e_1, C_1) \quad \Gamma \vdash e_2^- : (e_2, C_2) \quad C_1 \vee C_2 \text{ is defined}}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2^- : \\ (\mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, C_1 \vee C_2)\end{array}}\ (\vdash_{\mathsf{R}} Cond\ Lub)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash e_1^- : \mathsf{any\ type} \qquad \Gamma \vdash e_2^- : (e_2, C_2)}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2^- : \\ (\mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2, C_2)\end{array}}\ (\vdash_{\mathsf{R}} Cond_2)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash e_1^- : (e_1, C_1) \qquad \Gamma \vdash e_2^- : \mathsf{any\ type}}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2^- : \\ (\mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2, C_1)\end{array}}\ (\vdash_{\mathsf{R}} Cond_1)$$

Figure 2.3: FJEU type reconstruction.

Lemma 2.2.3 (Soundness of type reconstruction) *Let e^- be an expression with missing annotations, Γ be a context and $C \in \mathscr{C}$.*

1. *If $\mathcal{D} :: \Gamma \vdash e^- : \mathsf{any\ type}$ then $\Gamma \vdash e^- : C$.*
2. *If $\mathcal{D} :: \Gamma \vdash e^- : (e, C)$ then $\Gamma \vdash e : C$.*

Proof.

1. By a trivial induction on the derivation $\mathcal{D}$.
2. By induction on the derivation $\mathcal{D}$. We show some representative cases.

   Case

   $$\frac{\Gamma \vdash e_1^- : \mathsf{any\ type} \qquad \Gamma, x{:}D \vdash e_2^- : (e_2, C)}{\Gamma \vdash \mathsf{let}\ \ D\,x = e_1^-\ \mathsf{in}\ e_2^- : (\mathsf{let}\ \ D\,x = e_1^-\ \mathsf{in}\ e_2, C)}\ (\vdash_{\mathsf{R}} \mathit{Let\ Fail\ Ann})$$

   By 1. we get $\Gamma \vdash e_1^- : D$ and by induction hypothesis we get $\Gamma, x{:}D \vdash e_2 : C$, thus, we can finish with the rules $(\vdash_{\mathsf{F}} \mathit{Let})$ and $(\vdash_{\mathsf{F}} \mathit{Duplicate})$.

   Case

   $$\frac{x \in \Gamma \quad \Gamma \vdash e_1^- : (e_1, C_1) \quad \Gamma \vdash e_2^- : (e_2, C_2) \quad C_1 \vee C_2 \text{ is defined}}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1^-\ \mathtt{else}\ e_2^- : \\ (\mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, C_1 \vee C_2)\end{array}}$$

   By induction hypothesis we obtain $\Gamma \vdash e_1 : C_1$ and $\Gamma \vdash e_2 : C_2$. Then, since $C_i <: C_1 \vee C_2$, we obtain with $(\vdash_{\mathsf{F}} \mathit{Sub})$, $\Gamma \vdash e_1 : C_1 \vee C_2$ and $\Gamma \vdash e_2 : C_1 \vee C_2$ and finish with $(\vdash_{\mathsf{F}} \mathit{Conditional})$.

   □

2.3 Semantics of FJEU

In this section we present operational semantics for the language FJEU. First we define standard operational semantics and later we present slightly modified semantics for analysing the heap-space requirements of FJEU programs.

2.3.1 Standard operational semantics

We assume a set Loc of *locations*, ranged over by the letter ℓ. A *stack value* v is either a location $\ell \in \mathsf{Loc}$ or a special value $0 \notin \mathsf{Loc}$. A *stack* or environment η is a partial mapping from identifiers to stack values.

A *heap value* w is a record consisting of a class name $C \in \mathscr{C}$ and list of labelled stack values, written $(C, a_1 : v_1, \ldots, a_k : v_k)$. A *heap* σ is a partial mapping from locations to heap values. We write $\sigma[\ell.a \mapsto v]$ for

Operational Semantics of FJEU $\boxed{\eta, \sigma \vdash e \rightsquigarrow v, \tau}$

$$\frac{}{\eta, \sigma \vdash x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{S}} Var) \qquad \frac{}{\eta, \sigma \vdash \mathsf{null} \rightsquigarrow 0, \sigma} \ (\vdash_{\mathsf{S}} Null)$$

$$\frac{\ell \notin \mathrm{dom}(\sigma) \qquad \mathsf{A}(C) = \{a_1, \ldots, a_k\} \qquad \tau = \sigma[\ell \mapsto (C, a_1 : 0, \ldots, a_k : 0)]}{\eta, \sigma \vdash \mathtt{new}\ C \rightsquigarrow \ell, \tau} \ (\vdash_{\mathsf{S}} New)$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash \mathtt{free}(x) \rightsquigarrow 0, \sigma[\ell \mapsto (\mathsf{invalid})]} \ (\vdash_{\mathsf{S}} Free)$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (D, a_1 : v_1, \ldots, a_k : v_k) \qquad D <: C}{\eta, \sigma \vdash (C)x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{S}} Cast\,\mathrm{I})$$

$$\frac{\eta_x = 0}{\eta, \sigma \vdash (C)x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{S}} Cast\,\mathrm{II})$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash x.a_i \rightsquigarrow v_i, \sigma} \ (\vdash_{\mathsf{S}} Access)$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C_0, a_1 : v_1, \ldots, a_k : v_k) \qquad a = a_i \qquad \tau = \sigma[\ell.a_i \mapsto \eta_y]}{\eta, \sigma \vdash x.a\mathtt{<-}y \rightsquigarrow \ell, \tau} \ (\vdash_{\mathsf{S}} Update)$$

$$\frac{\eta, \sigma \vdash e_1 \rightsquigarrow v_1, \rho \qquad \eta[x \mapsto v_1], \rho \vdash e_2 \rightsquigarrow v_2, \tau}{\eta, \sigma \vdash \mathsf{let}\ x\ \mathtt{=}\ e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau} \ (\vdash_{\mathsf{S}} Let)$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k) \qquad \mathsf{M}_{\mathsf{body}}(C, m) = e_0 \\ [\mathtt{this} \mapsto \ell, x_1 \mapsto \eta_{y_1}, \ldots, x_j \mapsto \eta_{y_j}], \sigma \vdash e_0 \rightsquigarrow v, \tau\end{array}}{\eta, \sigma \vdash x.m(\vec{y}) \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{S}} Inv)$$

$$\frac{\eta_x = \ell \qquad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \quad D <: E \qquad \eta, \sigma \vdash e_1 \rightsquigarrow v, \tau}{\eta, \sigma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{S}} Cond\,\mathrm{I})$$

$$\frac{\eta_x = \ell \qquad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \qquad D \not<: E \qquad \eta, \sigma \vdash e_2 \rightsquigarrow v, \tau}{\eta, \sigma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{S}} Cond\,\mathrm{II})$$

$$\frac{\eta_x = 0 \qquad \eta, \sigma \vdash e_2 \rightsquigarrow v, \tau}{\eta, \sigma \vdash \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{S}} \text{Cond III})$$

Figure 2.4: Operational semantics of FJEU.

the heap $\sigma\big[\ell \mapsto (C, a_1 : v_1, \ldots, a_i : v, \ldots, a_k : v_k)\big]$, provided that $a_i = a$ and $\sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)$.

The judgement $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ defined by the rules in Fig. 2.4 shall mean that the expression e evaluates successfully to the value v, beginning with stack η, heap σ and ending with heap τ.

One possible way to interpret the presented operational semantics is the assumption of a freelist-based memory model. The freelist contains initially a certain number of locations corresponding to unused heap units. Upon memory allocation the required number of locations is taken from the freelist; upon deallocation a corresponding number of locations of freed heap units are returned to the freelist. Issues of alignment and fragmentation will be ignored here.

In the underlying informal physical model it is perfectly possible that such a freelist occasionally contains locations that are still reachable from the current environment by dangling pointers. Therefore a call to `new` may return a (stale) pointer that is therefore aliased from the beginning. Such an accidental runtime alias may even violate type safety, since a standard type system abstracts away from physical memory addresses. However, existing work may already be used to exclude such faulty behaviour (amongst other beneficial properties), e.g. the alias types by Walker and Morrisett [WM01], or the bunched implication logic as practised by Ishtiaq and O'Hearn [IO01].

Therefore, we choose the modular approach of switching essentially to a storeless semantics [RBR$^+$05, BIL03, Deu94, Jon81], hence allowing us to assume that a call to `new` will always return a fresh pointer and leave it to the reader to choose his or her preferred method to ensure this.

We admit a special heap value (invalid) to mark freed heap values. Hence locations are never removed from the domain of the heap and all allocated locations are thus trivially fresh. Locations pointing to disposed heap values are treated like dangling pointers. Hence pointers to invalid locations are harmless as long as they are never dereferenced, so cyclic data structures can be deallocated. In this context we write $\ell \in \sigma$ by meaning "$\ell \in \mathrm{dom}(\sigma)$ and $\sigma(\ell) \neq (\mathsf{invalid})$". We explicitly write $\ell \in \mathrm{dom}(\sigma)$ otherwise. Notice that any attempt to deallocate a previously deallocated object leads to immediate abortion of the program since in the rule ($\vdash_{\mathsf{S}} Free$) we assume $\ell \in \sigma$.

One may note that one way of directly modelling these semantics lies in the use of indirect pointers (symbolic handles) used by earlier implementations of the Sun JVM for the compacting garbage collector.

Soundness of the heap

Intuitively, a heap is sound when each object record contains exactly the attribute labels as specified in the class table and furthermore, for each alias, its actual class is a subtype of the static class as derivable from the typing context. In the following we define the concepts of access path and

alias formally for being able to formalise this definition of soundness of a heap.

Definition 2.3.1 (Access path) *An* access path *is a list of attribute names, written* $a_1.a_2.\ldots.a_n = \vec{p}$.

It is convenient to write $\mathsf{A}(C, \vec{p})$ for the class type reached by following the access path $\vec{p}$, i.e. $\mathsf{A}(C, \vec{p}.a) = \mathsf{A}(\mathsf{A}(C, \vec{p}), a)$. Sometimes we will abbreviate $\mathsf{A}(C, \vec{p})$ by writing $C.\vec{p}$. The empty access path is denoted by ε.

Definition 2.3.2 (Prefix of a path) *An access path* $\vec{q}$ *is a* proper prefix *of a path* $\vec{p}$*, written* $\vec{q} \prec \vec{p}$*, if and only if there is a non-empty access path* $\vec{a}$ *such that* $\vec{q}.\vec{a} = \vec{p}$*. Moreover, an access path* $\vec{q}$ *is a* prefix *of a path* $\vec{p}$*, written* $\vec{q} \preceq \vec{p}$ *if* $\vec{q} \prec \vec{p}$ *or* $\vec{q} = \vec{p}$.

If $\mathcal{D}$ is a multiset of classes, we simply write $\mathcal{D}.\vec{p}$ for the multiset $\{D.\vec{p} \mid D \in \mathcal{D}\}$. We also use the Kleene star for the set of paths possibly containing repetitions, i.e. $a.b.c.b.c.e \in a(.b.c)^*.e$. We allow ourselves to omit the dots in access path expressions.

Definition 2.3.3 (Alias) *An* alias $v.\vec{p}$ *is a pair of a stack value* v *and a (possibly empty) access path* $\vec{p}$*. For a given heap* σ *we recursively define* $[\![v.\vec{p}]\!]_\sigma$ *by*

$$[\![v.\vec{p}]\!]_\sigma = \begin{cases} v & \text{if } \vec{p} = \varepsilon \\ v_i & \text{if } \vec{p} = \vec{q}.a_i \wedge \sigma_{[\![v.\vec{q}]\!]_\sigma} = (C, a_1 : v_1, \ldots, a_k : v_k) \\ 0 & \text{otherwise} \end{cases}$$

We say that $v.\vec{p}$ *is an* alias for the location ℓ within σ *if* $[\![v.\vec{p}]\!]_\sigma = \ell$ *holds, i.e. the alias represents a valid path leading to a proper location.*

In order to relate an FJEU typing to a heap configuration, we define:

Definition 2.3.4 (Dynamic and static type of an alias)

$$\begin{aligned} \llbracket v.\vec{p} \rrbracket^{\mathrm{dyn}}_\sigma &= C \quad \text{iff} \quad \sigma([\![v.\vec{p}]\!]_\sigma) = (C, a_1 : v_1, \ldots, a_k : v_k) \\ \llbracket (v:C).\vec{p} \rrbracket^{\mathrm{stat}}_\sigma &= \begin{cases} C & \text{if } \vec{p} = \varepsilon \\ D.a & \text{if } \vec{p} = \vec{q}.a \ \wedge \ \llbracket v.\vec{q} \rrbracket^{\mathrm{dyn}}_\sigma = D \end{cases} \end{aligned}$$

Note that $\llbracket v.\vec{q} \rrbracket^{\mathrm{dyn}}_\sigma$ and $\llbracket (v : C).\vec{q}.a \rrbracket^{\mathrm{stat}}_\sigma$ are only defined if $[\![v.\vec{q}]\!]_\sigma$ is defined and within the domain of σ.

Lemma 2.3.5 *Let σ be a heap, v be a value and $\vec{p}$ an access path and $D <: C$. Then $[\![(v:D).\vec{p}]\!]^{\mathrm{stat}}_{\sigma} <: [\![(v:C).\vec{p}]\!]^{\mathrm{stat}}_{\sigma}$.*

Proof. By induction on $|\vec{p}|$. □

Definition 2.3.6 (Sound Heap) *We write $\sigma \vDash \eta : \Gamma$ to say that a memory configuration consisting of heap σ and stack η is sound and satisfies an FJEU typing context Γ, if*

$$\mathrm{ran}(\eta) \subseteq \mathrm{dom}(\sigma) \cup \{\mathsf{0}\} \tag{2.3.1}$$

$$\forall \ell \in \mathrm{dom}(\sigma)\,.\,\sigma(\ell) = (C, a_1 : v_1, \ldots, a_k : v_k) \implies \forall i \in \{1,\ldots,k\}\,.\,v_i \in \mathrm{dom}(\sigma) \cup \{\mathsf{0}\} \wedge \mathsf{A}(C) = \{a_1,\ldots,a_k\} \tag{2.3.2}$$

$$\forall x \in \mathrm{dom}(\eta)\,.\,\eta_x.\vec{p} \in \sigma\,.\,[\![\eta_x.\vec{p}]\!]^{\mathrm{dyn}}_{\sigma} <: [\![(\eta_x:\Gamma_x).\vec{p}]\!]^{\mathrm{stat}}_{\sigma} \tag{2.3.3}$$

We write $\sigma \vDash v : C$ as abbreviation for $\sigma \vDash [x \mapsto v] : x : C$. Klein and Nipkow [KN04] and other authors define the equivalent of our relation $\sigma \vDash v : C$ in a slightly different manner, namely they require (in our notation) that if $\sigma(v) = (D, \ldots)$ then $D <: C$ and whenever $\sigma(\ell) = (C, a_1 : v_1, \ldots, a_k : v_k)$ and $\sigma(v_i) = (E, \ldots)$ then $E <: C.a_i$.

Our notion is equivalent to Nipkow's when restricted to the reachable fragment of σ. We prefer our path-oriented version for two reasons: first, it allows us to give a precise meaning to static types of heap locations which, as is easily seen, depend on the access path leading up to the location. Furthermore, this definition generalises more smoothly to the annotated version RAJA of FJEU to be defined in Chapter 3. Indeed, in RAJA the static type will depend on the entire access path and not only on its last hop as is the case in FJEU. It is interesting to note that some of our constructs involving access paths are actually quite similar to constructs used in [BIL03] which deal with a proper storeless semantics.

Lemma 2.3.7 (Compatibility with subtyping) *Let $\Gamma <: \Theta$ and σ be a heap and η be a stack. If $\sigma \vDash \eta : \Gamma$ then also $\sigma \vDash \eta : \Theta$.*

Proof. We only need to show item (2.3.3) from Definition 2.3.6. We notice that $[\![\eta_x.\vec{p}]\!]^{\mathrm{dyn}}_{\sigma} <: [\![(\eta_x : \Gamma_x).\vec{p}]\!]^{\mathrm{stat}}_{\sigma}$ by assumption and $[\![(\eta_x : \Gamma_x).\vec{p}]\!]^{\mathrm{stat}}_{\sigma} <: [\![(\eta_x:\Theta_x).\vec{p}]\!]^{\mathrm{stat}}_{\sigma}$ by Lemma 2.3.5. Finally, the goal follows by transitivity. □

Lemma 2.3.8 *Let Γ be a context, $D, C_1, \ldots, C_j \in \mathscr{C}$ and σ be a heap and η be a stack. Then*

1. *If $\sigma \vDash \eta : (\Gamma, y : D)$ then also $\sigma \vDash \eta' : (\Gamma, x_1 : D, \ldots, x_j : D)$ where $\eta' = \eta[x_1 \mapsto \eta_y, \ldots, x_j \mapsto \eta_y]$.*
2. *If $\sigma \vDash \eta : \Gamma$ and $\Gamma = x_1 : C_1, \ldots, x_j : C_j$ then also $\sigma \vDash \eta' : (\Gamma, \hat{x}_1 : C_1, \ldots, \hat{x}_j : C_j)$ where $\eta' = \eta[\hat{x}_1 \mapsto \eta_{x_1}, \ldots, \hat{x}_j \mapsto \eta_{x_j}]$.*

Proof. In both cases the goal follows by $\mathrm{ran}(\eta') = \mathrm{ran}(\eta)$ and by assumption. □

Lemma 2.3.9 *If $\mathcal{D} :: \eta, \sigma \vdash e \rightsquigarrow v, \tau$ and $\mathrm{dom}(\eta) \cap \mathrm{dom}(\eta') = \emptyset$ then also $\eta\eta', \sigma \vdash e \rightsquigarrow v, \tau$.*

Proof. By induction on $\mathcal{D}$. □

Theorem 2.3.10 (Soundness of FJEU typing) *Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be a well-typed FJEU program, e be an expression, Γ, Δ be a context and $C \in \mathscr{C}$. Moreover let σ, τ be heaps and η be a stack. If*

$$\mathcal{D} :: \Gamma \vdash e : C \tag{2.3.4}$$

$$\mathcal{E} :: \eta, \sigma \vdash e \rightsquigarrow v, \tau \tag{2.3.5}$$

$$\sigma \vDash \eta : (\Gamma, \Delta) \tag{2.3.6}$$

then

$$\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C) \tag{2.3.7}$$

where x_{res} is assumed to be an unused auxiliary variable, i.e. $x_{res} \notin \Gamma, \Delta$.

Proof. The proof is by induction on the derivation $\mathcal{E}$ and a subordinate induction on the derivation $\mathcal{D}$.

Case ($\vdash_{\mathsf{F}} Sub$) Assume that $\mathcal{D}$ was established in the last step by application of rule ($\vdash_{\mathsf{F}} Sub$), so $\Theta \vdash e : D$ and $\sigma \vDash \eta : (\Gamma, \Delta)$. Since $\Gamma <: \Theta$ we obtain $\sigma \vDash \eta : (\Theta, \Delta)$ by Lemma 2.3.7. Then, by induction hypothesis, we get $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : D)$ and, after applying Lemma 2.3.7 again, we obtain the desired $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C)$ since $D <: C$.

Case ($\vdash_{\mathsf{F}} Duplicate$) Assume that $\mathcal{D}$ was established in the last step by application of rule ($\vdash_{\mathsf{F}} Duplicate$), so $\Gamma, x_1 : D, \ldots, x_j : D \vdash e : C$ and $\sigma \vDash \eta : (\Gamma, y : D, \Delta)$. By Lemma 2.3.8 we obtain $\sigma \vDash \eta[x_1 \mapsto \eta_y, \ldots, x_j \mapsto \eta_y] : (\Gamma, x_1 : D, \ldots, x_j : D, \Delta)$. Then, by induction hypothesis, we obtain the desired $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C)$.

Case $(\vdash_{\mathsf{S}} New)$ We have $\dfrac{}{\Gamma \vdash \mathsf{new}\, C : C}$ $(\vdash_{\mathsf{F}} New)$ and

$$\frac{\ell \notin \mathrm{dom}(\sigma) \quad \tau = \sigma[l \mapsto (C, a_1 : \mathsf{0}, \ldots, a_k : \mathsf{0})] : (\Delta, x_{res} \mapsto C)}{\eta, \sigma \vdash \mathsf{new}\, C \rightsquigarrow l, \tau}$$

were $\mathsf{A}(C) = \{a_1, \ldots, a_k\}$. (2.3.3) follows by (2.3.6) and $C <: C$ which follows by definition. Item (2.3.2) follows directly by (2.3.6) and the rule $(\vdash_{\mathsf{S}} New)$. Item (2.3.1) follows by (2.3.6) and $l \in \tau$.

Case $(\vdash_{\mathsf{S}} Free)$ We have $\dfrac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash \mathtt{free}(x) \rightsquigarrow \mathsf{0}, \sigma[\ell \mapsto (\mathsf{invalid})]}$

We notice that item (2.3.2) follows trivially from (2.3.6) because it only imposes a condition to $l \in \mathrm{dom}(\sigma)$ with $\sigma(l) = (C, a_1 : v_1, \ldots, a_k : v_k)$ and in this case $\sigma(l) = (\mathsf{invalid})$. Moreover, $\eta_{x_{res}} = \mathsf{0} \notin \tau$, thus items (2.3.1) (2.3.3) follow directly from (2.3.6).

Case $(\vdash_{\mathsf{S}} Cast\,\mathrm{I})$ We have $\dfrac{\eta_x = \ell \qquad \sigma_\ell = (D, a_1 : v_1, \ldots, a_k : v_k) \qquad D <: C}{\eta, \sigma \vdash (C)x \rightsquigarrow \eta_x, \sigma}$

Item (2.3.3) follows by (2.3.6) and by $\underbrace{[\![\eta_x.\epsilon]\!]^{\mathrm{dyn}}_{\sigma}}_{D} <: \underbrace{[\![(\eta_x : C).\epsilon]\!]^{\mathrm{stat}}_{\sigma}}_{C}$ which follows by assumption. The rest follows directly by (2.3.6).

Case $(\vdash_{\mathsf{S}} Access)$ We have $\dfrac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash x.a_i \rightsquigarrow v_i, \sigma}$ Item (2.3.1) follows by (2.3.6), item (2.3.2), i.e. for all $\sigma(l) = (C, a_1 : v_1, \ldots, a_k : v_k)$ holds $v_i \in \mathrm{dom}(\sigma)$ and $\eta_{x_{res}} = v_i$, thus, $\mathrm{ran}(\eta) \subseteq \mathrm{dom}(\sigma) \cup \mathsf{0}$ follows. The rest follows by (2.3.6).

Case $(\vdash_{\mathsf{S}} Update)$ We have

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C_0, a_1 : v_1, \ldots, a_k : v_k) \qquad a = a_i \qquad \tau = \sigma[\ell.a_i \mapsto \eta_y]}{\eta, \sigma \vdash x.a \texttt{<-} y \rightsquigarrow \ell, \tau}$$

Item (2.3.2) follows by (2.3.6), item (2.3.1), i.e. $\tau_l = (C_0, a_1 : v_1, \ldots, a_i : \eta_y, \ldots, a_k : v_k)$ and $\eta_y \in \mathrm{dom}(\sigma) \cup \{\mathsf{0}\}$ by (2.3.6), item (2.3.1). The rest follows by (2.3.6).

Case $(\vdash_{\mathsf{S}} Let)$ We have

$$\frac{\eta, \sigma \vdash e_1 \rightsquigarrow v_1, \rho \qquad \eta[x \mapsto v_1], \rho \vdash e_2 \rightsquigarrow v_2, \tau}{\eta, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau}\ (\vdash_{\mathsf{S}} Let)$$

$$\frac{\Gamma_1 \vdash e_1 : D \qquad \Gamma_2, x : D \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ D\, x = e_1\ \mathsf{in}\ e_2 : C}\ (\vdash_{\mathsf{F}} Let)$$

Then, by induction hypothesis we get $\rho \vDash \eta[x_{res} \mapsto v_1] : (\Gamma_2, x_{res} : D)$. Then by induction hypothesis again we obtain our goal $\tau \vDash \eta[x_{res} \mapsto v_2] : (x_{res} : C)$.

Case ($\vdash_{\mathsf{S}} Inv$) We have

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k) \qquad \mathsf{M}_{\mathsf{body}}(C, m) = e_0 \qquad \overbrace{[\mathtt{this} \mapsto \ell, x_1 \mapsto \eta_{y_1}, \ldots, x_j \mapsto \eta_{y_j}]}^{\eta'}, \sigma \vdash e_0 \rightsquigarrow v, \tau}{\eta, \sigma \vdash x.m(\vec{y}) \rightsquigarrow v, \tau} \; (\vdash_{\mathsf{S}} Inv)$$

$$\frac{C.m = E_1, \ldots, E_j \to H}{x : C, y_1 : E_1, \ldots, y_j : E_j \vdash x.m(\vec{y}) : H} \; (\vdash_{\mathsf{F}} Invocation)$$

By Lemma 2.3.8 we obtain $\sigma \vDash \eta\eta' : (\mathtt{this} : C, x_1 : E_1, \ldots, x_j : E_j)$. Moreover, since without loss of generality $\mathrm{dom}(\eta) \cap \mathrm{dom}(\eta') = \emptyset$, we get by Lemma 2.3.9 $\eta\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau$. Then, by induction hypothesis, we obtain the required $\sigma \vDash \eta\eta'[x_{res} \mapsto v] : (\Delta, x_{res} : H)$.

□

2.3.2 Special semantics for heap-space analysis

In Fig. 2.5 we define special operational semantics for FJEU programs which consists of adding counters (non-negative natural numbers m and m') to the standard rules for keeping track of the number of unused heap units before and after evaluating e respectively. We will use this semantics in Chapter 3, where we define a type system for the static prediction of the heap-space requirements of FJEU programs. Similar counters can be introduced for keeping track of the usage of other kind of resources.

Note that rule ($\vdash_{\mathsf{Sh}} New$) is only applicable if there are enough unused heap units available for the allocation of a new object, otherwise the evaluation of a ($\vdash_{\mathsf{Sh}} New$) expression gets stuck. We require 1 heap unit to allocate any object for simplicity. Practical implementations would take into account the size of the object and how many memory cells are needed for its allocation in the heap. Heap units are unconditionally made available again by the `free` expression. Our analysis will identify an upper bound on m and a lower bound on m' for a given expression and program.

We include the rule ($\vdash_{\mathsf{Sh}} Waste$) which allows us to abandon unused memory units in order to facilitate some proofs.

Lemma 2.3.11 *Let e be an expression. σ, τ be heaps and η be a stack. Then: $\exists\, m, m' \,.\, \mathcal{D} :: \eta, \sigma \overset{m}{\vdash}_{m'} e \rightsquigarrow v, \tau$ iff $\mathcal{E} :: \eta, \sigma \vdash e \rightsquigarrow v, \tau$.*

Proof.

Case "⇒" By induction on the derivation $\mathcal{D}$.

Case "⇐" By induction on the derivation $\mathcal{E}$.

□

Operational Semantics of FJEU for heap-space analysis $\boxed{\eta, \sigma \ \vdash^{m}_{m'} \ e \rightsquigarrow v, \tau}$

$$\frac{}{\eta, \sigma \ \vdash^{m}_{m} \ x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{Sh}} Var) \qquad \frac{}{\eta, \sigma \ \vdash^{m}_{m} \ \mathsf{null} \rightsquigarrow 0, \sigma} \ (\vdash_{\mathsf{Sh}} Null)$$

$$\frac{\ell \notin \mathrm{dom}(\sigma) \qquad \mathsf{A}(C) = \{a_1, \ldots, a_k\} \qquad \tau = \sigma[\ell \mapsto (C, a_1 : 0, \ldots, a_k : 0)]}{\eta, \sigma \ \vdash^{m+1}_{m} \ \mathtt{new}\ C \rightsquigarrow \ell, \tau} \ (\vdash_{\mathsf{Sh}} New)$$

$$\frac{\eta_x = \ell \quad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \ \vdash^{m}_{m+1} \ \mathtt{free}(x) \rightsquigarrow 0, \sigma[\ell \mapsto (\mathsf{invalid})]} \ (\vdash_{\mathsf{Sh}} Free) \qquad \frac{\eta, \sigma \ \vdash^{m}_{m'+d} \ e \rightsquigarrow v, \tau}{\eta, \sigma \ \vdash^{m}_{m'} \ e \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{Sh}} Waste)$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (D, a_1 : v_1, \ldots, a_k : v_k) \qquad D <: C}{\eta, \sigma \ \vdash^{m}_{m} \ (C)x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{Sh}} Cast\,\mathrm{I})$$

$$\frac{\eta_x = 0}{\eta, \sigma \ \vdash^{m}_{m} \ (C)x \rightsquigarrow \eta_x, \sigma} \ (\vdash_{\mathsf{Sh}} Cast\,\mathrm{II}) \qquad \frac{\eta_x = \ell \quad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \ \vdash^{m}_{m} \ x.a_i \rightsquigarrow v_i, \sigma} \ (\vdash_{\mathsf{Sh}} Access)$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C_0, a_1 : v_1, \ldots, a_k : v_k) a = a_i \qquad \tau = \sigma[\ell.a_i \mapsto \eta_y]}{\eta, \sigma \ \vdash^{m}_{m} \ x.a\mathtt{<-}y \rightsquigarrow \ell, \tau} \ (\vdash_{\mathsf{Sh}} Update)$$

$$\frac{\eta, \sigma \ \vdash^{m}_{m'} \ e_1 \rightsquigarrow v_1, \rho \qquad \eta[x \mapsto v_1], \rho \ \vdash^{m'}_{m''} \ e_2 \rightsquigarrow v_2, \tau}{\eta, \sigma \ \vdash^{m}_{m''} \ \mathsf{let}\ x\ \mathtt{=}\ e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau} \ (\vdash_{\mathsf{Sh}} Let)$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k) \qquad \mathsf{M}_{\mathsf{body}}(C, m) = e_0 \\ [\mathtt{this} \mapsto \ell, x_1 \mapsto \eta_{y_1}, \ldots, x_j \mapsto \eta_{y_j}], \sigma \ \vdash^{m}_{m'} \ e_0 \rightsquigarrow v, \tau\end{array}}{\eta, \sigma \ \vdash^{m}_{m'} \ x.m(\vec{y}) \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{Sh}} Inv)$$

$$\frac{\eta_x = \ell \qquad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \quad D <: E \qquad \eta, \sigma \ \vdash^{m}_{m'} \ e_1 \rightsquigarrow v, \tau}{\eta, \sigma \ \vdash^{m}_{m'} \ \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{Sh}} Cond\,\mathrm{I})$$

$$\frac{\eta_x = \ell \quad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \quad D \not<: E \qquad \eta, \sigma \ \vdash^{m}_{m'} \ e_2 \rightsquigarrow v, \tau}{\eta, \sigma \ \vdash^{m}_{m'} \ \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{Sh}} Cond\,\mathrm{II})$$

$$\frac{\eta_x = 0 \qquad \eta, \sigma \ \vdash^{m}_{m'} \ e_2 \rightsquigarrow v, \tau}{\eta, \sigma \ \vdash^{m}_{m'} \ \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \ (\vdash_{\mathsf{Sh}} \mathrm{Cond\ III})$$

Figure 2.5: Operational semantics of FJEU for heap-space analysis.

Lemma 2.3.12 (Waste)

1. $\forall d \in \mathbb{N}\,.\,\mathcal{D} :: \eta, \sigma \;\vdash^{m}_{m'}\; e \rightsquigarrow v, \tau \implies \eta, \sigma \;\vdash^{d+m}_{d+m'}\; e \rightsquigarrow v, \tau.$
2. *Let* $\mathcal{D} :: \eta, \sigma \;\vdash^{m}_{m'}\; e \rightsquigarrow v, \tau$. *Then, if* $n \geq m$ *also* $\eta, \sigma \;\vdash^{n}_{m'}\; e \rightsquigarrow v, \tau$.

Proof.

1. By induction on the derivation $\mathcal{D}$.
2. We have $n = m + x$ for some x. By item 1. we have $\eta, \sigma \;\vdash^{n}_{m'+x}\; e \rightsquigarrow v, \tau$, and by rule $(\vdash_{\mathsf{Sh}} Waste)$ we get $\eta, \sigma \;\vdash^{n}_{m'}\; e \rightsquigarrow v, \tau$.

□

It is convenient to extend the special semantics to real annotations so that these annotations can be found by solving linear arithmetic constraints.

Definition 2.3.13 *We extend the operational semantics to real annotations* $t, t' \in \mathbb{R}^{+}_{0}$ *in the following way:*

$$\eta, \sigma \;\vdash^{t}_{t'}\; e \rightsquigarrow v, \tau \;\overset{\text{def}}{\iff}\; \eta, \sigma \;\vdash^{\lceil t \rceil}_{\lceil t' \rceil}\; e \rightsquigarrow v, \tau$$

In the following chapters we shall study the problems of describing the annotations using refined types and finding them automatically via type inference.

Chapter 3

Resource Aware Java

In this chapter we extend the typing system of FJEU to a system of refined types called Resource Aware JAva (RAJA) for the compile-time analysis of the heap-space requirements of FJEU programs.

Recall that we assume a simple freelist based model where we maintain a set of free memory units, the freelist. When creating an object, a heap unit required to store it is taken from the freelist, provided it contains enough units. When deallocating an object, the unit returns to the freelist.

The goal of the analysis is to predict a bound on the initial size that the freelist must have so that a given program may be executed without causing unsuccessful abortion due to insufficient memory. We achieve this by combining amortised analysis [Tar85, Oka98] with type-based techniques in order to define potentials.

Essentially each object is ascribed an abstracted portion of the freelist, referred to as *potential*, which is just a nonnegative number, denoting the size of freelist portion associated with the object. Any object creation must be paid for from the potential in scope. The initial potential thus represents an upper bound on the total heap consumption.

In this chapter we shall describe the typing system RAJA. In Section 3.1 we describe the system informally and show its use in some examples. Section 3.2 then introduces the typing system formally. In Section 3.3 we describe how to calculate a potential function based on the RAJA typing and prove the soundness of the system. Finally, Section 3.4 describes the problems of type checking and type inference that we will attempt to solve in the following chapters.

3.1 Informal presentation and examples

Before we describe the RAJA system, we would like to demonstrate the front end of our method with a couple of small examples. In Fig. 3.1 we define singly linked lists using the Composite pattern.

```
class List {
  List copy() {
    return null;
  }
}
class Nil extends List {
  List copy() {
    return new Nil;
  }
}
class Cons extends List {
  Object elem;
  List next;

  List copy() {
    let res = new Cons() in
    let _ = res.elem <- this.elem in
    let _ = res.next <- this.next.copy() in
    return res;
  }
}
class Main {
  List main(List l) {
    return l.copy();
  }
}
```

Figure 3.1: Copying lists in FJEU.

We define the class List as an abstract list, the class Cons for describing the nodes and the class Nil for modelling the end of the list. Moreover, we define a method for copying lists. Running the analysis on the program yields the following results; no annotations by the programmer are required.

```
Program will execute successfully with a free-list >=
                                    1. + 1. * length of the input
```

It is clear that the heap-space consumption of this program is exactly the length of the list plus 1, since during the execution of the program one Cons object for each node of the list is allocated and additionally one Nil node is allocated.

Now let us consider a method for appending lists (Fig. 3.2). The call `this.appAux(y, dest)` imperatively appends `y` to `this` and places the re-

sult into `dest.next`. The auxiliary object `dest` is subsequently deallocated and the concatenated list is returned. When we analyse the program we obtain the following output:

```
Program will execute successfully with a free-list >= 1.
```

```
class List {
  Cons appAux(List y, Cons dest) {
    return null;
  }
  List append(List y) {
    let dest = new Cons in
    let _ = this.appAux(y, dest) in
    let result = dest.next in
    let _ = free(dest) in
    return result;
  }
}
class Nil extends List {
  Cons appAux(List y, Cons dest) {
    return dest.next <- y;
  }
}
class Cons extends List {
  Object elem;
  List next;

  Cons appAux(List y, Cons dest) {
    let _ = dest.next <- this in
    return this.next.appAux(y, this);
  }
}
class Main {
  List main(List l1, List l2) {
    return l1.append(l2);
  }
}
```

Figure 3.2: Appending lists in FJEU.

Notice that the heap-space consumption of the program is indeed constant since the list is appended in place and the only allocated object is `dest`. This example offers challenges to the analysis nevertheless, due to the aliasing

caused by the multiple use of the variable `this` in the method `appAux` in the class `Cons`. Later we will show how this example can be typed in our system.

3.1.1 Refined types, views and potential

As already mentioned, our approach is based on amortised analysis and we wish to assign data structures a potential that can be used to pay for any object creation. We could assign potential to classes, but then every object of a given class would have the same potential, which would be very inflexible. Indeed, we should be able to assign different objects of the same class different potentials, so that we need to *refine* the notion of classes. The solution is to introduce the *views* and to build *refined types*, that we can assign potential to, by combining classes and views. If C is a class and r is a view, then C^r is a refined type.

The set of views $\mathscr{V}^{\mathscr{C}}$ is defined coinductively on the basis of a given FJEU program by the maps $\Diamond(.)$, $\mathsf{A}^{\mathrm{get}}(.,.)$ and $\mathsf{A}^{\mathrm{set}}(.,.)$. The potential function

$$\Diamond(.) : \mathscr{C} \times \mathscr{V}^{\mathscr{C}} \to \mathbb{R}_0^+ \cup \infty$$

assigns each refined type C^r a nonnegative real number or infinite, the potential. The maps

$$\mathsf{A}^{\mathrm{get}}(.,.) : \forall C \in \mathscr{C} \,.\, \mathscr{V}^{\mathscr{C}} \times \mathsf{A}(C) \to \mathscr{V}^{\mathscr{C}}$$

and

$$\mathsf{A}^{\mathrm{set}}(.,.) : \forall C \in \mathscr{C} \,.\, \mathscr{V}^{\mathscr{C}} \times \mathsf{A}(C) \to \mathscr{V}^{\mathscr{C}}$$

assign views to the fields, where $\mathsf{A}^{\mathrm{get}}(C^r, a)$ represents the view used when reading the field a of class C under the view r, and $\mathsf{A}^{\mathrm{set}}(C^r, a)$ is the view used when writing the field a.

We define views coinductively because in this way they have a semantic status, namely infinite labelled trees. This is useful in the constraint generation and solving.

3.1.2 Monomorphic and polymorphic method types

The refined types for methods consist of views for the method's arguments (including this), a view for its result and two numbers representing the potential consumed and released by the method, respectively. More concretely, if a method m has a method type $C^{r_0}; E_1^{r_1}, \ldots, E_n^{r_n} \xrightarrow{m/m'} H^{r_{n+1}}$, this means that it is defined in the refined type C^{r_0} and may be called with arguments $v_1 : E_1^{r_1}, \ldots, v_n : E_n^{r_n}$, whose associated potential will be consumed, as well as an additional potential of m. The return value will be of type $H^{r_{n+1}}$, carrying an according potential. In addition to this a potential of another

m' units will be returned. We call these types *monomorphic RAJA method types*.

In the original description of the system RAJA [HJ06] Hofmann and Jost introduced the notion of *polymorphic RAJA method types* as a set of monomorphic RAJA method types. In this thesis we give a different definition of polymorphic method types: they are as monomorphic RAJA method types, with views and numbers replaced by variables and constraints upon them. Concretely, a polymorphic type of a method m of class C with FJEU type $\vec{E} \to H$ consists of view variables for its arguments, a view variable for its result and two number variables, together with a conjunction of subtyping and arithmetic constraints, written

$$\phi = \forall \vec{v}, q_1, q_2 \,.\, C^{v_0}; \vec{E}^{\vec{v}} \xrightarrow{q_1/q_2} H^{v_{n+1}} \;\&\; \mathcal{C}$$

The subtyping constraints describe relations between refined types which can be reduced to relations between views as we will see later. For instance, if a view r satisfies $\mathsf{A}^{\mathrm{get}}(C^{v_0}, a) \sqsubseteq v_0$ then the get view of the field a of class C under r must be a subtype of r. A view r is a subtype of a view s if for all classes C holds $\Diamond(C^r) \geq \Diamond(C^s)$, i.e. the potential of C under r is greater or equal than the potential of C under s. Moreover, for each field a of C we must have $\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^s, a)$, thus, we say that subtyping is *covariant* in the get views. For the set views, however, subtyping is *contravariant*, which means that if r is a subtype of s, then $\mathsf{A}^{\mathrm{set}}(C^s, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a)$.

The arithmetic constraints describe relations between the potential of views and real numbers. For instance, for any two views r_0 and r_1 and number n satisfying $\Diamond(C^{v_0}) \geq \Diamond(C^{v_1}) + p$ must hold that the potential of the class C under r_0 is greater or equal than the potential of C under r_1 plus n.

3.1.3 Potential of a runtime configuration

One runtime object can have several refined types at once, since it can be regarded through different views at the same time. The overall potential of a runtime configuration is the (possibly infinite) sum over all access paths in scope that lead to an actual object. Thus, if an object has several access paths leading to it (aliasing) it may make several contributions to the total potential. Our type system has an explicit contraction rule: If a variable is used more than once, the associated potential is split by assigning different views to each use.

3.1.4 Examples

In the following we wish to illustrate the system by showing the details of the analysis of the programs for copying and appending lists from Figures 3.1 and 3.2.

Copying lists

We wish to analyse the heap-space requirements of the method copy(), which is equal to the length of the list to be copied plus 1, as we have seen before. Recall the definition of copy() in class Cons:

```
List copy() {
  let res = new Cons() in
  let _ = res.elem <- this.elem in
  let _ = res.next <- this.next.copy() in
  return res;
}
```

For each node of the list, a new node is created, the rest of the list is copied recursively and the new node's element and pointer to the next node in the list are updated.

Each node of the original list needs to have enough potential to pay for the creation of the new node. This requirement is captured in the constraint $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}) + 1$, where $\mathsf{v}_{\mathsf{self}}$ is the view variable that corresponds to the argument this and $\mathsf{v}_{\mathsf{res}}$ is the view variable corresponding to the resulting list. Recall that for simplicity we assume that every object can be allocated in only one cell of the heap.

Moreover, since the method is called recursively with the next item in the list, the potential of the next node must be greater or equal than the potential of the current node. More generally, the refined type of the next node must be a subtype of the refined type of the current node, which is expressed in the constraint $\mathsf{A}^{\mathsf{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}; \mathsf{next}) \sqsubseteq \mathsf{v}_{\mathsf{self}}$. There are no restrictions to the potential of the resulting node but its refined type should be a subtype of the refined type of its next node ($\mathsf{v}_{\mathsf{res}} \sqsubseteq \mathsf{A}^{\mathsf{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}; \mathsf{next})$).

In summary, the RAJA polymorphic method type of the method copy() is the following:

$$\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v}_{\mathsf{res}}} \;\&\; \begin{array}{l} \mathsf{A}^{\mathsf{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}; \mathsf{next}) \sqsubseteq \mathsf{v}_{\mathsf{self}} \wedge \mathsf{v}_{\mathsf{res}} \sqsubseteq \mathsf{A}^{\mathsf{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}; \mathsf{next}) \\ \wedge \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}) + 1 \end{array}$$

Now recall the definition of copy() in class Nil:

```
List copy() {
  return new Nil();
}
```

The variable this must pay for the object creation, thus, the polymorphic type for this method is:

$$\mathsf{Nil}^{\mathsf{v}_{\mathsf{self}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v}_{\mathsf{res}}} \;\&\; \Diamond(\mathsf{Nil}^{\mathsf{v}_{\mathsf{self}}}) \geq \Diamond(\mathsf{Nil}^{\mathsf{v}_{\mathsf{res}}}) + 1$$

The body of the method copy() in class List is just null, since this method is not supposed to be called. Thus, there are no restrictions to this method's type. However, the analysis would not be sound if the method's polymorphic type contained no constraints at all because an object of type List can have dynamic type Cons, causing that the actual method to be executed at runtime is the method copy of class Cons, leading to unpredictable heap-space consumption. Hence the correct type for copy in List is built by adding the constraints of the copy method in the subclasses Cons and Nil:

$$\begin{aligned}\mathsf{List}^{\mathsf{v}_{\mathsf{self}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v}_{\mathsf{res}}} \;\&\quad & \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}, \mathsf{next}) \sqsubseteq \mathsf{v}_{\mathsf{self}} \wedge \mathsf{v}_{\mathsf{res}} \sqsubseteq \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}, \mathsf{next}) \\ & \wedge \diamondsuit(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) \geq \diamondsuit(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}) + 1 \\ & \wedge \diamondsuit(\mathsf{Nil}^{\mathsf{v}_{\mathsf{self}}}) \geq \diamondsuit(\mathsf{Nil}^{\mathsf{v}_{\mathsf{res}}}) + 1\end{aligned}$$

Recall that the main method calls the copy method with the input list l:

```
class Main {
  List main(List l) {
    return l.copy();
  }
}
```

It is intuitively clear that the heap-space consumption of main is just the heap-space consumption of the copy method. In fact, the constraints of method calls consist of the constraints of the called method after variable renaming. We obtain the following type for main:

$$\begin{aligned}\mathsf{Main}^{\mathsf{v}_{\mathsf{self}}}; \mathsf{List}^{\mathsf{v}_{\mathsf{l}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v}_{\mathsf{res}}} \;\&\quad & \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{l}}}, \mathsf{next}) \sqsubseteq \mathsf{v}_{\mathsf{l}} \wedge \mathsf{v}_{\mathsf{res}} \sqsubseteq \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}, \mathsf{next}) \\ & \wedge \diamondsuit(\mathsf{Cons}^{\mathsf{v}_{\mathsf{l}}}) \geq \diamondsuit(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}) + 1 \\ & \wedge \diamondsuit(\mathsf{Nil}^{\mathsf{v}_{\mathsf{l}}}) \geq \diamondsuit(\mathsf{Nil}^{\mathsf{v}_{\mathsf{res}}}) + 1\end{aligned}$$

It is not hard to see that the valuation

$$\pi = (\{\mathsf{v}_{\mathsf{self}} \mapsto \mathsf{rich}, \mathsf{v}_{\mathsf{l}} \mapsto \mathsf{rich}, \mathsf{v}_{\mathsf{res}} \mapsto \mathsf{poor}\}, \{q_1 \mapsto 0, q_2 \mapsto 0\}$$

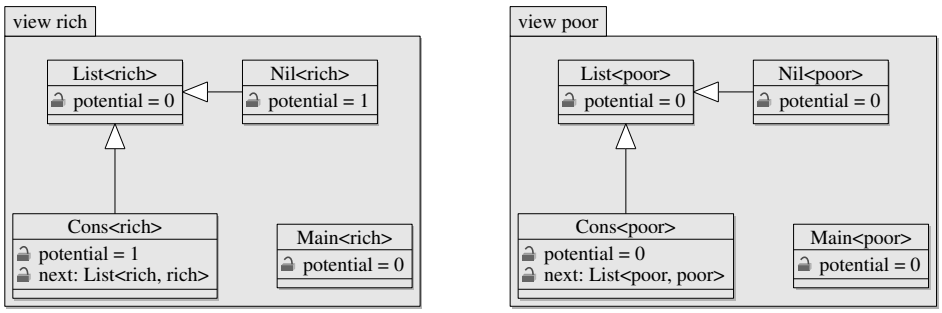
builds the best possible solution for the constraints of all methods, where rich and poor are the following views:

$\diamondsuit(\cdot)$	rich	poor
List	0	0
Nil	1	0
Cons	1	0
Main	0	0

	$\mathsf{Cons}^{\mathsf{rich}}$	$\mathsf{Cons}^{\mathsf{poor}}$
$\mathsf{A}^{\mathrm{get}}(\cdot, \mathsf{next})$	rich	poor
$\mathsf{A}^{\mathrm{set}}(\cdot, \mathsf{next})$	rich	poor

(3.1.1)

The names rich and poor were chosen for remarking that some classes under the view rich have potential 1 and the classes under the view poor have potential 0. We could compare potential with money metaphorically, since you can spend it for paying for object creation. In this sense a view with some potential can be said to be "rich" and a view with no potential to be "poor". Notice that we could visualise the views rich and poor as follows:

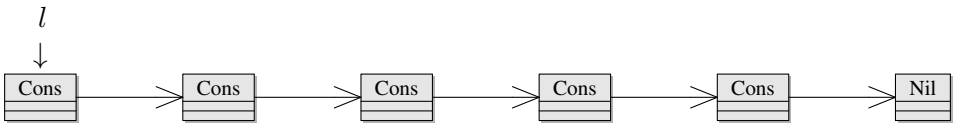


We then obtain the following monomorphic RAJA method types:

$$\begin{array}{lcl} \mathsf{M}(\mathsf{List}, \mathsf{copy}) & = & \mathsf{List}^{\mathsf{rich}} \xrightarrow{0/0} \mathsf{List}^{\mathsf{poor}} \\ \mathsf{M}(\mathsf{Cons}, \mathsf{copy}) & = & \mathsf{Cons}^{\mathsf{rich}} \xrightarrow{0/0} \mathsf{List}^{\mathsf{poor}} \\ \mathsf{M}(\mathsf{Nil}, \mathsf{copy}) & = & \mathsf{Nil}^{\mathsf{rich}} \xrightarrow{0/0} \mathsf{List}^{\mathsf{poor}} \\ \mathsf{M}(\mathsf{Main}, \mathsf{main}) & = & \mathsf{Main}^{\mathsf{rich}}; \mathsf{List}^{\mathsf{rich}} \xrightarrow{0/0} \mathsf{List}^{\mathsf{poor}} \end{array}$$

These types state that copy is only defined in $\mathsf{List}^{\mathsf{rich}}$, $\mathsf{Nil}^{\mathsf{rich}}$ and $\mathsf{Cons}^{\mathsf{rich}}$, but not in, e.g., $\mathsf{List}^{\mathsf{poor}}$ and the potential of this will be consumed. The return value will be of type $\mathsf{List}^{\mathsf{poor}}$ hence carry potential 0. Moreover, since the type of the argument l in the main method is $\mathsf{List}^{\mathsf{rich}}$, the method copy can be called and the potential of l will be consumed, which is equal to the length of the list plus 1. This is the overall potential consumed by the program which is equal to its heap-space consumption.

Now, why is the potential of a list of type $\mathsf{List}^{\mathsf{rich}}$ equal to $n+1$, when n is the length of the list? Suppose that l points to a list of length 5.



The potential is calculated as the sum over all access paths starting from l and not leading to null. In this case, these access paths are

- $\vec{p}_1 = l$
- $\vec{p}_2 = l.\,\mathsf{next}$
- $\vec{p}_3 = l.\,\mathsf{next}.\,\mathsf{next}$

- $\vec{p}_4 = l.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}$
- $\vec{p}_5 = l.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}$
- $\vec{p}_6 = l.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}.\,\mathsf{next}$

Each of these has a dynamic type: Cons for $\vec{p}_1$ to $\vec{p}_5$ and Nil for $\vec{p}_6$. Each of them also has a view that can be computed by chaining the view of l along the get views, which is the view rich in each case. For each access path, we now look up the potential annotation of its dynamic type under its view. It equals 1 in every case given $\Diamond(\mathsf{Cons}^{\mathsf{rich}}) = \Diamond(\mathsf{Nil}^{\mathsf{rich}}) = 1$, yielding a sum of 6.

Appending lists

Now let us look at the analysis of the program for appending lists imperatively in more detail. Recall the definition of the method appAux in class Cons.

```
Cons appAux(List y, Cons dest) {
  let _ = dest.next <- this in
  return this.next.appAux(y, this);
}
```

Here we wish to focus on how the system handles aliasing. Notice that the variable this is used three times in the method's body, hence its potential has to be shared among all occurrences for the analysis to be sound. The view variable corresponding to this is again $\mathsf{v}_{\mathsf{self}}$ and the view variable corresponding to dest is $\mathsf{v}_{\mathsf{dest}}$. Then, we obtain the constraint $\mathsf{v}_{\mathsf{self}} \sqsubseteq \mathsf{v}_{\mathsf{self}} \oplus \mathsf{v}_{\mathsf{self}} \oplus \mathsf{v}_{\mathsf{dest}}$ which implies that $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) + \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) + \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{dest}}})$ must hold. Notice that this implies that $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}) = 0$.

Moreover we get the constraint $\mathsf{v}_{\mathsf{self}} \sqsubseteq \mathsf{A}^{\mathrm{set}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{dest}}}, \mathsf{next})$ for the update `dest.next <- this` to be correct and the constraint $\mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v}_{\mathsf{self}}}, \mathsf{next}) \sqsubseteq \mathsf{v}_{\mathsf{self}}$ since the method is called recursively with this. next. The following valuation builds a solution for these constraints.

$$\pi = \{\mathsf{v}_{\mathsf{self}} \mapsto \mathsf{poor}, \mathsf{v}_{\mathsf{dest}} \mapsto \mathsf{n}\}$$

where poor and n are defined as follows:

$\Diamond(\cdot)$	poor	n
Cons	0	0
List	0	0
Nil	0	0
Main	0	0

	$\mathsf{Cons}^{\mathsf{poor}}$	$\mathsf{Cons}^{\mathsf{n}}$
$\mathsf{A}^{\mathrm{get}}(\cdot\,, \mathsf{next})$	poor	n
$\mathsf{A}^{\mathrm{set}}(\cdot\,, \mathsf{next})$	poor	poor

(3.1.2)

We say in this case that poor can be split into poor, poor and n, written $\curlyvee(\mathsf{poor}\,|\mathsf{poor}, \mathsf{poor}, \mathsf{n}\,)$, which is equivalent to $\mathsf{poor} \sqsubseteq \mathsf{poor} \oplus \mathsf{poor} \oplus \mathsf{n}$.

Circular lists

As a final example, we wish to show how our system prevents cyclic lists from being copied. If this was allowed, the program would not terminate and would require infinite amount of memory. Assume the following main method that creates a list with a loop and copies it.

```
List main() {
  let cons = new Cons in
  let _ = cons.next <- cons in
  return cons.copy();
}
```

Let $\mathsf{v}_{\mathsf{cons}}$ be the view variable corresponding to cons, then we obtain the constraint $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{cons}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{res}}}) + 1$ by substituting the variable $\mathsf{v}_{\mathsf{self}}$ in the constraints from the copy method with the variable $\mathsf{v}_{\mathsf{cons}}$. Notice that this requires the potential of Cons under $\mathsf{v}_{\mathsf{cons}}$ to be at least 1.

Further, handling aliasing gives us the constraint $\mathsf{v}_{\mathsf{cons}} \sqsubseteq \mathsf{v}_{\mathsf{cons}} \oplus \mathsf{v}_{\mathsf{cons}} \oplus \mathsf{v}_{\mathsf{cons}}$, making the system of constraints unsolvable since, on the one hand, $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{cons}}}) \geq 1$ and on the other hand, $\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{cons}}}) \geq 3\Diamond(\mathsf{Cons}^{\mathsf{v}_{\mathsf{cons}}})$.[1]

Notice that this is again a consequence of our rule for aliasing that guarantees soundness at all times.

3.2 Introduction to the RAJA system

In this section we describe the RAJA system formally including views, refined types and the RAJA typing system.

3.2.1 Views and refined types

We set $\mathbb{D} = \mathbb{R}_0^+ \cup \{\infty\}$, i.e., the set of nonnegative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}_0^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$.

Definition 3.2.1 *We define the set $\mathscr{V}^{\mathscr{C}}$ of views coinductively by*

- $\Diamond(\cdot)$ *assigns to each view $r \in \mathscr{V}^{\mathscr{C}}$ and class $C \in \mathscr{C}$ a number $\Diamond(C^r)$.*
- $\mathsf{A}^{\mathrm{get}}(\cdot,\cdot)$ *assigns to each view $r \in \mathscr{V}^{\mathscr{C}}$ and class $C \in \mathscr{C}$ and field $a \in \mathsf{A}(C)$ a view $s = \mathsf{A}^{\mathrm{get}}(C^r, a)$.*
- $\mathsf{A}^{\mathrm{set}}(\cdot,\cdot)$ *assigns to each view $r \in \mathscr{V}^{\mathscr{C}}$ and class $C \in \mathscr{C}$ and field $a \in \mathsf{A}(C)$ a view $s' = \mathsf{A}^{\mathrm{set}}(C^r, a)$.*

[1] This analysis has been simplified for the explanation but it shows the main idea.

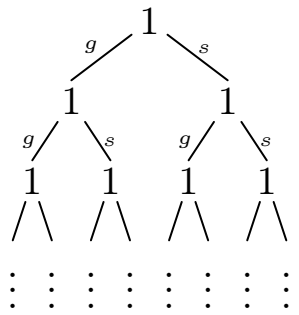


Figure 3.3: Infinite tree representing view rich.

We remark that a view can be regarded as an infinite tree given by

$$\mathscr{V}^{\mathscr{C}} = \{t \mid t{:}\mathcal{L}^*_{\mathscr{C}} \to \mathbb{D}\}$$

where the set of labels $\mathcal{L}_{\mathscr{C}}$ is defined as follows:

$$\mathcal{L}_{\mathscr{C}} = \{C_k.a_j.t \mid C_k \in \mathscr{C}, a_j \in \mathsf{A}(C_k)\,, t \in \{\mathsf{get}, \mathsf{set}\}\}$$

Further, we say that a view is *regular* if it is a regular infinite tree, i.e. if it contains a finite number of different sub-trees. For instance, the view rich described in Section 3.1.4 can be represented as the regular infinite tree displayed in Fig. 3.2.1 where g stands for Cons. next.get and s for Cons. next.set.

We define an inequality relation between views $r \sqsubseteq s$ coinductively.

Definition 3.2.2 ($\mathbf{r \sqsubseteq s}$) *Let $r, s \in \mathscr{V}^{\mathscr{C}}$. We define $r \sqsubseteq s$ coinductively by*

$$\forall C \in \mathscr{C}\,.\,\langle\!\langle C^r\rangle\!\rangle \geq \langle\!\langle C^s\rangle\!\rangle \tag{3.2.1}$$

$$\forall C \in \mathscr{C}\,\forall a \in \mathsf{A}(C)\;.\,\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^s, a) \tag{3.2.2}$$

$$\forall C \in \mathscr{C}\,\forall a \in \mathsf{A}(C)\;.\,\mathsf{A}^{\mathrm{set}}(C^s, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a) \tag{3.2.3}$$

Notice that the defined preorder $\sqsubseteq$ on views is *covariant* in the get views and *contravariant* in the set views.
We define the operations $\vee : \mathscr{V}^{\mathscr{C}} \times \mathscr{V}^{\mathscr{C}} \to \mathscr{V}^{\mathscr{C}}$ and $\wedge : \mathscr{V}^{\mathscr{C}} \times \mathscr{V}^{\mathscr{C}} \to \mathscr{V}^{\mathscr{C}}$ simultaneously. Let $s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$, then, for each $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$ we set:

$$\begin{array}{rcl}
\langle\!\langle C^{s_1 \vee s_2}\rangle\!\rangle & = & \mathsf{min}(\,\langle\!\langle C^{s_1}\rangle\!\rangle, \langle\!\langle C^{s_2}\rangle\!\rangle) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 \vee s_2}, a) & = & \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 \vee s_2}, a) & = & \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \wedge \mathsf{A}^{\mathrm{set}}(C^{s_2}, a)
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle C^{s_1 \wedge s_2}\rangle\!\rangle & = & \mathsf{max}(\,\langle\!\langle C^{s_1}\rangle\!\rangle, \langle\!\langle C^{s_2}\rangle\!\rangle) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 \wedge s_2}, a) & = & \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \wedge \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 \wedge s_2}, a) & = & \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}, a)
\end{array}$$

Lemma 3.2.3 *Let $s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$. Then for all $i \in \{1,2\}$ holds*

1. $s_i \sqsubseteq s_1 \vee s_2$.
2. $s_1 \wedge s_2 \sqsubseteq s_i$.

Proof. Simultaneously by coinduction.

1. We show (3.2.1), (3.2.2) and (3.2.3). Let $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$. Then:
   (a) (3.2.1) follows by $\Diamond\!(C^{s_i}) \geq \mathsf{min}(\Diamond\!(C^{s_1}), \Diamond\!(C^{s_2}))$.
   (b) (3.2.2) follows by $\mathsf{A}^{\mathrm{get}}(C^{s_i}; a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{get}}(C^{s_2}; a)$, which follows by the coinduction hypothesis (1).
   (c) (3.2.3) follows by $\mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \wedge \mathsf{A}^{\mathrm{set}}(C^{s_2}; a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_i}; a)$, which follows by the coinduction hypothesis (2).
2. We show (3.2.1), (3.2.2) and (3.2.3). Let $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$. Then:
   (a) (3.2.1) follows by $\mathsf{max}(\Diamond\!(C^{s_1}), \Diamond\!(C^{s_2})) \geq \Diamond\!(C^{s_i})$.
   (b) (3.2.2) follows by $\mathsf{A}^{\mathrm{get}}(C^{s_1}; a) \wedge \mathsf{A}^{\mathrm{get}}(C^{s_2}; a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_i}; a)$, which follows by the coinduction hypothesis (2).
   (c) (3.2.3) follows by $\mathsf{A}^{\mathrm{set}}(C^{s_i}; a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}; a)$, which follows by the coinduction hypothesis (1).

□

Lemma 3.2.4 *Let $s_1, s_2, r \in \mathscr{V}^{\mathscr{C}}$. Then for all $i \in \{1,2\}$ holds*

1. *If $s_i \sqsubseteq r$ then $s_1 \vee s_2 \sqsubseteq r$.*
2. *If $r \sqsubseteq s_i$ then $r \sqsubseteq s_1 \wedge s_2$.*

Proof. Simultaneously by coinduction.

1. We show (3.2.1), (3.2.2) and (3.2.3). Let $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$. Then:
   (a) (3.2.1) follows by $\mathsf{min}(\Diamond\!(C^{s_1}), \Diamond\!(C^{s_2})) \geq \Diamond\!(C^{r})$, which follows by the assumption $\Diamond\!(C^{s_i}) \geq \Diamond\!(C^{r})$.
   (b) (3.2.2) follows by $\mathsf{A}^{\mathrm{get}}(C^{s_i}; a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{r}; a)$ implies $\mathsf{A}^{\mathrm{get}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{get}}(C^{s_2}; a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{r}; a)$, which follows by the coinduction hypothesis (1).
   (c) (3.2.3) follows by $\mathsf{A}^{\mathrm{set}}(C^{r}; a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_i}; a)$ implies $\mathsf{A}^{\mathrm{set}}(C^{r}; a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \wedge \mathsf{A}^{\mathrm{set}}(C^{s_2}; a)$, which follows by the coinduction hypothesis (2).
2. We show (3.2.1), (3.2.2) and (3.2.3). Let $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$. Then:
   (a) (3.2.1) follows by $\Diamond\!(C^{r}) \geq \mathsf{max}(\Diamond\!(C^{s_1}), \Diamond\!(C^{s_2}))$, which follows by the assumption $\Diamond\!(C^{r}) \geq \Diamond\!(C^{s_i})$.

(b) (3.2.2) follows by $\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_i}, a)$ implies $\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \wedge \mathsf{A}^{\mathrm{get}}(C^{s_2}, a)$, which follows by the coinduction hypothesis (2).

(c) (3.2.3) follows by $\mathsf{A}^{\mathrm{set}}(C^{s_i}, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a)$ implies $\mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a)$, which follows by the coinduction hypothesis (1).

□

Corollary 3.2.5 (Least upper bound and greatest lower bound)

1. *$s_1 \vee s_2$ is the least upper bound of s_1 and s_2.*

2. *$s_1 \wedge s_2$ is the greatest lower bound of s_1 and s_2.*

Proof. Follows by the Lemmas 3.2.3 and 3.2.4. □

Lemma 3.2.6 (Monotonicity of $\wedge$ and $\vee$) *Let $r, r', s, s' \in \mathscr{V}^{\mathscr{C}}$. Then:*

1. *if $r \sqsubseteq r'$ and $s \sqsubseteq s'$ then $r \wedge s \sqsubseteq r' \wedge s'$.*

2. *if $r' \sqsubseteq r$ and $s' \sqsubseteq s$ then $r' \vee s' \sqsubseteq r \vee s$.*

Proof. Simultaneously by coinduction. □

We define the addition operations on views $\oplus : \mathscr{V}^{\mathscr{C}} \times \mathscr{V}^{\mathscr{C}} \to \mathscr{V}^{\mathscr{C}}$ and $\boxplus : \mathscr{V}^{\mathscr{C}} \times \mathscr{V}^{\mathscr{C}} \to \mathscr{V}^{\mathscr{C}}$ simultaneously as follows. Let $s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$, then, for each $C \in \mathscr{C}$, $a \in \mathsf{A}(C)$ we set:

$$\begin{array}{rcl} \Diamond(C^{s_1 \oplus s_2}) & = & \Diamond(C^{s_1}) + \Diamond(C^{s_2}) \\ \mathsf{A}^{\mathrm{get}}(C^{s_1 \oplus s_2}, a) & = & \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\ \mathsf{A}^{\mathrm{set}}(C^{s_1 \oplus s_2}, a) & = & \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \end{array}$$

$$\begin{array}{rcl} \Diamond\big(C^{s_1 \boxplus s_2}\big) & = & \mathsf{min}(\, \Diamond(C^{s_1}), \Diamond(C^{s_2})) \\ \mathsf{A}^{\mathrm{get}}\big(C^{s_1 \boxplus s_2}, a\big) & = & \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\ \mathsf{A}^{\mathrm{set}}\big(C^{s_1 \boxplus s_2}, a\big) & = & \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \end{array}$$

Lemma 3.2.7 *Let $r, s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$ and let $i \in \{1, 2\}$. Then:*

1. *If $r \sqsubseteq s_1 \oplus s_2$ then $r \sqsubseteq s_i$.*

2. *If $s_1 \boxplus s_2 \sqsubseteq r$ then $s_i \sqsubseteq r$.*

Proof. Simultaneously by coinduction. □

Lemma 3.2.8 (Monotonicity of $\oplus$ and $\boxplus$.) *Let $r, r', s, s' \in \mathscr{V}^{\mathscr{C}}$ and let $i \in \{1, 2\}$.*

1. *If $r \sqsubseteq r'$ and $s \sqsubseteq s'$ then $r \oplus s \sqsubseteq r' \oplus s'$.*
2. *If $r' \sqsubseteq r$ and $s' \sqsubseteq s$ then $r' \boxplus s' \sqsubseteq r \boxplus s$.*

Proof. Simultaneously by coinduction. □

A *RAJA type* or *refined type* consists of a class C and a view r and is written C^r. If C^r is a refined type then we denote by $|C^r| = C$ the underlying FJEU type C and by $\langle\!\langle C^r \rangle\!\rangle = r$ its view. We extend the subtyping of FJEU classes (Fig. 2.2) to refined types as follows.

Definition 3.2.9 ($\mathbf{C^r <: D^s}$) *We extend subtyping to refined types by*

$$C^r <: D^s \iff C <: D \text{ and } r \sqsubseteq s \tag{3.2.4}$$

Notice that since both $\sqsubseteq$ and $<:$ on FJEU are reflexive and transitive so is $<:$ on RAJA. In the following, when we refer to subtyping, unless specified otherwise, we mean subtyping of refined types.

In contrast to the original RAJA system [HJ06], our modified system does not require the set view of a field a of a refined type C^s to be a subtype of its get view in all cases, i.e. we do not require $\mathsf{A}^{\text{set}}(C^s, a) \sqsubseteq \mathsf{A}^{\text{get}}(C^s, a)$ to hold for every refined type C^s and field $a \in \mathsf{A}(C)$. We shall see later that, for proving soundness, $\mathsf{A}^{\text{set}}(C^s, a) \sqsubseteq \mathsf{A}^{\text{get}}(C^s, a)$ is only required for the dynamic type of a location in a heap and all its fields, and the main view that corresponds to that location. The advantage of this modification will become clear in Chapter 5, when we present a type inference algorithm for the system based on constraint generation; the modification will cause that less constraints are generated, which increases efficiency.

Definition 3.2.10 *Let C^r be a refined type. We say that C^r is a* main RAJA type *and write C^r main, if $\forall a \in \mathsf{A}(C)\ .\ \mathsf{A}^{\text{set}}(C^r, a) \sqsubseteq \mathsf{A}^{\text{get}}(C^r, a)$.*

3.2.2 Constraints

In this section we define formally the subtyping and arithmetic constraints that build polymorphic RAJA method types. Let $\mathsf{ViewVars}$ be an infinite set of view variables and $\mathsf{NumVars}$ be an infinite set of arithmetic variables. In the following grammar, we define a set of view expressions that we call $\mathsf{ViewExpr}$. A view expression is either a view variable v, a get or a set view of a given field a of a class C ($\mathsf{A}^{\text{get}}(C^v, a)$ or $\mathsf{A}^{\text{set}}(C^v, a)$) or the addition of two view variables $v_1 \oplus v_2$. Moreover, we define a set of arithmetic expressions called $\mathsf{ArithExpr}$. An arithmetic expression is either a

nonnegative real number n, an arithmetic variable p, a potential expression $\Diamond(C^v)$ where C is a class and v is a view variable or the sum of two arithmetic expressions $(\mathsf{ae}_1 + \mathsf{ae}_2)$. Further, we define a set of subtyping constraints $\mathsf{SConstr}$ with $C^{\mathsf{vexp}_1} <: D^{\mathsf{vexp}_2} \in \mathsf{SConstr}$, a set of constraints over views $\mathsf{VConstr}$ with $\mathsf{vexp}_1 \sqsubseteq \mathsf{vexp}_2 \in \mathsf{VConstr}$ and $\mathsf{vexp}_1 = \mathsf{vexp}_2 \in \mathsf{VConstr}$ and a set of arithmetic constraints $\mathsf{AConstr}$ with $\mathsf{ae}_1 \geq \mathsf{ae}_2 \in \mathsf{AConstr}$ and $\mathsf{ae}_1 \leq \mathsf{ae}_2 \in \mathsf{AConstr}$. tt is the empty constraint, i.e. a constraint that is always satisfied.

$$
\begin{array}{lcll}
\mathsf{vexp} & ::= & v \mid \mathsf{A}^{\mathrm{get}}(C^v, a) \mid \mathsf{A}^{\mathrm{set}}(C^v, a) \mid v \oplus v & \in \mathsf{ViewExpr} \\
\mathsf{ae} & ::= & n \mid p \mid \Diamond(C^v) \mid \mathsf{ae} + \mathsf{ae} & \in \mathsf{ArithExpr} \\
\mathcal{SC} & ::= & C^{\mathsf{vexp}} <: D^{\mathsf{vexp}} & \in \mathsf{SConstr} \\
\mathcal{VC} & ::= & \mathsf{vexp} \sqsubseteq \mathsf{vexp} \mid \mathsf{vexp} = \mathsf{vexp} & \in \mathsf{VConstr} \\
\mathcal{AC} & ::= & \mathsf{ae}_1 \geq \mathsf{ae}_2 \mid \mathsf{ae}_1 \leq \mathsf{ae}_2 & \in \mathsf{AConstr} \\
\mathcal{C} & ::= & \mathcal{AC} \mid \mathcal{SC} \mid \mathcal{VC} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathsf{tt} &
\end{array}
$$

Let $\pi = (\pi_v, \pi_a)$ be a pair of maps: $\pi_v : \mathsf{ViewVars} \to \mathscr{V}^{\mathscr{C}}$ is a map from view variables to views and $\pi_a : \mathsf{NumVars} \to \mathbb{D}$ is a map from number variables to nonnegative real numbers or infinity. We define *the meaning of arithmetic expressions* $\pi(\mathsf{ae})$ inductively as follows.

$$
\begin{array}{lcl}
\pi(p) & = & \pi_a(p) \\
\pi(n) & = & n \\
\pi(\Diamond(C^v)) & = & \Diamond\big(C^{\pi_v(v)}\big) \\
\pi(\mathsf{ae}_1 + \mathsf{ae}_2) & = & \pi(\mathsf{ae}_1) + \pi(\mathsf{ae}_2)
\end{array}
$$

The meaning of view expressions $\pi(\mathsf{vexp})$ is defined as follows.

$$
\begin{array}{lcl}
\pi(v) & = & \pi_v(v) \\
\pi(\mathsf{A}^{\mathrm{get}}(C^v, a)) & = & \mathsf{A}^{\mathrm{get}}\big(C^{\pi_v(v)}, a\big) \\
\pi(\mathsf{A}^{\mathrm{set}}(C^v, a)) & = & \mathsf{A}^{\mathrm{set}}\big(C^{\pi_v(v)}, a\big) \\
\pi(v \oplus v') & = & \pi_v(v) \oplus \pi_v(v')
\end{array}
$$

The meaning of constraints $\pi \models \mathcal{C}$ is defined as follows.

$$
\begin{array}{lll}
\pi \models C^{\mathsf{vexp}_1} <: D^{\mathsf{vexp}_2} & \text{iff} & C <: D \ \text{ and } \ \pi(\mathsf{vexp}_1) \sqsubseteq \pi(\mathsf{vexp}_2) \\
\pi \models \mathsf{vexp}_1 \sqsubseteq \mathsf{vexp}_2 & \text{iff} & \pi(\mathsf{vexp}_1) \sqsubseteq \pi(\mathsf{vexp}_2) \\
\pi \models \mathsf{vexp}_1 = \mathsf{vexp}_2 & \text{iff} & \pi(\mathsf{vexp}_1) \sqsubseteq \pi(\mathsf{vexp}_2) \ \text{ and } \ \pi(\mathsf{vexp}_2) \sqsubseteq \pi(\mathsf{vexp}_1) \\
\pi \models \mathsf{ae}_1 \geq \mathsf{ae}_2 & \text{iff} & \pi(\mathsf{ae}_1) \geq \pi(\mathsf{ae}_2) \\
\pi \models \mathsf{ae}_1 \leq \mathsf{ae}_2 & \text{iff} & \pi(\mathsf{ae}_1) \leq \pi(\mathsf{ae}_2) \\
\pi \models \mathcal{C} \wedge \mathcal{D} & \text{iff} & \pi \models \mathcal{C} \text{ and } \pi \models \mathcal{D} \\
\pi \models \mathsf{tt} & &
\end{array}
$$

If $\pi = (\pi_v, \pi_a)$ is a pair of maps, we write $\mathrm{dom}(\pi)$ for meaning $\mathrm{dom}(\pi_v) \cup \mathrm{dom}(\pi_a)$. Moreover, if $\pi' = (\pi'_v, \pi'_a)$, we write $\pi\pi'$ for meaning $(\pi_v\pi'_v, \pi_a\pi'_a)$, when $\mathrm{dom}(\pi) \cap \mathrm{dom}(\pi') = \emptyset$. Recall that for two maps f, g we write fg for

the union of the maps when $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$. Further, if $\bar{\pi} = (\bar{\pi}_v, \bar{\pi}_a)$ with $\mathrm{dom}(\pi) \supseteq \mathrm{dom}(\bar{\pi})$, then we write $\pi|_{\mathrm{dom}(\bar{\pi})}$ for meaning the pair of maps $(\pi_v|_{\mathrm{dom}(\bar{\pi}_v)}, \pi_a|_{\mathrm{dom}(\bar{\pi}_a)})$.

While constraints are quantifier free we will often need quantification over views in side conditions. To ease notation we allow quantification over view variables in such situations which is understood as quantifying over views. For example $\forall w \,.\, \mathcal{D}(w) \Rightarrow \exists v \,.\, \mathcal{C}(w, v)$ means: for all valuations π with $\pi \models \mathcal{D}$, there exists a valuation π' with $\pi'(w) = \pi(w)$ and $\pi' \models \mathcal{C}$.

We write $\mathcal{C}(\vec{v}, \vec{q})$ for meaning that the conjunction of constraints $\mathcal{C}$ contains the variables $\vec{v} \in \mathsf{ViewVars}$ and $\vec{q} \in \mathsf{NumVars}$.

3.2.3 Monomorphic and polymorphic RAJA method types

Now that we have defined views, refined types and subtyping and arithmetic constraints, we are ready to define the RAJA method types.

If $\vec{v} = v_0, \ldots, v_{n+1}$ is a vector of length $n+2$, with $n \geq 0$, we write $\vec{\mathfrak{v}}$ for meaning the (possibly empty) vector $v_1, \ldots, v_n$.

Definition 3.2.11 (An *n-ary monomorphic RAJA method type*) *An* n-ary monomorphic RAJA method type T *consists of* $n+2$ *views* $\vec{s}$ *and two numbers* m_1, m_2 *written* $T = s_0; \vec{\mathfrak{s}} \xrightarrow{m_1/m_2} s_{n+1}$.

We call the set of monomorphic RAJA method types $\mathsf{MonoType}$. We also write $C^{s_0}; \vec{E}^{\vec{\mathfrak{s}}} \xrightarrow{m_1/m_2} H^{s_{n+1}}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

Definition 3.2.12 (An *n-ary polymorphic RAJA method type*) *An* n-ary polymorphic RAJA method type ϕ *consists of* $n+2$ *view variables* $\vec{v}$ *and two arithmetic variables* $\vec{q} = q_1, q_2$ *and existentially quantified (view and arithmetic) variables* $\vec{w}$, $\vec{t}$ *and a conjunction of subtyping and arithmetic constraints on them written*

$$\phi = \forall \vec{v}, \vec{q} \; \exists \, \vec{w}, \vec{t} \,.\, v_0; \vec{\mathfrak{v}} \xrightarrow{q_1/q_2} v_{n+1} \;\&\; \mathcal{C}(\vec{v}, \vec{q}, \vec{w}, \vec{t})$$

We call the set of polymorphic RAJA method types $\mathsf{PolyType}$. We often write

$$\forall \vec{v}, \vec{q} \; \exists \, \vec{w}, \vec{t} \,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{q_1/q_2} H^{v_{n+1}} \;\&\; \mathcal{C}(\vec{v}, \vec{q}, \vec{w}, \vec{t})$$

to denote an FJEU method type combined with a corresponding polymorphic RAJA method type.

A polymorphic method type stands for the set of all monomorphic types that satisfy its constraints. Because this type does not depend on the method's callers, the resource analysis for the method can be performed modularly.

Definition 3.2.13 (Instance of a polymorphic method type)
Let $T = C^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{m_1/m_2} H^{s_{n+1}}$ *be a monomorphic RAJA method type and* $\forall \vec{v}, \vec{q}\ \exists\, \vec{w}, \vec{t}\,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{q_1/q_2} H^{v_{n+1}}\ \&\ \mathcal{C}(\vec{v}, \vec{q}, \vec{w}, \vec{t})$ *a polymorphic RAJA method type. We say that* T *is an instance of* ϕ*, written: "*T $\mathsf{instanceof}$ ϕ*" iff there exists a valuation* π *with* $\pi \models \mathcal{C}$ *such that* $\pi(v_i) = s_i$ *for* $i \in \{0, \ldots, n+1\}$ *and* $\pi(q_j) = m_j$ *for* $j \in \{1, 2\}$*.*

Definition 3.2.14 (RAJA program)
A RAJA program *is an annotation of an FJEU program* $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ *in the form of a tuple* $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *where* M *assigns to each class* C *and method* $m \in \mathsf{Meth}(C)$ *with* n *arguments an* n*-ary polymorphic RAJA method type* $\mathsf{M}(C, m)$*.*

We write $\mathsf{constr}(\phi)$ for the conjunction of constraints of a polymorphic method type ϕ, i.e.

$$\mathsf{constr}(\phi) = \mathcal{C} \quad \text{if} \quad \phi = \forall \vec{v}, \vec{q} \exists\, \vec{w}, \vec{t}\,.\, v_0; \vec{\mathfrak{v}} \xrightarrow{q_1/q_2} v_{n+1}\ \&\ \mathcal{C}$$

We define trivial polymorphic RAJA method types with no constraints for a given class C and method m, called $\top^{(C,m)}$, by:

$$\top^{(C,m)} = \forall \vec{v}, \vec{q}\,.\, v_0; \vec{\mathfrak{v}} \xrightarrow{q_1/q_2} v_{n+1}\ \&\ \mathsf{tt}$$

Definition 3.2.15 (Subtyping of monomorphic method types)
If $T = \vec{r} \xrightarrow{n_1/n_2} r_{n+1}$ *and* $T' = \vec{s} \xrightarrow{m_1/m_2} s_{n+1}$ *then* $T <: T'$ *is defined as* $n_1 \leq m_1$ *and* $n_2 \geq m_2$ *and* $r_0 = s_0$ *and* $s_i \sqsubseteq r_i$ *for* $i = 1, \ldots, n$ *and* $r_{n+1} \sqsubseteq s_{n+1}$*.*

Definition 3.2.16 (Subtyping of polymorphic method types)
Let $C, D, \vec{E}, H \in \mathscr{C}$ *with* $C <: D$ *and let* ϕ *and* ψ *be polymorphic RAJA method types refining the FJEU method type* $\vec{E} \to H$ *of method* m *in class* C *and* D*, respectively. Then* $\phi <: \psi$ *iff:*

$$\forall T' \text{ with } T' \ \mathsf{instanceof}\ \psi\,.\, \exists T \text{ with } T \ \mathsf{instanceof}\ \phi \text{ such that } T <: T' \quad (3.2.5)$$

We call a polymorphic RAJA method type *empty* if its constraints are unsatisfiable and *nonempty* if they can be satisfied.

Fact 3.2.17 (Properties of subtyping of RAJA method types)

1. *If* $\phi <: \psi$ *and* ψ *is nonempty then* ϕ *is nonempty as well.*
2. *If* $\phi <: \psi$ *and* ϕ *is empty then* ψ *must be empty as well.*

Definition 3.2.18 ($\phi \vee \psi$)
Let $C, D, \vec{E}, H \in \mathscr{C}$ with $C <: D$ or $D <: C$ and let ϕ and ψ be polymorphic RAJA method types refining the FJEU method type $\vec{E} \to H$ of method m in class C and D, respectively.

$$\overbrace{\forall \vec{v}, \vec{p} \exists \vec{w}, \vec{t} \,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; \mathcal{C}}^{\phi =} \quad \overbrace{\forall \vec{v'}, \vec{p'} \,\exists \vec{w'}, \vec{t'} \,.\, D^{v'_0}; \vec{E}^{\vec{\mathfrak{v}}'} \xrightarrow{p'_1/p'_2} H^{v'_{n+1}} \;\&\; \mathcal{D}}^{\psi =}$$

Then, we define $\phi \vee \psi$ by:

$$\begin{aligned}
&\phi \vee \psi = \forall \vec{v}, \vec{v'}, \vec{u}, \vec{p}, \vec{p'}, \vec{q} \exists \vec{w}, \vec{w'}, \vec{t}, \vec{t'} \,.\, (C \vee D)^{u_0}; \vec{E}^{\vec{u}} \xrightarrow{t_1/t_2} H^{u_{n+1}} \;\&\; \mathcal{E} \quad \textit{where} \\
&\mathcal{E} = \mathcal{C}\,[u_0/v_0] \;\wedge\; \mathcal{D}\,[u_0/v'_0] \;\wedge\; \bigwedge_i (u_i \sqsubseteq v_i \;\wedge\; u_i \sqsubseteq v'_i) \;\wedge\; (v_{n+1} \sqsubseteq u_{n+1}) \\
&\quad \wedge\; (v'_{n+1} \sqsubseteq u_{n+1}) \;\wedge\; (q_1 \geq p_1) \;\wedge\; (q_1 \geq p'_1) \;\wedge\; (q_2 \leq p_2) \;\wedge\; (q_2 \leq p'_2)
\end{aligned}$$

Lemma 3.2.19 (l.u.b. of RAJA polymorphic method types)
Let ϕ, ψ polymorphic RAJA method types. Then $\phi \vee \psi$ is the least upper bound *of ϕ and ψ, i.e.*

1. *$\phi <: (\phi \vee \psi)$ and $\psi <: (\phi \vee \psi)$.*
2. *$\forall \xi$ with $\phi <: \xi$ and $\psi <: \xi$ holds $(\phi \vee \psi) <: \xi$.*

Proof. Let

$$\overbrace{\forall \vec{v}, \vec{p} \exists \vec{w}\,, \vec{t}\,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; \mathcal{C}}^{\phi =} \quad \overbrace{\forall \vec{u}, \vec{q} \exists \vec{w'}\,, \vec{t'}\,.\, D^{v'_0}; \vec{E}^{\vec{\mathfrak{v}}'} \xrightarrow{p'_1/p'_2} H^{v'_{n+1}} \;\&\; \mathcal{D}}^{\psi =}$$

and let $\phi \vee \psi$ be defined as in Def. 3.2.18:

$$\begin{aligned}
&\phi \;\vee\; \psi = \forall \vec{v}, \vec{v'}, \vec{u}, \vec{p}, \vec{p'}, \vec{q} \exists \vec{w}, \vec{w'}, \vec{t}, \vec{t'} \,.\, (C \vee D)^{u_0}; \vec{E}^{\vec{u}} \xrightarrow{t_1/t_2} H^{u_n} \;\&\; \mathcal{E} \quad \text{where} \\
&\mathcal{E} = \mathcal{C}\,[u_0/v_0] \;\wedge\; \mathcal{D}\,[u_0/v'_0] \;\wedge\; \bigwedge_i (u_i \sqsubseteq v_i \;\wedge\; u_i \sqsubseteq v'_i) \;\wedge\; (v_{n+1} \sqsubseteq u_{n+1}) \\
&\quad \wedge\; (v'_{n+1} \sqsubseteq u_{n+1}) \;\wedge\; (q_1 \geq p_1) \;\wedge\; (q_1 \geq p'_1) \;\wedge\; (q_2 \leq p_2) \;\wedge\; (q_2 \leq p'_2)
\end{aligned}$$

1. We show $\phi <: (\phi \vee \psi)$, in particular we show

$$\begin{aligned}
\forall\, \vec{u}\,, \vec{q} \exists \vec{w'}\,, \vec{t'}\,.\, \mathcal{E} \implies \exists\, \vec{v}\,, \vec{p}\,, \vec{w}\,, \vec{t}\,.\, \mathcal{C} \;\wedge\; v_0 = u_0 \;\wedge \\
\bigwedge_{i=1..n} u_i \sqsubseteq v_i \;\wedge\; v_{n+1} \sqsubseteq u_{n+1} \;\wedge\; p_1 \leq q_1 \;\wedge\; p_2 \geq q_2 \quad (3.2.6)
\end{aligned}$$

   which follows clearly when we substitute $\mathcal{E}$ by its definition. $\psi <: (\phi \vee \psi)$ follows just as trivially.

2. Let $\xi = \forall \vec{o}, \vec{l} \exists \vec{\hat{w}}, \vec{\hat{t}}\,.\, F^{o_0}; \vec{E}^{\vec{\sigma}\, l_1/l_2}\!\!\longrightarrow H^{o_{n+1}}$ & $\mathcal{G}$. By assumption we have $C <: F$ and $D <: F$ and :

$$\phi <: \xi \iff \forall\, \vec{o}, \vec{l}\ \exists \vec{\hat{w}}, \vec{\hat{t}}\,.\, \mathcal{G} \implies \exists\, \vec{v}, \vec{p} \exists \vec{w}, \vec{t}\,.\, \mathcal{C} \ \wedge\ v_0 = o_0 \ \wedge \bigwedge_{i=1..n} o_i \sqsubseteq v_i \ \wedge\ v_{n+1} \sqsubseteq o_{n+1} \ \wedge\ p_1 \leq l_1 \ \wedge\ p_2 \geq l_2 \ \text{ and } \quad (3.2.7)$$

$$\psi <: \xi \iff \forall\, \vec{o}, \vec{l}\ \exists \vec{\hat{w}}, \vec{\hat{t}}\,.\, \mathcal{G} \implies \exists\, \vec{v'}, \vec{p'} \exists \vec{w'}, \vec{t'}\,.\, \mathcal{D} \ \wedge\ v'_0 = o_0 \ \wedge \bigwedge_{i=1..n} o_i \sqsubseteq v'_i \ \wedge\ v'_{n+1} \sqsubseteq o_{n+1} \ \wedge\ p'_1 \leq l_1 \ \wedge\ p'_2 \geq l_2 \quad (3.2.8)$$

We show $\phi \vee \psi <: \xi \iff$

$$\forall\, \vec{o}, \vec{l}\ \exists \vec{\hat{w}}, \vec{\hat{t}}\,.\, \mathcal{G} \implies \exists\, \vec{u}, \vec{q}, \vec{v}, \vec{v'}, \vec{p}, \vec{p'}, \vec{w}, \vec{t}, \vec{w'}, \vec{t'}\,.\, \mathcal{E} \ \wedge\ u_0 = o_0 \ \wedge \bigwedge_{i=1..n} o_i \sqsubseteq u_i \ \wedge\ u_{n+1} \sqsubseteq o_{n+1} \ \wedge\ q_1 \leq l_1 \ \wedge\ q_2 \geq l_2 \quad (3.2.9)$$

Let $\pi \models \mathcal{G}$ with $\pi(o_i) = r_i$ and $\pi(l_i) = m_i$ and let $i \in \{1, \ldots, n\}$. Then, by (3.2.7), there exists π_1 with $\pi_1|_{\mathrm{dom}(\pi)} = \pi$ and $\pi_1(v_i) = s_i$, $\pi_1(p_i) = n_i$ with $\pi_1 \models \mathcal{C}$ and $r_i \sqsubseteq s_i$ and $s_{n+1} \sqsubseteq r_{n+1}$ and $n_1 \leq m_1$ and $n_2 \geq m_2$.

Moreover, by (3.2.8), there exists π_2 with $\pi_2|_{\mathrm{dom}(\pi)} = \pi$ and $\pi_2(u_i) = \bar{s}_i, \pi_2(p_i) = \bar{n}_i$ with $\pi_2 \models \mathcal{D}$ and $r_i \sqsubseteq \bar{s}_i$ and $\bar{s}_{n+1} \sqsubseteq r_{n+1}$ and $\bar{n}_1 \leq m_1$ and $\bar{n}_2 \geq m_2$. Then (3.2.9) follows with $\tilde{\pi} = \pi_1\, \pi_2\, \pi_3$ where:

$$\pi_3 = \{u_0 \mapsto s_0, u_1 \mapsto s_1 \wedge \bar{s}_1, \ldots, u_n \mapsto s_n \wedge \bar{s}_n, u_{n+1} \mapsto s_{n+1} \vee \bar{s}_{n+1}\}, \{t_1 \mapsto \mathsf{max}(n_1, \bar{n}_1), t_2 \mapsto \mathsf{min}(n_2, \bar{n}_2)\}$$

since $s_i \wedge \bar{s}_i \sqsubseteq s_i$ and $s_i \wedge \bar{s}_i \sqsubseteq \bar{s}_i$ and $r_i \sqsubseteq s_i \wedge \bar{s}_i$ because $s_i \wedge \bar{s}_i$ is the g.l.b. of s_i and $\bar{s}_i$ by Corollary 3.2.5.

Moreover, since $s_{n+1} \vee \bar{s}_{n+1}$ is the l.u.b. of s_{n+1} and $\bar{s}_{n+1}$ by Corollary 3.2.5, $s_{n+1} \vee \bar{s}_{n+1} \sqsubseteq r_{n+1}$ and $s_{n+1} \sqsubseteq s_{n+1} \vee \bar{s}_{n+1}$ and $\bar{s}_{n+1} \sqsubseteq s_{n+1} \vee \bar{s}_{n+1}$. Further, we show:

(a) $\mathsf{max}(n_1, \bar{n}_1) \geq n_1$ and $\mathsf{max}(n_1, \bar{n}_1) \geq \bar{n}_1$ which follows trivially.

(b) $\mathsf{min}(n_2, \bar{n}_2) \leq n_2$ and $\mathsf{min}(n_2, \bar{n}_2) \leq \bar{n}_2$ which follows trivially.

(c) $\mathsf{max}(n_1, \bar{n}_1) \leq m_1$ which follows from $n_1 \leq m_1$ and $\bar{n}_1 \leq m_1$.

(d) $\mathsf{min}(n_2, \bar{n}_2) \geq m_2$ which follows from $n_2 \geq m_2$ and $\bar{n}_2 \geq m_2$.

□

3.2.4 Sharing Relation

In a RAJA program, if a variable is to be used more than once, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available for that variable. We introduce the *sharing relation* $\curlyvee(r \,|s_1, \ldots, s_n)$ for splitting the potential of a view r among a multiset of views $s_1, \ldots, s_n$. For example, if we have $l : \mathsf{List}^{\mathsf{rich}}$ we can use the variable l with the types $\mathsf{List}^{\mathsf{s}_1}$ and $\mathsf{List}^{\mathsf{s}_2}$ if $\curlyvee(\mathsf{rich} \,|s_1, s_2)$ holds.

The relation we define here is stronger than the sharing relation from the original RAJA system [HJ06]. With the new relation we are able to identify the sharing relation with the subtyping relation, since we can prove $\curlyvee(r \,|s_1, s_2)$ is equivalent to $r \sqsubseteq s_1 \oplus s_2$. This way, we can concentrate on subtyping when describing type checking and type inference algorithms for RAJA in the following chapters.

Definition 3.2.20 (Sharing Relation)
We define the sharing relation *between a single view r and a multiset of views $\mathcal{D}$ written $\curlyvee(r \,|\mathcal{D})$ simultaneously with the relation $\curlywedge(r \,|\mathcal{D})$ coinductively by*

1. *if $\curlyvee(r \,|\mathcal{D})$ then for all $C \in \mathscr{C}$:*

$$\Diamond\!\!\!\!\!\curlyvee(C^r) \geq \sum_{s \in \mathcal{D}} \Diamond\!\!\!\!\!\curlyvee(C^s) \tag{3.2.10}$$

$$\forall s \in \mathcal{D} \,.\, r \sqsubseteq s \tag{3.2.11}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C)) \,.\, \curlyvee\big(\mathsf{A}^{\mathrm{get}}(C^r, a) \,\big|\, \mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big) \tag{3.2.12}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C)) \,.\, \curlywedge\big(\mathsf{A}^{\mathrm{set}}(C^r, a) \,\big|\, \mathsf{A}^{\mathrm{set}}\big(C^{\mathcal{D}}, a\big)\big) \tag{3.2.13}$$

2. *if $\curlywedge(r \,|\mathcal{D})$ then for all $C \in \mathscr{C}$:*

$$\min_{s \in \mathcal{D}} \Diamond\!\!\!\!\!\curlyvee(C^s) \geq \Diamond\!\!\!\!\!\curlyvee(C^r) \tag{3.2.14}$$

$$\forall s \in \mathcal{D} \,.\, s \sqsubseteq r \tag{3.2.15}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C)) \,.\, \curlywedge\big(\mathsf{A}^{\mathrm{get}}(C^r, a) \,\big|\, \mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big) \tag{3.2.16}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C)) \,.\, \curlyvee\big(\mathsf{A}^{\mathrm{set}}(C^r, a) \,\big|\, \mathsf{A}^{\mathrm{set}}\big(C^{\mathcal{D}}, a\big)\big) \tag{3.2.17}$$

where $\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big) = \{\mathsf{A}^{\mathrm{get}}(C^s, a) \mid s \in \mathcal{D}\}$, $\mathsf{A}^{\mathrm{set}}\big(C^{\mathcal{D}}, a\big) = \{\mathsf{A}^{\mathrm{set}}(C^s, a) \mid s \in \mathcal{D}\}$.

If $\mathcal{D} = \{s_1, \ldots, s_i\}$ is a finite multiset, we also write $\curlyvee(r \,|s_1, \ldots, s_i)$ for $\curlyvee(r \,|\mathcal{D})$. In the following we prove several properties of the sharing relation.

Lemma 3.2.21 *Let $r \in \mathscr{V}^{\mathscr{C}}$. Then:*

1. $\curlyvee(r\,|\{r\}\,)$
2. $\curlywedge(r\,|\{r\}\,)$

Proof. Simultaneously by coinduction. □

Lemma 3.2.22 *Let $r \in \mathscr{V}^{\mathscr{C}}$ and $\mathcal{D}$ be a multiset $\mathcal{D} \subseteq \mathscr{V}^{\mathscr{C}}$. Then:*

1. *If $\curlyvee(r\,|\mathcal{D})$ then $\forall \mathcal{E} \subseteq \mathcal{D}\,.\,\curlyvee(r\,|\mathcal{E})$.*
2. *If $\curlywedge(r\,|\mathcal{D})$ then $\forall \mathcal{E} \subseteq \mathcal{D}\,.\,\curlywedge(r\,|\mathcal{E})$.*

Proof. Simultaneously by coinduction.

1. (3.2.10) follows by $\Diamond\!(C^r) \geq \sum_{s\in\mathcal{D}} \Diamond\!(C^s) \geq \sum_{s\in\mathcal{E}} \Diamond\!(C^s)$. (3.2.11) follows by assumption. (3.2.12) follows by the co. hyp. (1) and (3.2.13) follows by the co. hyp. (2).
2. (3.2.14) follows by $\min_{s\in\mathcal{E}} \Diamond\!(C^s) \geq \min_{s\in\mathcal{D}} \Diamond\!(C^s) \geq \Diamond\!(C^r)$. (3.2.15) follows by assumption. (3.2.16) follows by the co. hyp. (2) and (3.2.17) follows by the co. hyp. (1).

□

Lemma 3.2.23 *Let $r, s \in \mathscr{V}^{\mathscr{C}}$ and $\mathcal{D}, \mathcal{E}$ be multisets $\mathcal{D} \subseteq \mathscr{V}^{\mathscr{C}}$ and $\mathcal{E} \subseteq \mathscr{V}^{\mathscr{C}}$. Then:*

1. $\curlyvee(r\,|\mathcal{D} \cup \{s\}\,) \wedge \curlyvee(s\,|\mathcal{E}\,) \implies \curlyvee(r\,|\mathcal{D} \cup \mathcal{E}\,)$
2. $\curlywedge(r\,|\mathcal{D} \cup \{s\}\,) \wedge \curlywedge(s\,|\mathcal{E}\,) \implies \curlywedge(r\,|\mathcal{D} \cup \mathcal{E}\,)$

Proof. Simultaneously by coinduction.

1. (3.2.10), $\Diamond\!(C^r) \geq \sum_{s_{\mathcal{D}}\in\mathcal{D}} \Diamond\!(C^{s_{\mathcal{D}}}) + \sum_{s_{\mathcal{E}}\in\mathcal{E}} \Diamond\!(C^{s_{\mathcal{E}}})$ follows by $\Diamond\!(C^r) \geq \sum_{s_{\mathcal{D}}\in\mathcal{D}} \Diamond\!(C^{s_{\mathcal{D}}}) + \Diamond\!(C^s)$ and $\Diamond\!(C^s) \geq \sum_{s_{\mathcal{E}}\in\mathcal{E}} \Diamond\!(C^{s_{\mathcal{E}}})$ with transitivity.

   (3.2.11), $r \sqsubseteq s_{\mathcal{D}}$ follows by assumption and $r \sqsubseteq s_{\mathcal{E}}$ follows by $r \sqsubseteq s$ and $s \sqsubseteq s_{\mathcal{E}}$, which hold by assumption, and transitivity.

   (3.2.12) follows by the co. hyp. (1) and (3.2.13) by the co. hyp. (2).
2. (3.2.14) We show

$$\min_{s'\in\mathcal{D}\cup\mathcal{E}} \Diamond\Big(C^{s'}\Big) \geq \Diamond\!(C^r) \tag{3.2.18}$$

   By assumption we have

$$\min_{s'\in\mathcal{D}\cup\{s\}} \Diamond\Big(C^{s'}\Big) \geq \Diamond\!(C^r) \tag{3.2.19}$$

$$\min_{s''\in\mathcal{E}} \Diamond\Big(C^{s''}\Big) \geq \Diamond\!(C^s) \tag{3.2.20}$$

   We obtain the following two cases:

Case $\Diamond\!\!\!\!\diagup(C^s) \leq \min_{s' \in \mathcal{D}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right)$, i.e. $\min_{s' \in \mathcal{D} \cup \{s\}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right) = \Diamond\!\!\!\!\diagup(C^s)$.

Then, we get $\underbrace{\min_{s' \in \mathcal{D}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right)}_{n} \geq \Diamond\!\!\!\!\diagup(C^r)$ by (3.2.19) and transitivity.

Moreover, we get $\underbrace{\min_{s' \in \mathcal{E}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right)}_{m} \geq \Diamond\!\!\!\!\diagup(C^r)$ (3.2.20) and transitivity

and the goal follows since $\min_{s'' \in \mathcal{D} \cup \mathcal{E}} \Diamond\!\!\!\!\diagup\left(C^{s''}\right)$ is either n or m.

Case $\Diamond\!\!\!\!\diagup(C^s) \geq \min_{s' \in \mathcal{D}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right)$, i.e. $\min_{\bar{s} \in \mathcal{D} \cup \{\bar{s}\}} \Diamond\!\!\!\!\diagup(C^{\bar{s}}) = \min_{\bar{s} \in \mathcal{D}} \Diamond\!\!\!\!\diagup(C^{\bar{s}})$.

Then, by (3.2.20), $\min_{s' \in \mathcal{D} \cup \mathcal{E}} \Diamond\!\!\!\!\diagup\left(C^{s'}\right) = \min_{\bar{s} \in \mathcal{D}} \Diamond\!\!\!\!\diagup(C^{\bar{s}})$, thus, the goal follows by (3.2.19).

(3.2.15) follows by assumption and transitivity. (3.2.16) follows by the co. hyp. (2) and (3.2.17) follows by the co. hyp. (1).

□

Lemma 3.2.24 *Let $r, r', s, s' \in \mathscr{V}^{\mathscr{C}}$ and $\mathcal{D}$ be a multiset $\mathcal{D} \subseteq \mathscr{V}^{\mathscr{C}}$.*

1. $r' \sqsubseteq r \wedge s' \sqsubseteq s \wedge \curlyvee(r \,|\mathcal{D} \cup \{s'\}) \implies \curlyvee(r' \,|\mathcal{D} \cup \{s\})$
2. $r \sqsubseteq r' \wedge s \sqsubseteq s' \wedge \curlywedge(r \,|\mathcal{D} \cup \{s'\}) \implies \curlywedge(r' \,|\mathcal{D} \cup \{s\})$

Proof. Simultaneously by coinduction. Let $s_i \in \mathcal{D}$.

1. (3.2.10) follows by $\Diamond\!\!\!\!\diagup\left(C^{r'}\right) \geq \Diamond\!\!\!\!\diagup(C^r)$ and $\Diamond\!\!\!\!\diagup\left(C^{s'}\right) \geq \Diamond\!\!\!\!\diagup(C^s)$ and $\Diamond\!\!\!\!\diagup(C^r) \geq \sum_{s_i \in \mathcal{D}} \Diamond\!\!\!\!\diagup(C^{s_i}) + \Diamond\!\!\!\!\diagup\left(C^{s'}\right)$ which follow by assumption, and transitivity.

   (3.2.11) follows by $r' \sqsubseteq r \sqsubseteq s_i$ and $r' \sqsubseteq r \sqsubseteq s' \sqsubseteq s$, which follow by assumption, and transitivity. (3.2.12) follows by the co. hyp. (1) and (3.2.13) follows by the co. hyp. (2).

2. (3.2.14) We show $\min_{s_i \in \mathcal{D} \cup \{s\}} \Diamond\!\!\!\!\diagup(C^{s_i}) \geq \Diamond\!\!\!\!\diagup\left(C^{r'}\right)$ and we have

$$\Diamond\!\!\!\!\diagup(C^s) \geq \Diamond\!\!\!\!\diagup\left(C^{s'}\right) \tag{3.2.21}$$

$$\Diamond\!\!\!\!\diagup(C^r) \geq \Diamond\!\!\!\!\diagup\left(C^{r'}\right) \tag{3.2.22}$$

$$\min_{s_i \in \mathcal{D} \cup \{s'\}} \Diamond\!\!\!\!\diagup(C^{s_i}) \geq \Diamond\!\!\!\!\diagup(C^r) \tag{3.2.23}$$

   By transitivity and (3.2.22) and (3.2.23) we obtain

$$\min_{s_i \in \mathcal{D} \cup \{s'\}} \Diamond\!\!\!\!\diagup(C^{s_i}) \geq \Diamond\!\!\!\!\diagup\left(C^{r'}\right) \tag{3.2.24}$$

   Then, there are two cases:

Case
$$\Diamond\Big(C^{s'}\Big) \leq \min_{s_i \in \mathcal{D}} \Diamond(C^{s_i}) \tag{3.2.25}$$
$$\text{We get } \Diamond\Big(C^{s'}\Big) \geq \Diamond\Big(C^{r'}\Big) \tag{3.2.26}$$

by (3.2.24). Then, if $\min_{s_i \in \mathcal{D} \cup \{s\}} \Diamond(C^{s_i}) = \Diamond(C^s)$, we finish by (3.2.21) and (3.2.26) with transitivity. Otherwise, the goal follows by (3.2.25) and (3.2.26) with transitivity.

Case
$$\Diamond\Big(C^{s'}\Big) \geq \min_{s_i \in \mathcal{D}} \Diamond(C^{s_i}) \tag{3.2.27}$$
$$\text{We get } \min_{s_i \in \mathcal{D}} \Diamond(C^{s_i}) \geq \Diamond\Big(C^{r'}\Big) \tag{3.2.28}$$

by (3.2.24). Then, $\min_{s_i \in \mathcal{D} \cup \{s\}} \Diamond(C^{s_i}) = \min_{s_i \in \mathcal{D}} \Diamond(C^{s_i})$ by (3.2.21) and (3.2.27) and the goal follows.

(3.2.15) follows by $s_i \sqsubseteq r \sqsubseteq r'$ and $s \sqsubseteq s' \sqsubseteq r \sqsubseteq r'$, which follow by assumption, and transitivity. (3.2.16) follows by the co. hyp. (2) and (3.2.17) follows by the co. hyp. (1).

□

Lemma 3.2.25 *Let $r, s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$. Then*

1. $\curlyvee(r \,|s_1, s_2) \iff r \sqsubseteq s_1 \oplus s_2$
2. $\curlywedge(r \,|s_1, s_2) \iff s_1 \boxplus s_2 \sqsubseteq r$

Proof. Simultaneously by coinduction.

1. *Case* "⇒" We have $\curlyvee(r \,|s_1, s_2)$ and we show $r \sqsubseteq s_1 \oplus s_2$. In particular, we show (3.2.1), (3.2.2) and (3.2.3).
   (a) (3.2.1) follows by assumption.
   (b) (3.2.2) follows by $\curlyvee\big(\mathsf{A}^{\mathrm{get}}(C^r, a) \,\big|\mathsf{A}^{\mathrm{get}}(C^{s_1}, a), \mathsf{A}^{\mathrm{get}}(C^{s_2}, a)\big)$ implies $\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a)$, which follows by the coinduction hypothesis (1).
   (c) (3.2.3) follows by $\curlywedge\big(\mathsf{A}^{\mathrm{set}}(C^r, a) \,\big|\mathsf{A}^{\mathrm{set}}(C^{s_1}, a), \mathsf{A}^{\mathrm{set}}(C^{s_2}, a)\big)$ implies $\mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a)$, which follows by the coinduction hypothesis (2).

   Case "⇐" We have $r \sqsubseteq s_1 \oplus s_2$ and we show $\curlyvee(r \,|s_1, s_2)$. In particular, we show (3.2.10), (3.2.11), (3.2.12) and (3.2.13).
   (a) (3.2.10) follows by assumption.
   (b) (3.2.11) follows by Lemma 3.2.7.
   (c) (3.2.12) follows by the coinduction hypothesis (1).
   (d) (3.2.13) follows by the coinduction hypothesis (2).

2. *Case* "⇒" We have $\lambda\!\!\!\lambda(r\,|s_1, s_2\,)$ and we show $s_1 \boxplus s_2 \sqsubseteq r$. In particular, we show (3.2.1), (3.2.2) and (3.2.3).
   (a) (3.2.1) follows by assumption.
   (b) (3.2.2) follows by $\lambda\!\!\!\lambda\big(\mathsf{A}^{\mathrm{get}}(C^r, a)\,\big|\mathsf{A}^{\mathrm{get}}(C^{s_1}, a)\,, \mathsf{A}^{\mathrm{get}}(C^{s_2}, a)\big)$ implies $\mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^r, a)$, which follows by the coinduction hypothesis (2).
   (c) (3.2.3) follows by $\curlyvee\big(\mathsf{A}^{\mathrm{set}}(C^r, a)\,\big|\mathsf{A}^{\mathrm{set}}(C^{s_1}, a)\,, \mathsf{A}^{\mathrm{set}}(C^{s_2}, a)\big)$ implies $\mathsf{A}^{\mathrm{set}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a)$, which follows by the coinduction hypothesis (1).

   Case "⇐" We have $s_1 \boxplus s_2 \sqsubseteq r$ and we show $\lambda\!\!\!\lambda(r\,|s_1, s_2\,)$. In particular, we show (3.2.14), (3.2.15), (3.2.16) and (3.2.17).
   (a) (3.2.14) follows by assumption.
   (b) (3.2.15) follows by Lemma 3.2.7.
   (c) (3.2.16) follows by the coinduction hypothesis (2).
   (d) (3.2.17) follows by the coinduction hypothesis (1).

□

3.2.5 Typing RAJA

The RAJA-typing judgement is formally defined by the rules in Fig. 3.4. The type system allows us to derive assertions of the form $\Gamma \vdash^{n}_{n'} e : C^r$ where e is an expression or program phrase, C is an FJEU class, r is a view (so C^r is a refined type) and Γ is a RAJA context, i.e. a map from variables to refined types. Finally n, n' are nonnegative real numbers. The meaning of such a judgement is as follows. If e terminates successfully in some environment η and heap σ with unbounded memory resources available then it will also terminate successfully with a bounded freelist of size at least n plus the potential ascribed to η, σ with respect to the typing in Γ. Furthermore, the freelist size upon termination will be at least n' plus the potential of the result with respect to the view r.

The typing rules extend the typing rules of FJEU. Notice that the rules $(\Diamond Share)$ and $(\Diamond Waste)$ are not syntax directed. Thus, they need to be eliminated when we come to implement a type checker in Chapter 4. $(\Diamond Waste)$ corresponds to the rule of subsumption of FJEU ($\vdash_{\mathsf{F}} Sub$) and weakens context, type, and effect.

We say that a RAJA context is a subcontext of another, and write $\Gamma <: \Theta$, iff forall $x \in \mathrm{dom}(\Theta)$ holds $\Gamma_x <: \Theta_x$. Notice that, for $\Gamma <: \Theta$ to be defined, $\mathrm{dom}(\Theta) \subseteq \mathrm{dom}(\Gamma)$ must hold. We write $\varnothing$ to denote the empty RAJA context. Then, by definition, $\Gamma <: \varnothing$ for any RAJA context Γ.

The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used more than once without duplication of potential. Recall the copy method from Fig. 3.1 and suppose we have the following expression:

$$\Gamma, l : \mathsf{List}^{\mathsf{rich}} \vdash^{n}_{n'} \mathsf{let}\ nl = l.\mathsf{copy}()\ \mathsf{in}\ l.\mathsf{copy}() : \mathsf{List}^{\mathsf{poor}} \tag{3.2.29}$$

RAJA Typing $\boxed{\Gamma \vdash^{n}_{n'} e : C^r}$

$$\frac{C^r \ \mathsf{main}}{\varnothing \vdash^{\Diamond(C^r)+1}_{0} \ \mathtt{new}\ C : C^r}\ (\Diamond New) \qquad \frac{n' = \min\{\Diamond(D^r) \mid D <: C\}}{x{:}C^r \vdash^{0}_{n'+1} \ \mathtt{free}(x) : E^s}\ (\Diamond Free)$$

$$\frac{C <: E}{x{:}E^r \vdash^{0}_{0} (C)x : C^r}\ (\Diamond Cast) \qquad \frac{}{\varnothing \vdash^{0}_{0} \mathsf{null} : E^s}\ (\Diamond Null)$$

$$\frac{}{x{:}C^r \vdash^{0}_{0} x : C^r}\ (\Diamond Var) \qquad \frac{\forall E_i <: C \,.\, \mathsf{A}^{\mathrm{get}}(E_i^r, a) \sqsubseteq s \quad D = C.a}{x{:}C^r \vdash^{0}_{0} x.a : D^s}\ (\Diamond Access)$$

$$\frac{\forall E_i <: C \,.\, s \sqsubseteq \mathsf{A}^{\mathrm{set}}(E_i^r, a) \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a \texttt{<-} y : C^r}\ (\Diamond Update)$$

$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ D\,x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

$$\frac{\mathsf{M}(C,m) = \phi \quad (C^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{n_1/n_2} H^{s_{n+1}})\ \mathsf{instanceof}\ \phi}{x{:}C^{s_0}, y_1{:}E_1^{s_1}, \ldots, y_n{:}E_n^{s_n} \vdash^{n_1}_{n_2} x.m(\vec{y}) : H^{s_{n+1}}}\ (\Diamond Invocation)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash^{n}_{n'} e_1 : C^r \qquad \Gamma \vdash^{n}_{n'} e_2 : C^r}{\Gamma \vdash^{n}_{n'} \mathtt{if}\ x\ \mathsf{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : C^r}\ (\Diamond Conditional)$$

$$\frac{\curlyvee(s \,|\, s_1, \ldots, s_j) \qquad \Gamma, y_1{:}D^{s_1}, \ldots, y_j{:}D^{s_j} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y_1, \ldots, x/y_j] : C^r}\ (\Diamond Share)$$

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Gamma <: \Theta \quad D^s <: C^r \quad \Theta \vdash^{u}_{u'} e : D^s}{\Gamma \vdash^{n}_{n'} e : C^r}\ (\Diamond Waste)$$

RAJA Method Typing $\boxed{\vdash m : \phi\ \mathsf{ok}}$

$$\frac{\begin{array}{c} m \in \mathsf{Meth}(C) \qquad \phi = \mathsf{M}(C,m) \\ \forall T = (C^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{n_1/n_2} H^{s_{n+1}})\ \mathsf{instanceof}\ \phi \qquad \curlyvee(s_0 \,|\, r_1, r_2) \\ \mathtt{this}{:}C^{r_1}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \vdash^{n_1 + \Diamond(C^{r_2})}_{n_2} \mathsf{M}_{\mathsf{body}}(C,m) : H^{s_{n+1}} \end{array}}{\vdash m : \phi\ \mathsf{ok}}\ (\Diamond MBody)$$

Figure 3.4: RAJA Typing.

We cannot allow that because objects of type $\mathsf{List}^{\mathsf{rich}}$ carry potential to copy themselves just once. If we allowed the second call to the copy method we would be creating objects without "paying" for them, which would be unsound. Since the method copy is only defined for the view rich, the only possibility of typing (3.2.29) would be that we could split the view rich into the views $\mathsf{rich}, \mathsf{rich}$, i.e. $\curlyvee(\mathsf{rich}\,|\mathsf{rich}, \mathsf{rich}\,)$, but that is not possible because $\Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) < \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) + \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big)$.

The judgement $\vdash m : \phi\ \mathsf{ok}$ defined in Fig. 3.4, where m is the name of a method, means that ϕ is a valid polymorphic RAJA method type for m if the method body of m can be typed with the arguments, return type and effects as specified in all instances of ϕ. However, notice that we split the view s_0 specified for this in the concrete instance of ϕ into two views r_1 and r_2, such that $\curlyvee(s_0\,|r_1, r_2\,)$ holds. We then put this in the context with the view r_1 and use the potential of r_2 in the method body. This mechanism allows us to use potential from the variable this for paying for object creations. We cannot use the potential from other variables in similar ways because at compile time it is not known whether the variable references a null pointer.

We shall illustrate this mechanism with an example. In the copy method from Fig. 3.1 we need one item of potential in order to create a $\mathsf{Cons}^{\mathsf{poor}}$ object. We said before that this object creation will be paid for with the potential of this, but how exactly? In order to use the potential of this of refined type $\mathsf{Cons}^{\mathsf{rich}}$, we put it in the context with a modified refined type, say, $\mathsf{Cons}^{\mathsf{s}_1}$. Moreover, we find another view s_2 with potential 1, such that $\curlyvee(\mathsf{rich}\,|s_1, s_2\,)$ holds. Then we can derive:

$$\mathsf{this}\colon \mathsf{Cons}^{\mathsf{s}_1} \vdash^{\frac{1}{0}} \mathsf{let}\,\mathsf{List}\ \mathsf{res} = \mathsf{new}\ \mathsf{Cons}^{\mathsf{poor}}\ \mathsf{in}\ \ \ldots\ \ \mathsf{in}\ \mathsf{return}\ \ \mathsf{res};$$

Notice that in many cases, the set of instances of a polymorphic RAJA method type is infinite, so that the rule ($\Diamond MBody$) cannot be implemented. However, in Chapter 4 we present rules for type checking RAJA programs effectively when we are given a *finite* set of monomorphic RAJA method types for each method. Moreover, in Chapter 5 we give a rule for generating subtyping and arithmetic constraints for methods that we prove sound with respect to this rule.

Programs are well-typed if all method bodies admit the announced polymorphic RAJA method type and this type is nonempty and the type of a method m in a class C is a subtype of the type of the method m in the superclass of C. Formally,

Definition 3.2.26 (Well-typed RAJA-program)
A RAJA-program $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *is* well-typed *if the following conditions are satisfied:*

1. $\forall C \in \mathscr{C}, m \in \mathsf{Meth}(C)\,.\ \vdash m : \mathsf{M}(C, m)\ \mathsf{ok}$
2. $\forall C \in \mathscr{C}, m \in \mathsf{Meth}(C)\,.\,\mathsf{M}(C, m)$ *is nonempty.*

3. $\forall C, D \in \mathscr{C}$ *with* $S(C) = D \Rightarrow \mathsf{M}(C, m) <: \mathsf{M}(D, m)$.

We remark that if an expression e can be typed in the RAJA system, it can also be typed in the FJEU system. If Γ is a RAJA context, we write $|\Gamma|$ for meaning its underlying FJEU context.

Lemma 3.2.27 *If* $\mathcal{D} :: \Gamma \vdash^{n}_{n'} e : C^r$ *then* $|\Gamma| \vdash e : C$.

Proof. By induction on $\mathcal{D}$. □

3.3 Heap soundness and potential

In this section we define a potential function based on RAJA typing and we prove the soundness of the system, which consist of proving that if e can be evaluated successfully in some environment η and heap σ with an unbounded freelist, then it will also evaluate successfully with a bounded freelist of size at least n plus the potential ascribed to η, σ with respect to the typing in Γ.

Recall that in FJEU we assign both a static type $\llbracket (v : C).\vec{p} \rrbracket^{\mathrm{stat}}_{\sigma}$ and a dynamic type $\llbracket v.\vec{p} \rrbracket^{\mathrm{dyn}}_{\sigma}$ to each accessible location in a heap specified by a variable v and an access path $\vec{p}$.

We emphasise that RAJA uses the same runtime model as FJEU, in particular no RAJA typing information will be attached to objects in the heap. Therefore, we will be able to define a static RAJA type in analogy with the static FJEU type, but no dynamic RAJA type will be available. Of course, as in the case of FJEU the static RAJA type of a value and an access path will depend on the heap. In RAJA it will depend on the entire access path and not just on its last component. For example, if v points to a list of length 2 in σ then the static RAJA type of $v.\,\mathsf{next}$ under the view rich is $\mathsf{Cons}^{\mathsf{rich}}$ and the static RAJA type of $v.\,\mathsf{next}.\,\mathsf{next}$ under the view poor is $\mathsf{Nil}^{\mathsf{poor}}$.

Definition 3.3.1 (Static RAJA Type) *For a given heap* σ*, a stack value* v*, a (possibly empty) access path* $\vec{p}$ *and a RAJA type* C^r *such that* $\sigma \vDash v : C$ *we recursively define* $\llbracket (v : r).\vec{p} \rrbracket^{\mathrm{stat}}_{\sigma}$ *by*

$$\llbracket (v : r).\vec{p} \rrbracket^{\mathrm{stat}}_{\sigma} = \begin{cases} r & \text{if } \vec{p} = \varepsilon \\ s & \text{if } \vec{p} = \vec{q}.a \text{ and } \llbracket (v : r).\vec{q} \rrbracket^{\mathrm{stat}}_{\sigma} = t \\ & \quad \text{and } \llbracket v.\vec{q} \rrbracket^{\mathrm{dyn}}_{\sigma} = E \text{ and } \mathsf{A}^{\mathrm{get}}\big(E^t_{;}, a\big) = s \end{cases}$$

If $C = \llbracket v.\vec{p} \rrbracket^{\mathrm{dyn}}_{\sigma}$ *and* $s = \llbracket (v : r).\vec{p} \rrbracket^{\mathrm{stat}}_{\sigma}$*, then* C^s *is the static RAJA type of* $v.\vec{p}$ *under the view* r.

Note that $\llbracket (v : r).\vec{q}.a \rrbracket^{\mathrm{stat}}_{\sigma}$ is only defined if $[\![v.\vec{q}]\!]_{\sigma} \in \sigma$.

Furthermore, assume that Γ is a RAJA context and σ is a sound heap with respect to its underlying FJEU context $|\Gamma|$ and an environment η, i.e. $\sigma \vDash \eta : |\Gamma|$. We then define the multiset of views associated with all aliases of a location $\ell \in \sigma$ by

$$\mathscr{V}_{\sigma,\eta,\Gamma}(\ell) = \{\!\{ (\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p} \}\!\}_\sigma \mid x \in \Gamma, [\![\eta_x.\vec{p}]\!]_\sigma = \ell\}$$

In the following we define when a heap is sound with respect to a RAJA context Γ and environment η. Clearly, the heap should be sound with respect to $|\Gamma|$ and η. Moreover, for each location $\ell \in \sigma$ must hold that there exists a view r, its *proto-view*, that can be split among the multiset of views associated with all aliases of l. Finally, if D is the dynamic type of ℓ, D^r should be a main RAJA type.

Definition 3.3.2 (Sound Heap) *We say that a memory configuration consisting of heap σ and stack η satisfies a RAJA context Γ, written $\sigma \vDash \eta : \Gamma$, if*

$$\sigma \vDash \eta : |\Gamma| \tag{3.3.1}$$

$$\forall \ell \in \sigma \,.\, \exists\, r \in \mathscr{V}^{\mathscr{C}} \,.\, \curlyvee(r \,|\mathscr{V}_{\sigma,\eta,\Gamma}(\ell)) \tag{3.3.2}$$

$$\text{for} \quad D = [\![\ell]\!]^{\mathrm{dyn}}_\sigma \,.\, D^r \ \mathsf{main} \tag{3.3.3}$$

We write $\sigma \vDash v : C^r$ for meaning $\sigma \vDash [x{\mapsto}v] : x{:}C^r$.

One may wonder how the mere existence of a "common" view for all the aliases of some location could be of any use; the answer lies in the following lemma:

Lemma 3.3.3 *Let $C \in \mathscr{C}$ and $r \in \mathscr{V}^{\mathscr{C}}$ and $\mathcal{D} \subseteq \mathscr{V}^{\mathscr{C}}$ and C^r main. If $\curlyvee(r \,|\mathcal{D})$ then for all $a \in \mathsf{A}(C)$ and $s \in \mathcal{D}$ holds $\curlyvee\big(\mathsf{A}^{\mathrm{set}}(C^s, a) \,\big|\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big)$.*

Proof. $\curlyvee(r \,|\mathcal{D})$ implies $\curlyvee\big(\mathsf{A}^{\mathrm{get}}(C^r, a) \,\big|\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big)$.
The goal follows by Lemma 3.2.24 with $\mathsf{A}^{\mathrm{set}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^r, a)$. □

The potential of an object in the heap is the sum over the potential of the static RAJA types corresponding to all access paths in scope that start in the object's location. The potential of a runtime configuration is the sum over the potential of all reachable objects.

Definition 3.3.4 (Potential) *If $\sigma \vDash v : C^s$ we define $\Phi_\sigma(v : r) \in \mathbb{D}$ by*

$$\Phi_\sigma(v : r) = \sum_{\vec{p}} \phi_\sigma\big((v{:}r).\vec{p}\big)$$

$$\text{where } \phi_\sigma\big((v{:}r).\vec{p}\big) = \begin{cases} \Diamond(D^s) & \text{if } [\![v.\vec{p}]\!]^{\mathrm{dyn}}_\sigma = D \ \text{ and } \ \{\!\{(v{:}r).\vec{p}\}\!\}^{\mathrm{stat}}_\sigma = s \\ 0 & \text{otherwise} \end{cases}$$

The potential is extended to environments and contexts as follows:

$$\Phi_\sigma(\eta : \Gamma) = \sum_{x \in \Gamma} \Phi_\sigma(\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle)$$

where we assume $\sigma \vDash \eta : |\Gamma|$.

This sum is possibly infinite, e.g. in the presence of circular data structures or an infinite heap. Furthermore each alias of a location makes a different contribution to the sum, whose value depends on the static RAJA type of that alias. Notice that $\phi_\sigma\big((v\!:\!r).\vec{p}\big) = 0$ if $[\![v.\vec{p}]\!]_\sigma = \mathsf{0}$.

The potential interacts with subtyping and sharing as might be expected:

Lemma 3.3.5 (Potential and subtyping) *Let* $C \in \mathscr{C}$ *and* $r, s \in \mathscr{V}^{\mathscr{C}}$ *and* σ *be a heap and* v *be a heap value and* $r \sqsubseteq s$ *and* $[\![v.\vec{p}]\!]_\sigma \in \sigma$. *Then:*

1. $\|(v\!:\!r).\vec{p}\|^{\mathrm{stat}}_\sigma \sqsubseteq \|(v\!:\!s).\vec{p}\|^{\mathrm{stat}}_\sigma$.
2. $\Phi_\sigma(v : r) \geq \Phi_\sigma(v : s)$.

Proof.

1. By induction on $\vec{p}$. If $\vec{p} = \epsilon$ the goal follows by assumption.

   Case $\vec{p} = \vec{q}.a$. Let $\|\!|v.\vec{q}|\!\|^{\mathrm{dyn}}_\sigma = E$ and $\|(v\!:\!r).\vec{q}\|^{\mathrm{stat}}_\sigma = t$ and $\|(v\!:\!s).\vec{q}\|^{\mathrm{stat}}_\sigma = t'$.

   $$\begin{array}{lcl} \|(v\!:\!r).q.\vec{a}\|^{\mathrm{stat}}_\sigma & \sqsubseteq & \|(v\!:\!s).q.\vec{a}\|^{\mathrm{stat}}_\sigma \iff \\ \mathsf{A}^{\mathrm{get}}\big(E, t, a\big) & \sqsubseteq & \mathsf{A}^{\mathrm{get}}\Big(E, t', a\Big) \end{array}$$

   which follows by $t \sqsubseteq t'$, which holds by induction hypothesis.

2. We show, for each $\vec{p}$, $\phi_\sigma\big((v : r).\vec{p}\big) \geq \phi_\sigma\big((v : s).\vec{p}\big)$. Let $\|\!|v.\vec{p}|\!\|^{\mathrm{dyn}}_\sigma = D$ and $\|(v\!:\!r).\vec{p}\|^{\mathrm{stat}}_\sigma = t$ and $\|(v\!:\!s).\vec{p}\|^{\mathrm{stat}}_\sigma = t'$. Then, the goal follows by $\Diamond\big(D^t\big) \geq \Diamond\Big(D^{t'}\Big)$, which follows by 1.

□

Lemma 3.3.6 (Potential and sharing) *Let* $C \in \mathscr{C}$ *and* σ *be a heap and* v *be a heap value and* $r, s_1, \ldots, s_n \in \mathscr{V}^{\mathscr{C}}$ *and* $[\![v.\vec{p}]\!]_\sigma \in \sigma$ *and* $\curlyvee(r\,|s_1, \ldots, s_n)$. *Then:*

1. $\curlyvee\big(\|(v\!:\!r).\vec{p}\|^{\mathrm{stat}}_\sigma \,\big|\, \|(v\!:\!s_1).\vec{p}\|^{\mathrm{stat}}_\sigma, \ldots, \|(v\!:\!s_n).\vec{p}\|^{\mathrm{stat}}_\sigma\big)$
2. $\Phi_\sigma(v : r) \geq \sum_i \Phi_\sigma(v : s_i)$.

Proof.

1. By induction on $\vec{p}$. If $\vec{p} = \epsilon$ the goal follows by assumption.

Case $\vec{p} = \vec{q}.a$. Let $\llbracket v.\vec{q} \rrbracket_\sigma^{\mathrm{dyn}} = E$ and $\lBag (v : s_i).\vec{q} \rBag_\sigma^{\mathrm{stat}} = t_i$.

$$\curlyvee\big(\lBag (v:r).\vec{q}.a \rBag_\sigma^{\mathrm{stat}} \,\big|\, \lBag (v:s_1).\vec{q}.a \rBag_\sigma^{\mathrm{stat}}, \ldots, \lBag (v:s_n).\vec{q}.a \rBag_\sigma^{\mathrm{stat}}\big) \iff \curlyvee\big(\mathsf{A}^{\mathrm{get}}\big(E^t, a\big) \,\big|\, \mathsf{A}^{\mathrm{get}}\big(E^{t_1}, a\big), \ldots, \mathsf{A}^{\mathrm{get}}\big(E^{t_n}, a\big)\big)$$

which follows by $\curlyvee(t \,|\, t_1, \ldots, t_n)$, which holds by induction hypothesis.

2. We show, for each $\vec{p}$, $\phi_\sigma\big((v:r).\vec{p}\big) \geq \sum_{s_i} \phi_\sigma\big((v:s_i).\vec{p}\big)$. Let $\llbracket v.\vec{p} \rrbracket_\sigma^{\mathrm{dyn}} = D$ and $\lBag (v:r).\vec{p} \rBag_\sigma^{\mathrm{stat}} = t$ and $\lBag (v:s_i).\vec{p} \rBag_\sigma^{\mathrm{stat}} = t_i$. Then, the goal follows by $\Diamond\big(D^t\big) \geq \sum_{t_i} \Diamond\big(D^{t_i}\big)$, which follows by 1.

□

The following definition will be useful for proving the soundness of field update. It allows us to describe formally the paths that are not affected by the update.

Definition 3.3.7 *Let σ be a heap and v be a heap value and $\vec{p}$ be an access path and l be a location. Then we say that* $v.\vec{p}$ passes through $\ell.a$ in σ, *written $[\sigma, v, \vec{p} \between \ell, a]$ iff there exists $\vec{q} \prec \vec{p}$ with $\llbracket v.\vec{q} \rrbracket_\sigma = \ell \wedge \vec{q}.a \preceq \vec{p}$.*

Lemma 3.3.8 *Let σ be a heap and v, w be heap values and $\vec{p}$ be an access path and l be a location and $\tau = \sigma[\ell.a \mapsto w]$. Then $\neg[\sigma, v, \vec{p} \between \ell, a]$ implies $\llbracket v.\vec{p} \rrbracket_\sigma = \llbracket v.\vec{p} \rrbracket_\tau$.*

Proof. By induction on $\vec{p}$.

Case $\vec{p} = \epsilon$. Then $\llbracket v.\vec{p} \rrbracket_\sigma = \llbracket v.\vec{p} \rrbracket_\tau = v$.

Case $\vec{p} = \vec{q}.b$. By assumption, for all $\vec{q} \prec \vec{p}$ holds that if $\llbracket v.\vec{q} \rrbracket_\sigma = \ell$ then $\vec{q}.a \not\preceq \vec{p}$. We have by induction hypothesis $\llbracket v.\vec{q} \rrbracket_\sigma = \llbracket v.\vec{q} \rrbracket_\tau$. Then, $\sigma_{\llbracket v.\vec{q} \rrbracket_\sigma} = \tau_{\llbracket v.\vec{q} \rrbracket_\tau} = (C, a_1 : v_1, \ldots, a_n : v_n)$. If $\llbracket v.\vec{q} \rrbracket_\sigma = \ell$ then $b \neq a$ by assumption. If $b = a_i$ then $\llbracket v.\vec{p} \rrbracket_\sigma = \llbracket v.\vec{p} \rrbracket_\tau = v_i$, otherwise $\llbracket v.\vec{p} \rrbracket_\sigma = \llbracket v.\vec{p} \rrbracket_\tau = \mathsf{0}$.

□

Lemma 3.3.9 (Heap soundness and subtyping) *Let σ be a heap, η an environment and Γ, Θ be RAJA contexts. If $\sigma \vDash \eta : \Gamma$ and $\Gamma <: \Theta$ then also $\sigma \vDash \eta : \Theta$.*

Proof. $\sigma \vDash \eta : |\Theta|$ follows by Lemma 2.3.7. Next we show (3.3.2). Let $\ell \in \sigma$, by assumption we have a proto-view q with:

$$\curlyvee\big(q \,\big|\, \{\lBag (\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p} \rBag_\sigma^{\mathrm{stat}} \mid x \in \Gamma,\ \llbracket \eta_x.\vec{p} \rrbracket_\sigma = \ell\}\big)$$

Moreover, for $x \in \Gamma$ holds: $\lBag (\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p} \rBag_\sigma^{\mathrm{stat}} \sqsubseteq \lBag (\eta_x : \langle\!\langle \Theta_x \rangle\!\rangle).\vec{p} \rBag_\sigma^{\mathrm{stat}}$ by Lemma 3.3.5 and assumption, thus, by Lemma 3.2.24, we obtain the desired

$$\curlyvee\big(q \,\big|\, \{\lBag (\eta_x : \langle\!\langle \Theta_x \rangle\!\rangle).\vec{p} \rBag_\sigma^{\mathrm{stat}} \mid x \in \Theta,\ \llbracket \eta_x.\vec{p} \rrbracket_\sigma = \ell\}\big)$$

Finally, (3.3.3) follows by assumption. □

Lemma 3.3.10 (Heap soundness and sharing) *Let σ be a heap, η an environment and let $\Gamma = x:C^r, \Delta$ and $\eta' = \eta[y_1 \mapsto \eta_x, \ldots, y_n \mapsto \eta_x]$ and $\Gamma' = y_1 : C^{s_1}, \ldots, y_n : C^{s_n}, \Delta$ and $\curlyvee(r\,|s_1, \ldots, s_n)$. Then if $\sigma \vDash \eta : \Gamma$ then also $\sigma \vDash \eta' : \Gamma'$.*

Proof. $\sigma \vDash \eta' : |\Gamma'|$ follows by Lemma 2.3.8. Next we show (3.3.2). Let $\ell \in \sigma$, by assumption we have a proto-view q with:

$$\curlyvee\big(q\,\big|\mathscr{V}_{\sigma,\eta,\Delta}(\ell) \cup \{\, \text{⦇}(\eta_x : r).\vec{p}\text{⦈}^{\text{stat}}_{\sigma} \mid \llbracket \eta_x.\vec{p} \rrbracket_\sigma = \ell \}\big)$$

By Lemma 3.3.6 and assumption we obtain:

$$\curlyvee(\text{⦇}(\eta_x : r).\vec{p}\text{⦈}_\sigma \,|\, \text{⦇}(\eta_x : s_1).\vec{p}\text{⦈}_\sigma, \ldots, \text{⦇}(\eta_x : s_n).\vec{p}\text{⦈}_\sigma)$$

Then, applying Lemma 3.2.23 for each $\eta_x.\vec{p}$ with $\llbracket \eta_x.\vec{p} \rrbracket_\sigma = \ell$ gives us the desired

$$\curlyvee(q\,|\mathscr{V}_{\sigma,\eta,\Delta}(\ell) \bigcup_i \mathscr{V}_{\sigma,\eta,x:C^{s_i}}(\ell))$$

Finally, (3.3.3) follows by assumption. □

Theorem 3.3.11 (Soundness of RAJA typing)
Let $\mathscr{R}$ be a well-typed RAJA program, e be an expression, C^r be a refined type and Γ, Δ be RAJA contexts. Let moreover σ, τ be heaps and η be an environment.

$$\mathcal{D} :: \Gamma \vdash^{n}_{n'} e : C^r \tag{3.3.4}$$

$$\mathcal{E} :: \eta, \sigma \vdash e \rightsquigarrow v, \tau \tag{3.3.5}$$

$$\sigma \vDash \eta : (\Gamma, \Delta) \tag{3.3.6}$$

Then

$$\eta, \sigma \vdash^{n + \Phi_\sigma(\eta:\Gamma) + \Phi_\sigma(\eta:\Delta)}_{n' + \Phi_\tau(v:r) + \Phi_\tau(\eta:\Delta)} e \rightsquigarrow v, \tau \tag{3.3.7}$$

$$\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r) \tag{3.3.8}$$

where x_{res} is assumed to be an unused auxiliary variable, i.e. $x_{res} \notin \Gamma, \Delta$. Note that (3.3.6) implies $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ by definition of notation.

Proof. The proof is by induction on $\mathcal{E}$ and a subordinate induction on $\mathcal{D}$. For establishing (3.3.8) we will address only parts (3.3.2) and (3.3.3) of Def. 3.3.2 since (3.3.1), soundness of FJEU typing, follows by Theorem 2.3.10, since, by Lemma 3.2.27, $|\Gamma| \vdash e : C$.

Case ($\Diamond Share$) Without loss of generality, we assume $x, y_1, \ldots, y_n \notin \Delta$. Let $\Gamma = \Theta, x : D^s$, $e = e_0[x/y_1, \ldots, x/y_n]$ for some e_0 and $\Gamma' = \Theta, y_1 : D^{q_1}, \ldots, y_n : D^{q_n}$. From (3.3.4) and rule ($\Diamond Share$) we obtain $\Gamma' \vdash^{n}_{n'} e_0 : C^r$ and $\curlyvee(s\,|q_1, \ldots, q_n)$. Furthermore let $\eta' = \eta[y_1 \mapsto \eta_x, \ldots, y_n \mapsto \eta_x]$.

From (3.3.5) we obtain $\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau$ by a derivation of the same length. $\sigma \vDash \eta' : (\Gamma', \Delta)$ holds by Lemma 3.3.10. Applying the induction hypothesis now yields

$$\eta', \sigma \mid^{\frac{n + \Phi_\sigma(\eta' : \Gamma') + \Phi_\sigma(\eta' : \Delta)}{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta' : \Delta)}} e_0 \rightsquigarrow v, \tau$$

By Lemma 3.3.6 holds $\Phi_\sigma(\eta_x : s) \geq \sum_i \Phi_\sigma(\eta_x : q_i)$ and hence $\Phi_\sigma(\eta : \Gamma) \geq \Phi_\sigma(\eta' : \Gamma')$. Since η and η' coincide on Δ, we also have $\Phi_\sigma(\eta : \Delta) = \Phi_\sigma(\eta' : \Delta)$ and $\Phi_\tau(\eta : \Delta) = \Phi_\tau(\eta' : \Delta)$. Hence we obtain

$$\eta, \sigma \mid^{\frac{n + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)}{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta : \Delta)}} e \rightsquigarrow v, \tau$$

by Lemma 2.3.12, item 2, as required.

Case ($\Diamond Waste$) Assume that (3.3.4) was established in the last step by application of rule ($\Diamond Waste$), so $\Theta \vdash^{u}_{u'} e : D^s$ and $\Gamma <: \Theta$ and $D^s <: C^r$. Furthermore $n \geq u$ and $n' + u \leq u' + n$. Since $\Gamma <: \Theta$ we have by Lemma 3.3.9, $\sigma \vDash \eta : (\Theta, \Delta)$. The induction hypothesis gives $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : D^s)$ from which we get $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r)$ again by Lemma 3.3.9. We also obtain

$$\eta, \sigma \mid^{\frac{u + \Phi_\sigma(\eta : \Theta) + \Phi_\sigma(\eta : \Delta)}{u' + \Phi_\tau(v : s) + \Phi_\tau(\eta : \Delta)}} e \rightsquigarrow v, \tau$$

We have by Lemma 3.3.5 $\Phi_\sigma(\eta : \Gamma) \geq \Phi_\sigma(\eta : \Theta)$, so we get by Lemma 2.3.12, item 1

$$\eta, \sigma \mid^{\frac{u + (n-u) + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)}{u' + (n-u) + \Phi_\tau(v : s) + \Phi_\tau(\eta : \Delta)}} e \rightsquigarrow v, \tau$$

and finish with rule ($\vdash_{\mathsf{Sh}} Waste$), since $n' \leq u' + n - u$.

Case ($\vdash_{\mathsf{S}} New$) By (3.3.4) we have $e = \texttt{new}\ C$, $n = \Diamond(C^r) + 1$, $n' = 0$, $\Gamma = \varnothing$ and $v = \ell$. We have $\Phi_\tau(v : r) = \Diamond(C^r)$ since $[\![v.a_i]\!]_\tau = \mathsf{0}$ by rule ($\vdash_{\mathsf{S}} New$). Furthermore $\Phi_\sigma(\eta : \Delta) = \Phi_\tau(\eta : \Delta)$ since existent locations are not altered and even more importantly $\ell \notin \mathrm{dom}(\sigma)$ and (3.3.6) guarantees by (2.3.2) and (2.3.1) that ℓ does not occur anywhere in η and σ. Therefore (3.3.7) follows by Lemma 2.3.12, item 2. For (3.3.8) we note that $\mathscr{V}_{\tau, \eta[x_{res} \mapsto \ell], \Gamma}(\ell) = \{r\}$, but $\curlyvee(r \,|\, r)$ follows by Lemma 3.2.21 and C^r main follows by assumption.

Case ($\vdash_{\mathsf{S}} Inv$) In this case

$$e = x.m(\vec{y}) \tag{3.3.9}$$

$$\Gamma = x : C^{s_0}, y_1 : E_1^{s_1}, \ldots y_j : E_n^{s_n} \tag{3.3.10}$$

$$\mathsf{M}(C, m) = \forall \vec{v}, \vec{q} \exists \vec{w}, \vec{t}\,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{q_1/q_2} H^{v_{n+1}} \ \&\ \mathcal{C}(\vec{v}, \vec{q}, \vec{w}, \vec{t}) \tag{3.3.11}$$

$$\pi = \pi'(\{v_1 \mapsto s_1, \ldots, v_{n+1} \mapsto s_{n+1}\}, \{q_1 \mapsto n_1, q_2 \mapsto n_2\}) \models \mathcal{C} \tag{3.3.12}$$

The premises of (3.3.5) yield:

$$\eta_x \in \sigma \tag{3.3.13}$$

$$\sigma(\eta_x) = (D, a_1 : v_1, \ldots, a_k : v_k) \tag{3.3.14}$$

$$\mathsf{M}_{\mathsf{body}}(D, m) = e_0 \tag{3.3.15}$$

$$\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau \tag{3.3.16}$$

where $\eta' = [\mathtt{this} \mapsto \eta_x, x_1 \mapsto \eta_{y_1}, \ldots, x_n \mapsto \eta_{y_n}]$. Without loss of generality $\mathrm{dom}(\eta) \cap \mathrm{dom}(\eta') = \varnothing$. Moreover we have:

$$\mathsf{M}(D, m) = \forall \vec{u}, \vec{p} \exists \vec{w'}, \vec{t'} \,.\, D^{u_0}; \vec{E}^{\vec{u}} \xrightarrow{p_1/p_2} H^{u_{n+1}} \;\&\; \mathcal{D}(\vec{u}, \vec{p}, \vec{w'}, \vec{t'}) \tag{3.3.17}$$

We have $D <: C$ by (3.3.6) which implies $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$ because $\mathscr{R}$ is a well-typed RAJA program, i.e. by Def. 3.2.26. Then we obtain, by the definition of subtyping, a valuation $\bar{\pi}$ with $\bar{\pi}|_{\mathrm{dom}(\pi)} = \pi$ such that $\bar{\pi} \models \mathcal{D}$ and $s_0 = \bar{\pi}(u_0)$ and $s_i \sqsubseteq \bar{\pi}(u_i)$ and $\bar{\pi}(u_{n+1}) \sqsubseteq s_{n+1}$ and $\bar{\pi}(p_1) \leq n_1$ and $\bar{\pi}(p_2) \geq n_2$. Notice that $n_2 \leq \bar{\pi}(p_2) + \underbrace{(n_1 - \bar{\pi}(p_1))}_{\geq 0}$.

Since $\bar{\pi} \models \mathcal{D}$, by $\vdash m\colon \mathsf{M}(D, m)\ \mathsf{ok}$ and $(\Diamond Waste)$ we get

$$\frac{\mathtt{this} : D^{r_1}, x_1 : E_1^{\bar{\pi}(u_1)}, \ldots, x_n : E_n^{\bar{\pi}(u_n)} \;\vdash^{\bar{\pi}(p_1) + \Diamond(D^{r_2})}_{\bar{\pi}(p_2)}\; e_0 : H^{\bar{\pi}(u_{n+1})}}{\mathtt{this} : D^{r_1}, x_1 : E_1^{s_1}, \ldots, x_n : E_n^{s_n} \;\vdash^{n_1 + \Diamond(D^{r_2})}_{n_2}\; e_0 : H^{s_{n+1}}}$$

and $\curlyvee(s_0 \,|r_1, r_2\,)$. By (3.3.16) and Lem.2.3.9 we have $\eta\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau$. Application of the induction hypothesis then yields

$$\eta\eta', \sigma \;\vdash^{n_0}_{n_2 + \Phi_\tau(\eta[x_{res} \mapsto v] : \Delta, x_{res} : H^{s_{n+1}})}\; e_0 \rightsquigarrow v, \tau \tag{3.3.18}$$

where $n_0 = n_1 + \Diamond(D^{r_2}) + \Phi_\sigma(\eta\eta' : \Delta, \mathtt{this} : D^{r_1}, x_1 : E_1^{s_1}, \ldots, x_n : E_n^{s_n})$. We have:

$$\begin{aligned}
&\Phi_\sigma(\eta : \Delta, x : C^{s_0}, y_1 : E_1^{s_1}, \ldots, y_n : E_n^{s_n}) \\
&= \Phi_\sigma\big(\eta\eta' : \Delta, x_1 : E_1^{s_1}, \ldots, x_n : E_n^{s_n}\big) + \Phi_\sigma(\eta_x : s_0) \\
&\underbrace{\geq}_{\text{Lem. 3.3.6}} \Phi_\sigma\big(\eta\eta' : \Delta, x_1 : E_1^{s_1}, \ldots, x_n : E_n^{s_n}\big) + \Phi_\sigma(\eta_x : r_1) + \Phi_\sigma(\eta_x : r_2) \\
&\geq \Phi_\sigma\big(\eta\eta' : \Delta, \mathtt{this} : D^{r_1}, x_1 : E_1^{s_1}, \ldots x_n : E_n^{s_n}\big) + \Diamond(D^{r_2})
\end{aligned} \tag{3.3.19}$$

where the crucial part of the last step follows by $[\![\eta_x.\varepsilon]\!]^{\mathrm{dyn}}_\sigma = D$ and therefore $\Diamond(D^{r_2})$ is a part of the sum $\Phi_\sigma(\eta_x : r_2)$. The conclusion then follows by Lem. 2.3.12, item 2.

Case ($\vdash_{\mathsf{S}}$ *Access*) Suppose that (3.3.4) and (3.3.5) have been derived by rules ($\Diamond$*Access*) and ($\vdash_{\mathsf{S}}$ *Access*). So we have $\eta_x = \ell$, $\tau = \sigma$, $C_0 =$

$\llbracket \eta_x \rrbracket^{\mathrm{dyn}}_\sigma$, $\mathsf{A}^{\mathrm{get}}(C^r_0, a) = s_0$, $s_0 \sqsubseteq s$, $C.a = D$ and $\Gamma = x : C^r$ and $v = [\![\eta_x.a]\!]_\sigma$. We show

$$\eta, \sigma \vdash^{\frac{\Phi_\sigma(\eta_x : r) + \Phi_\sigma(\eta : \Delta)}{\Phi_\sigma(v : s) + \Phi_\sigma(\eta : \Delta)}} x.a \rightsquigarrow v, \tau$$

which specialises to $\Phi_\sigma(\eta_x : r) \geq \Phi_\sigma(v : s)$. By definition of potential we know $\Phi_\sigma(\eta_x : r) \geq \Phi_\sigma(v : s_0)$. Moreover $s_0 \sqsubseteq s$ implies by Lemma 3.3.5, $\Phi_\sigma(v : s_0) \geq \Phi_\sigma(v : s)$, thus, we finish by transitivity.

Case $(\vdash_{\mathsf{S}} Update)$

Suppose that (3.3.4) and (3.3.5) have been derived by rules $(\Diamond Update)$ and $(\vdash_{\mathsf{S}} Update)$.

So we have $\eta_x = \ell$, $\tau = \sigma[\ell.a \mapsto \eta_y]$, $C_0 = \llbracket \eta_x \rrbracket^{\mathrm{dyn}}_\sigma$, $s_0 = \mathsf{A}^{\mathrm{get}}(C^r_0, a)$, $s = \mathsf{A}^{\mathrm{get}}(C^r, a)$ and $\Gamma = x : C^r, y : D^s$. Let furthermore $\Theta := \Delta, x : C^r$ and $\Lambda = \Gamma, \Delta$.

We assume $\eta_y \in \sigma$ for otherwise $\forall z \in \Theta, \vec{p}\,.\ [\![\eta_z.\vec{p}]\!]_\tau \in \tau \implies [\![\eta_z.\vec{p}]\!]_\tau = [\![\eta_z.\vec{p}]\!]_\sigma$ hence the claim is trivially true. The case $\eta_y \in \sigma$ is more interesting for then τ may contain new valid path, possibly circular ones, whose effects on the potential and heap soundness we will now study.

For any locations ℓ_1, ℓ_2 let

$$\mathcal{P}(\ell_1, \ell_2) = \left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\sigma = \ell_2 \wedge \neg[\sigma, \ell_1, \vec{p} \lozenge \ell, a] \right\} \tag{3.3.20}$$

denote the set of access paths from ℓ_1 to ℓ_2, which do not pass through $\ell.a$. Notice that by Lem. 3.3.8 we have

$$\left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\sigma = \ell_2 \wedge \neg[\sigma, \ell_1, \vec{p} \lozenge \ell, a] \right\} = \left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\tau = \ell_2 \wedge \neg[\tau, \ell_1, \vec{p} \lozenge \ell, a] \right\}$$

or in other words, the set $\mathcal{P}(\ell_1, \ell_2)$ remains unchanged if we replace σ by τ within the defining equation (3.3.20) for $\mathcal{P}(\cdot\,, \cdot)$.

We now define for $z \in \Gamma, \Delta$

$$\mathcal{D}_z = \mathcal{P}(\eta_z, \ell) \qquad\qquad \mathcal{N}_z = \bigcup_{\ell_2 \in \sigma} \mathcal{P}(\eta_z, \ell_2)$$

Notice $\varepsilon \in \mathcal{D}_x$ and $\forall z\,.\,\mathcal{D}_z \subseteq \mathcal{N}_z$.

Observe for $z \in \Theta$ that

$$\left\{ \vec{p} \;\middle|\; [\![\eta_z.\vec{p}]\!]_\tau \in \tau \right\} = \mathcal{N}_z \uplus (\mathcal{D}_z(.a.\mathcal{D}_y)^*.a.\mathcal{N}_y)$$

For the "$\subseteq$"-direction, use the fact that a path $\vec{p}$ such that $[\![\eta_z.\vec{p}]\!]_\tau \in \tau$ either does not pass through $\ell.a$ in which case $\vec{p} \in \mathcal{N}_z$, or else it passes

through $\ell.a$ a finite number of times and can then be decomposed as $\vec{p} = \vec{d}.a.\vec{e_1}\ldots a.\vec{e_k}.\vec{r}$ where $\vec{d} \in \mathcal{D}_z$, $\vec{e_i} \in \mathcal{D}_y$ and $\vec{r} \in \mathcal{N}_y$.

By assumption (3.3.6) and Def. 3.3.2 there exists r' with $\curlyvee(r' \,|\mathscr{V}_{\sigma,\eta,\Lambda}(\ell)\,)$ where

$$\begin{aligned}\mathscr{V}_{\sigma,\eta,\Lambda}(\ell) &= \left\{ \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\sigma} \mid z \in \Lambda, [\![\eta_z.\vec{p}]\!]_{\sigma} = \ell \right\} \\ &\supseteq \left\{ \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\sigma} \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\} \\ &= \left\{ \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\}\end{aligned}$$

where the last step follows since $\mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\sigma} = \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau}$ for $\vec{p} \in \mathcal{D}_z$. Therefore by Lemma 3.2.22 we obtain

$$\curlyvee\big(r' \,\big|\, \left\{ \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\}\big)$$

Moreover, $r \in \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau}$ since $x \in \Lambda$ and $\varepsilon \in \mathcal{D}_x$ and $C_0^{r'}\ \mathsf{main}$ holds by the definition of a sound heap, so that we can apply Lemma 3.3.3 and obtain:

$$\curlyvee\big(\mathsf{A}^{\mathrm{set}}(C_0^r, a) \,\big|\, \left\{ \mathbb{\llbracket}(\eta_z : \Lambda_z).\vec{p}.a\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\}\big)$$

By (3.3.6) we have $C_0 <: C$, thus, we have by assumption $s \sqsubseteq \mathsf{A}^{\mathrm{set}}(C_0^r, a)$. Therefore by Lemma 3.2.24

$$\curlyvee\left(s \;\middle|\; \begin{array}{c} \left\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid z \in \Theta, \vec{q} \in \mathcal{D}_z.a \right\} \\ \uplus \quad \left\{ \mathbb{\llbracket}(\eta_y : s).\vec{q}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid \vec{q} \in \mathcal{D}_y.\boldsymbol{a} \right\} \end{array} \right) \tag{3.3.21}$$

We will now prove

$$\curlyvee\big(s \,\big|\, \left\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^* a \right\}\big) \tag{3.3.22}$$

by proving $\forall n\,.\,\mathrm{IH}(n)$ where we have

$$\mathrm{IH}(n) \equiv \curlyvee\big(s \,\big|\, \left\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \right\}\big)$$

as our induction hypothesis. IH(0) follows directly from (3.3.21) and Lemma 3.2.22.

Now suppose IH(n) holds for some n. For any $\vec{q_2} \in \mathcal{D}_y.a$ we have $[\![\eta_y.\vec{q_2}]\!]_{\tau} = \eta_y \in \tau$. By Lemma 3.3.6 we have

$$\curlyvee\Big(\mathbb{\llbracket}(\eta_y : s).\vec{q_2}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\Big|\; \left\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}.\vec{q_2}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \right\}\Big)$$

Hence we may replace the multiset $\left\{ \mathbb{\llbracket}(\eta_y : s).\vec{q}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \mid \vec{q} \in \mathcal{D}_y.\boldsymbol{a} \right\}$ by $\left\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}.\vec{q_2}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \right\}$ in (3.3.21) thanks to Lemma 3.2.23 yielding IH($n+1$) as required.

We will now prove (3.3.7) which specialises to

$$\Phi_\sigma(\eta_y : s) + \Phi_\sigma(\eta : \Theta) \geq \Phi_\tau(\eta : \Theta)$$

In order to prove this equation, we expand

$$\Phi_\tau(\eta : \Theta) = \sum_{z\in\Theta} \sum_{\vec{p}} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big)$$

which can be split by our earlier observations into

$$= \sum_{z\in\Theta} \sum_{\vec{p}\in\mathcal{N}_z} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big) + \sum_{z\in\Theta} \sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^* a} \sum_{\vec{r}\in\mathcal{N}_y} \phi_\tau\big((\eta_z : \Theta_z).\vec{q}.\vec{r}\big)$$

For the first addend we calculate

$$\begin{aligned}
&\sum_{z\in\Theta} \sum_{\vec{p}\in\mathcal{N}_z} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big)\\
=&\sum_{z\in\Theta} \sum_{\vec{p}\in\mathcal{N}_z} \phi_\sigma\big((\eta_z : \Theta_z).\vec{p}\big)\\
\leq&\, \Phi_\sigma(\eta : \Theta)
\end{aligned}$$

and for the second addend we calculate

$$\begin{aligned}
&\sum_{z\in\Theta} \sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^* a} \sum_{\vec{r}\in\mathcal{N}_y} \phi_\tau\big((\eta_z : \Theta_z).\vec{q}.\vec{r}\big)\\
=&\sum_{z\in\Theta} \sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^* a} \sum_{\vec{r}\in\mathcal{N}_y} \phi_\tau\big((\eta_y : [\![(\eta_z : \Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau).\vec{r}\big)\\
=&\sum_{z\in\Theta} \sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^* a} \sum_{\vec{r}\in\mathcal{N}_y} \phi_\sigma\big((\eta_y : [\![(\eta_z : \Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau).\vec{r}\big)\\
\leq&\sum_{z\in\Theta} \sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^* a} \Phi_\sigma\big(\eta_y : [\![(\eta_z : \Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau\big)\\
\leq&\, \Phi_\sigma(\eta_y : s)
\end{aligned}$$

where the last inequality follows by (3.3.22) and Lemmma 3.3.6.

For proving (3.3.8) let ℓ_2 be an arbitrary location in $\mathrm{dom}(\tau) = \mathrm{dom}(\sigma)$. From (3.3.6) we have the existence of a q such that $\curlyvee(q \,|\mathscr{V}_{\sigma,\eta,\Lambda}(\ell_2))$. Therefore by Lem. 3.2.22

$$\curlyvee\left(q \,\middle|\, \begin{array}{c} \big\{ [\![(\eta_z : \Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau \;\big|\; z \in \Theta, \vec{p} \in \mathcal{P}(\eta_z, \ell_2)\big\} \\ \uplus\ \big\{ [\![(\eta_y : s).\vec{p}]\!]^{\mathrm{stat}}_\tau \;\big|\; \vec{p} \in \mathcal{P}(\eta_y, \ell_2)\big\} \end{array}\right) \tag{3.3.23}$$

Note the change from σ to τ which is legitimate since only access paths in $\mathcal{P}(\cdot,\cdot)$ are considered. From (3.3.22) and Lemma 3.3.6 we obtain $\forall \vec{p} \in \mathcal{P}(\eta_y, \ell_2)$

$$\curlyvee\big([\![(\eta_y : s).\vec{p}]\!]^{\mathrm{stat}}_\tau \,\big|\big\{ [\![(\eta_z : \Theta_z).\vec{q}.\vec{p}]\!]^{\mathrm{stat}}_\tau \;\big|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^* a\big\}\big)$$

Hence from (3.3.23) and Lemma 3.2.23 we obtain

$$\curlyvee\left(q \;\middle|\; \begin{array}{c} \left\{ \mathbb{\llbracket}(\eta_z{:}\Theta_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{p} \in \mathcal{P}(\eta_z, \ell_2) \right\} \\ \uplus\ \left\{ \mathbb{\llbracket}(\eta_z{:}\Theta_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{p} \in \mathcal{D}_z(a\mathcal{D}_y)^* a\mathcal{P}(\eta_y, \ell_2) \right\} \end{array} \right)$$

and therefore $\curlyvee(q \,|\mathscr{V}_{\tau,\eta,\Theta}(\ell_2))$ as required, since

$$\begin{aligned} \mathscr{V}_{\tau,\eta,\Theta}(\ell_2) &= \left\{ \mathbb{\llbracket}(\eta_z{:}\Theta_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, [\![\eta_z.\vec{p}]\!]_{\tau} = \ell_2 \right\} \\ &= \left\{ \mathbb{\llbracket}(\eta_z{:}\Theta_z).\vec{p}\mathbb{\rrbracket}^{\mathrm{stat}}_{\tau} \;\middle|\; z \in \Theta, \vec{p} \in \mathcal{P}(\eta_z, \ell_2) \cup \right. \\ &\qquad \left. \mathcal{D}_z(a\mathcal{D}_y)^* a\mathcal{P}(\eta_y, \ell_2) \right\} \end{aligned}$$

Finally, (3.3.3) follows by (3.3.6).

□

This corollary is a direct consequence of the main result and it is in this form that we intend to use it.

Corollary 3.3.12 *Suppose that* $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ *is an FJEU program containing the classes* List, Cons *and* Nil *for building singly-linked lists, a class* C *containing a method* `D main(List args)`*, and arbitrary other classes and methods.*

Suppose furthermore, that there exists a RAJA annotation of this program containing the following type for main*:* $C^{r'}; \mathsf{List}^{\mathsf{r}} \xrightarrow{n/n'} D^{s}$*, where* $\Diamond(\mathsf{Cons}^{\mathsf{r}}) = m$ *and* $\Diamond(\mathsf{Nil}^{\mathsf{r}}) = k$ *and* $\mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{r}}_{\mathsf{i}}, \mathsf{next}) = r$*. Then, evaluating* main *in a heap where* args *points to a linked list of length* l *requires at most* $k + n + ml$ *memory cells.*

3.4 Algorithmic problems

We have presented a type system for analysing the heap-space consumption of FJEU programs and we have proved its soundness. However, the declarative system presented here cannot be implemented for the following reasons: first, the typing rules are not syntax directed; second, for checking the polymorphic types of methods we need to check all their (possibly infinite) instances, which is infeasible. Hence we need an algorithmic version of the system that is more suitable for a practical implementation. Moreover, the analysis would not be practical if the programmers would need to provide the views and potential annotations by hand. Therefore a type inference algorithm is vital.

Algorithmic typechecking
Given: A RAJA program $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$.
Wanted: Is $\mathscr{R}$ a well-typed RAJA program?

Type inference
Given: An FJEU program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$.
Wanted: A well-typed RAJA program $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$.

Heap-space analysis
Given: An FJEU program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$.
Wanted: Number of heap cells required for executing main as a function of main's arguments.

Figure 3.5: Algorithmic problems regarding the type system RAJA.

In the remaining of this thesis we attempt to solve those two problems: algorithmic type checking and type inference, which we define formally in Fig. 3.5. Notice that, by Corollary 3.3.12, solving the type inference problem for a particular program $\mathcal{P}$ implies solving the heap-space analysis problem for $\mathcal{P}$.

Chapter 4

Type Checking RAJA Programs

4.1 Overview

The main goal of this chapter is to define a typing system that is equivalent to the system described in the previous chapter, but that is more suitable for algorithmic tasks like type checking and type inference than the declarative system.

Some of the typing rules from the previous chapter are not syntax-directed. One of them is the rule ($\Diamond Waste$).

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Gamma <: \Theta \quad D^s <: C^r \quad \Theta \vdash^{u}_{u'} e : D^s}{\Gamma \vdash^{n}_{n'} e : C^r} \ (\Diamond Waste)$$

To create syntax-directed typing rules, subtyping and weakening of potential annotations can be integrated in the other rules and so ($\Diamond Waste$) can be removed from the system. Another non-syntax-directed rule is ($\Diamond Share$).

$$\frac{\curlyvee(s\,|s_1, \ldots, s_j) \qquad \Gamma, y_1 : D^{s_1}, \ldots, y_j : D^{s_j} \vdash^{n}_{n'} e : C^r}{\Gamma, x : D^s \vdash^{n}_{n'} e[x/y_1, \ldots, x/y_j] : C^r} \ (\Diamond Share)$$

Unlike with ($\Diamond Waste$), integrating ($\Diamond Share$) in the remaining rules is not so simple: the system gives no information about how to find the views $s_1, \ldots, s_j$.

Another rule that is difficult to implement, although it is syntax-directed, is the rule ($\Diamond Let$).

$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x : D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ D\,x = e_1\ \mathsf{in}\ e_2 : C^r} \ (\Diamond Let)$$

When type checking a let expression, even if we know the potential annotations n, n'', there is no way of knowing the annotation n'.

Finally, the rule for method typing ($\Diamond MBody$) is also problematic: knowing the polymorphic type here is not enough for type checking the method. Similarly to the rule ($\Diamond Share$), this rule gives no information about how to find the views r_1, r_2.

$$\frac{\begin{array}{c} m \in \mathsf{Meth}(C) \qquad \phi = \mathsf{M}(C,m) \\ \forall\, T = (C^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{n/n'} H^{s_{n+1}})\ \mathsf{instanceof}\ \phi \qquad \curlyvee(s_0\,|r_1, r_2\,) \\ \mathtt{this}{:}C^{r_1}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \ \vdash^{\frac{n\,+\,\Diamond(C^{r_2})}{n'}}\ \mathsf{M}_{\mathsf{body}}(C,m) : H^{s_{n+1}} \end{array}}{\vdash m : \phi\ \mathsf{ok}}\ (\Diamond MBody)$$

For solving all those difficulties when type checking RAJA programs, we need to annotate the programs with more information than only the polymorphic RAJA types. We need view and potential annotations inside the body of methods as well. In particular, we need to annotate all the variable occurrences with views, and the let expressions with potentials. Moreover, the annotations depend on the particular instance of the polymorphic type. Thus, we would need a map from monomorphic types to annotated expressions. You could think that these are just too many annotations, and that it would be definitely infeasible to write them all by hand. That is true, but we aim at describing type inference for the system. You can regard the system described in this chapter as an intermediate step towards type inference: only when we know how much information we actually need, we can infer it.

This chapter is organised as follows. In Section 4.2 we describe RAJA programs with explicit types, which are RAJA programs with more annotations and that contain sets of instances of the polymorphic RAJA types. In Section 4.3 we describe algorithmic views that will be useful for describing an algorithmic rule for typing method bodies. Then, we describe the algorithmic typing rules in Section 4.4 and prove them sound and complete with respect to the declarative rules from Chapter 3. We conclude with a discussion about efficiency of algorithmic typing of RAJA programs, given that they are finite, in Section 4.5.

Notation. In this chapter we use the letters r, s, p, q for referring to views and the letters n, m, u, w when referring to elements of $\mathbb{D}$. Notice than in other chapters we use p, q for referring to arithmetic variables and u, w for referring to view variables. Since in this chapter we do not speak about variables or constraints, there should be no room for confusion.

4.2 RAJA program with explicit types

We consider here RAJA programs extended with annotations in the types and bodies of methods in a way that we can type check them deterministically. First, we require each method to be annotated with a set of mono-

$$
\begin{array}{rll}
e^\circ ::= & x^s & \text{(Variable)} \\
| & \mathsf{null} & \text{(Constant)} \\
| & \texttt{new}\ C & \text{(Construction)} \\
| & \texttt{free}(x^s) & \text{(Destruction)} \\
| & (C)x^s & \text{(Cast)} \\
| & x^s.a_i & \text{(Access)} \\
| & x^s.a_i\texttt{<-}x^s & \text{(Update)} \\
| & x^s.m(x_1^{s_1}, \ldots, x_n^{s_n}) & \text{(Invocation)} \\
| & \texttt{if}\ x^s\ \texttt{instanceof}\ C\ \texttt{then}\ e_1^\circ\ \texttt{else}\ e_2^\circ & \text{(Conditional)} \\
| & \mathsf{let}\ [D]\, x^n = e_1^\circ\ \mathsf{in}\ e_2^\circ & \text{(Let)}
\end{array}
$$

Figure 4.1: Syntax of annotated FJEU expressions.

morphic RAJA types that are instances of the method's polymorphic type. These types can be built by solving the resource constraints of the method.

Moreover, we require variables to be annotated with a view, except for when the variable is declared. This is necessary due to the non-deterministic nature of the rule ($\Diamond Share$) that provides no information about how the refined types for the different occurrences of a variable can be found. With the variable occurrences annotated with views, the task of the type checker is to check the correctness of a given sharing. We also require the let expressions to be annotated with a nonnegative number $n \in \mathbb{D}$, which is necessary for type checking the expression. In summary, we say that an FJEU expression e is *annotated*, written e°, when it is defined by the grammar in Fig. 4.1.

Definition 4.2.1 (RAJA program with explicit types)
A RAJA program with explicit types *is an annotation of an FJEU program* $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ *in the form of a tuple* $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *where*

1. $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *is a RAJA program.*
2. $\mathscr{V} \subseteq \mathscr{V}^{\mathscr{C}}$ *is a set of views.*
3. Inst *assigns to each class* C *and method* $m \in \mathsf{Meth}(C)$ *with* n *arguments a nonempty set of* n*-ary monomorphic RAJA method types* $\mathsf{Inst}(C, m)$ *such that for all* $T \in \mathsf{Inst}(C, m)$ *holds* $T\ \mathsf{instanceof}\ \mathsf{M}(C, m)$ *and if* $T = s_0; \vec{\mathfrak{s}} \xrightarrow{n/n'} s_{n+1}$ *then for all* i *holds* $s_i \in \mathscr{V}$.
4. *For each class* $C \in \mathscr{C}$ *and method* $m \in \mathsf{Meth}(C)$ *and* $T \in \mathsf{Inst}(C, m)$*, if* $\mathsf{M}_{\mathsf{body}}(C, m) = e$ *then* $\mathsf{A}_{\mathsf{body}}(C, m, T) = (e^\circ, n)$*, where* e° *is an annotation of* e *and* $n \in \mathbb{D}$*. Moreover, for each view* s *that appears in* e° *holds* $s \in \mathscr{V}$.

Definition 4.2.2 *Let* $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *be a RAJA program with explicit types. We say that* $\mathscr{R}^+$ *is* complete *if for each class* $C \in \mathscr{C}$ *and* $m \in \mathsf{Meth}(C)$ *holds if* $T\ \mathsf{instanceof}\ \mathsf{M}(C, m)$ *then* $T \in \mathsf{Inst}(C, m)$.

We do not require all RAJA programs with explicit types to be complete since, in most cases, this would mean that the map Inst is infinite, which would preclude the type checking algorithm from Section 4.4 from terminating. However, if the RAJA program with explicit types is complete and well-typed, we can prove that its underlying RAJA program is also well-typed.

We define subtyping between sets of monomorphic RAJA method types as follows.

Definition 4.2.3 (Subtyping of explicit types) *Let $C, D \in \mathscr{C}$ with $C <: D$ and $m \in \mathsf{Meth}(C)$. Then $\mathsf{Inst}(C, m) <: \mathsf{Inst}(D, m)$ iff*

$$\forall\, T' \in \mathsf{Inst}(D, m)\, \exists\, T \in \mathsf{Inst}(C, m)\, .\, T <: T'$$

Notice that this definition of subtyping coincide with Def. 3.2.16 when the set $\mathsf{Inst}(C, m)$ is equal to the set of instances of $\mathsf{M}(C, m)$.

4.3 Algorithmic views

In this section we define algorithmic views that include operations to construct the intermediate views r_1 and r_2 in the declarative rule for method typing ($\Diamond MBody$), since the mentioned rule gives no information about how to find them. We define the views n_C and $(s \dot{-} \mathsf{n})_\mathsf{C}$ for this purpose.

The view n_C is defined such that the potential of C under n_C is n and the potential of any other class D under n_C is 0. It is intended to be used together with the view $(s \dot{-} \mathsf{n})_\mathsf{C}$, which is nothing but the view s, with n units of potential stripped-off in the class C. This way, we can split s into $(s \dot{-} \mathsf{n})_\mathsf{C}$ and n_C, such that $\curlyvee\big(s \,\big|(s \dot{-} \mathsf{n})_\mathsf{C}, \mathsf{n}_\mathsf{C}\big)$ holds. If we need to use n units of potential from the type C^s of this in the method body of a given method, we give this the type $C^{(s \dot{-} \mathsf{n})_\mathsf{C}}$ and use the potential of $C^{\mathsf{n}_\mathsf{C}}$ in the method.

Recall the copy method from Fig. 3.1. In order to use the potential of the variable this of RAJA type $\mathsf{Cons}^{\mathsf{rich}}$, we can put it in the context with the type $\mathsf{Cons}^{(\mathsf{rich} \dot{-} 1)\,\mathsf{Cons}}$, then we can derive:

$$\mathsf{this}\colon \mathsf{Cons}^{(\mathsf{rich} \dot{-} 1)\,\mathsf{Cons}} \vdash \frac{\overbrace{0 + \Diamond(\mathsf{Cons}^{1\,\mathsf{Cons}})}^{1}}{0}\ \mathsf{let\,List\ res} = \mathsf{new\ Cons}^{\mathsf{poor}}\ \mathsf{in}\ \ldots\ \mathsf{in\ return\ res;}$$

If $s \in \mathscr{V}^{\mathscr{C}}$, $n \in \mathbb{D}$ and $D \in \mathscr{C}$, we then define the algorithmic views n_D, n^-_D, $(s \dot{-} \mathsf{n})_\mathsf{D}$ coinductively as shown in Fig. 4.2. In the following we prove the desirable properties of algorithmic views.

For each $C, D \in \mathscr{C}$ and $a \in \mathsf{A}(C)$ we set:

$$\begin{array}{rcl}
\Diamond\!\!\left(C^{\mathsf{n}_\mathsf{D}}\right) & = & \begin{cases} n & \text{if } C = D \\ 0 & \text{otherwise} \end{cases} \\
\Diamond\!\!\left(C^{\mathsf{n}_\mathsf{D}^-}\right) & = & \begin{cases} n & \text{if } C = D \\ \infty & \text{otherwise} \end{cases} \\
\Diamond\!\!\left(C^{(s \dot{-} \mathsf{n})_\mathsf{D}}\right) & = & \begin{cases} \Diamond\!(C^s) \dot{-} n & \text{if } C = D \\ \Diamond\!(C^s) & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{A}^{\mathrm{get}}(C^{\mathsf{n}_\mathsf{D}}, a) & = & \mathsf{0}_\mathsf{D} \\
\mathsf{A}^{\mathrm{get}}\!\left(C^{\mathsf{n}_\mathsf{D}^-}, a\right) & = & \infty_\mathsf{D}^- \\
\mathsf{A}^{\mathrm{get}}\!\left(C^{(s \dot{-} \mathsf{n})_\mathsf{D}}, a\right) & = & \mathsf{A}^{\mathrm{get}}(C^s, a) \\
\\
\mathsf{A}^{\mathrm{set}}(C^{\mathsf{n}_\mathsf{D}}, a) & = & \infty_\mathsf{D}^- \\
\mathsf{A}^{\mathrm{set}}\!\left(C^{\mathsf{n}_\mathsf{D}^-}, a\right) & = & \mathsf{0}_\mathsf{D} \\
\mathsf{A}^{\mathrm{set}}\!\left(C^{(s \dot{-} \mathsf{n})_\mathsf{D}}, a\right) & = & \mathsf{A}^{\mathrm{set}}(C^s, a)
\end{array}$$

where for $n, n' \in \mathbb{D}$, $n \dot{-} n'$ is defined by

$$n \dot{-} n' = \begin{cases} n - n' & \text{if } n \geq n' \\ 0 & \text{otherwise} \end{cases}$$

Figure 4.2: Definition of algorithmic views.

Lemma 4.3.1 *Let $C \in \mathscr{C}$ and $s \in \mathscr{V}^{\mathscr{C}}$ and $n \in \mathbb{D}$. Then*

1. $s \oplus \mathsf{0}_\mathsf{C} = s$.
2. $s \boxplus \infty_\mathsf{C}^- = s$.

Proof. Simultaneously by coinduction.

1. We prove simultaneously $s \oplus \mathsf{0}_\mathsf{C} \sqsubseteq s$ and $s \sqsubseteq s \oplus \mathsf{0}_\mathsf{C}$. (3.2.1) follows by $\Diamond\!(C^s) + 0 = \Diamond\!(C^s)$. (3.2.2) follows by coinduction hypothesis (1). (3.2.3) follows by coinduction hypothesis (2).
2. We prove simultaneously $s \boxplus \infty_\mathsf{C}^- \sqsubseteq s$ and $s \sqsubseteq s \boxplus \infty_\mathsf{C}^-$. (3.2.1) follows by $\min(\Diamond\!(C^s), \infty) = \Diamond\!(C^s)$. (3.2.2) follows by coinduction hypothesis (2). (3.2.3) follows by coinduction hypothesis (1).

□

Lemma 4.3.2 *Let $C \in \mathscr{C}$ and $s \in \mathscr{V}^{\mathscr{C}}$ and $n \in \mathbb{D}$. Then:*

1. *If $\Diamond\!(C^s) \geq n$ then $s \sqsubseteq (s \dot{-} \mathsf{n})_\mathsf{C}$.*
2. *If $\Diamond\!(C^s) \geq n$ then $s \sqsubseteq \mathsf{n}_\mathsf{C}$.*
3. *If $\Diamond\!(C^s) \leq n$ then $\mathsf{n}_\mathsf{C}^- \sqsubseteq s$.*

Proof. Let $D \in \mathscr{C}$ and $a \in \mathsf{A}(D)$.

1. We show (3.2.1). In case $C = D$ the goal follows by $\Diamond\!\!\downarrow(D^s) \geq \Diamond\!\!\downarrow(D^s) \dot{-} n$ which follows by assumption. Otherwise the goal follows by $\Diamond\!\!\downarrow(D^s) \geq \Diamond\!\!\downarrow(D^s)$. (3.2.2) and (3.2.3) follow trivially.

   Next we show 2. and 3. simultaneously by coinduction.

2. We show (3.2.1). In case $C = D$ the goal follows by $\Diamond\!\!\downarrow(D^s) \geq n$ which follows by assumption, otherwise it follows from $\Diamond\!\!\downarrow(D^s) \geq 0$.

   (3.2.2) follows by the coinduction hypothesis (2). (3.2.3) follows by the coinduction hypothesis (3).

3. We show (3.2.1). In case $C = D$ the goal follows by $\Diamond\!\!\downarrow(D^s) \leq n$ which follows by assumption, otherwise it follows from $\Diamond\!\!\downarrow(D^s) \leq \infty$.

   (3.2.2) follows by the coinduction hypothesis (3). (3.2.3) follows by the coinduction hypothesis (2).

□

Lemma 4.3.3 *Let $C \in \mathscr{C}$ and $s, s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$ and $n \in \mathbb{D}$.*

1. *If $\Diamond\!\!\downarrow(C^s) \geq n$ then $\curlyvee\big(s \,\big|\, (s \dot{-} \mathsf{n})_{\mathsf{C}}, \mathsf{n}_{\mathsf{C}}\big)$.*
2. *If $\Diamond\!\!\downarrow(C^s) \leq n$ then $\curlywedge\big(s \,\big|\, s, \mathsf{n}^-_{\mathsf{C}}\big)$.*

Proof. Simultaneously by coinduction.

1. Let $D \in \mathscr{C}$ and $a \in \mathsf{A}(D)$. We show (3.2.10), i.e. $\Diamond\!\!\downarrow(D^s) \geq \Diamond\!\!\downarrow\Big(D^{(s \dot{-} \mathsf{n})_{\mathsf{C}}}\Big) + \Diamond\!\!\downarrow(D^{\mathsf{n}_{\mathsf{C}}})$ and get the following cases:

   Case $C = D$, then the goal follows by $\Diamond\!\!\downarrow(D^s) \geq \Diamond\!\!\downarrow(D^s) \dot{-} n + n$, which follows by assumption.

   Case $C \neq D$, then the goal follows by $\Diamond\!\!\downarrow(D^s) \geq \Diamond\!\!\downarrow(D^s) + 0$.

   (3.2.11) follows by Lemma 4.3.2.

   (3.2.12) follows from $\curlyvee\big(\mathsf{A}^{\mathrm{get}}(D^s_?, a) \,\big|\, \mathsf{A}^{\mathrm{get}}(D^s_?, a), 0_{\mathsf{C}}\big)$ which follows by coinduction hypothesis (1), since for all views s holds $\Diamond\!\!\downarrow(D^s) \geq 0$ and $s = (s \dot{-} 0)_{\mathsf{C}}$.

   (3.2.13) follows by $\curlywedge\big(\mathsf{A}^{\mathrm{set}}(D^s_?, a) \,\big|\, \mathsf{A}^{\mathrm{set}}(D^s_?, a), \infty^-_{\mathsf{C}}\big)$ which follows by coinduction hypothesis (2).

2. We show (3.2.14), i.e. $\Diamond\!\!\downarrow(D^s) \leq \min(\Diamond\!\!\downarrow\Big(D^{(s \dot{-} \mathsf{n})_{\mathsf{C}}}\Big), \Diamond\!\!\downarrow(D^{\mathsf{n}_{\mathsf{C}}}))$ and get the following cases:

   Case $C = D$, then the goal follows by $\Diamond\!\!\downarrow(D^s) \leq \Diamond\!\!\downarrow(D^s)$ since $\Diamond\!\!\downarrow(D^s) \leq n$ by assumption.

Case $C \neq D$, then the goal follows by $\Diamond(D^s) \leq \min(\Diamond(D^s), \infty)$.

(3.2.15) follows by Lemma 4.3.2.

(3.2.16) follows from $\curlywedge\big(\mathsf{A}^{\mathrm{get}}(D^s_{\mathsf{C}}, a) \,\big|\, \mathsf{A}^{\mathrm{get}}(D^s_{\mathsf{C}}, a), \infty^-_{\mathsf{C}}\big)$ which follows by coinduction hypothesis (2), since for all views s holds $\Diamond(D^s) \leq \infty$.

(3.2.17) follows by $\curlyvee\big(\mathsf{A}^{\mathrm{set}}(D^s_{\mathsf{C}}, a) \,\big|\, \mathsf{A}^{\mathrm{set}}(D^s_{\mathsf{C}}, a), 0_{\mathsf{C}}\big)$ which follows by coinduction hypothesis (1).

□

Lemma 4.3.4 *Let $C \in \mathscr{C}$ and $s, s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$.*

1. *Then $\curlyvee(s \,|\, s_1, s_2)$ and $\Diamond(C^{s_2}) \geq n$ imply $(s \dot{-} \mathsf{n})_{\mathsf{C}} \sqsubseteq s_1$.*
2. *Then $s \sqsubseteq s_1$ and $\Diamond(C^s) \geq \Diamond(C^{s_1}) + n$ imply $(s \dot{-} \mathsf{n})_{\mathsf{C}} \sqsubseteq s_1$.*
3. *$(s \dot{-} \mathsf{n})_{\mathsf{C}} \sqsubseteq s_1$ implies $s \sqsubseteq s_1$.*

Proof.

1. Let $D \in \mathscr{C}$ and $a \in \mathsf{A}(D)$. We show (3.2.1), i.e. $\Diamond\Big(D^{(s \dot{-} \mathsf{n})_{\mathsf{C}}}\Big) \geq \Diamond(D^{s_1})$ and we obtain the following cases:

   Case $C = D$, then the goal follows by $\Diamond(D^s) \dot{-} n \geq \Diamond(D^{s_1})$ which follows by $\Diamond(D^s) \geq \Diamond(D^{s_1}) + \Diamond(D^{s_2})$ and $\Diamond(D^{s_2}) \geq n$ that follow by assumption.

   Case $C \neq D$, then the goal follows by $\Diamond(D^s) \geq \Diamond(D^{s_1})$ which follows by assumption.

   (3.2.2) follows by $\mathsf{A}^{\mathrm{get}}(D^s_{\mathsf{C}}, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^{s_1}_{\mathsf{C}}, a)$ that follows by $\curlyvee\big(\mathsf{A}^{\mathrm{get}}(D^s_{\mathsf{C}}, a) \,\big|\, \mathsf{A}^{\mathrm{get}}(D^{s_1}_{\mathsf{C}}, a), \mathsf{A}^{\mathrm{get}}(D^{s_2}_{\mathsf{C}}, a)\big)$ which follows by assumption.

   (3.2.3) follows by $\mathsf{A}^{\mathrm{set}}(D^{s_1}_{\mathsf{C}}, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(D^s_{\mathsf{C}}, a)$ that follows by $\curlywedge\big(\mathsf{A}^{\mathrm{set}}(D^s_{\mathsf{C}}, a) \,\big|\, \mathsf{A}^{\mathrm{set}}(D^{s_1}_{\mathsf{C}}, a), \mathsf{A}^{\mathrm{set}}(D^{s_2}_{\mathsf{C}}, a)\big)$ which follows by assumption.

2. Similar.

3. It follows by the fact that $\Diamond(C^s) \dot{-} n \geq \Diamond(C^{s_1})$ implies $\Diamond(C^s) \geq \Diamond(C^{s_1})$.

□

4.4 Typing

In this section we define when RAJA programs with explicit types are well-typed, by giving syntax-directed typing rules for annotated expressions and a rule for typing method bodies. Then, we will show that a RAJA program is well-typed iff it can be extended to a complete well-typed RAJA program with explicit types.

4.4.1 Syntax-directed typing rules

We wish to define syntax-directed typing rules for annotated FJEU expressions based on the rules from Chapter 3, thus, the rules ($\Diamond Share$) and ($\Diamond Waste$) must be integrated in the remaining rules. Instead of using ($\Diamond Waste$), we integrate subtyping in the rules.

The advantage of annotating all the variable occurrences of an annotated expression with a view is that the rule ($\Diamond Share$) can be integrated easily into the rule ($\Diamond Let$). The task of the type checker is to check that the potential available in the refined type of a variable can be split into the potential of all the views corresponding to each occurrence of the variable. More concretely, if the type of x is C^r, and x is used twice in e, and the first occurrence of x is annotated with the view s_1, and the second occurrence with the view s_2, then $\curlyvee(r\,|s_1, s_2\,)$ most hold. However, recall that we proved in last chapter that $\curlyvee(r\,|s_1, s_2\,)$ is equivalent to $r \sqsubseteq s_1 \oplus s_2$. Thus, the type checker builds $s_1 \oplus s_2$ and checks subtyping. Notice that we do not integrate ($\Diamond Share$) in other rules like ($\Diamond Invocation$) or ($\Diamond Update$), since we require that in those expressions a variable appears only once, as explained in Chapter 2.

We define the judgement $\Gamma \vdash^{n}_{n'} e^\circ \leftleftarrows C^r$ inductively by the rules in Fig. 4.3, where $|\Gamma|$, e°, C^r and n, n' are inputs and the multiset of views $\langle\!\langle \Gamma \rangle\!\rangle$ is output. Recall that if Γ is a RAJA context, we write $|\Gamma|$ for meaning its underlying FJEU context. Moreover, e° is an annotated FJEU expression and $|e^\circ|$ is the same FJEU expression with all the annotations removed. We sometimes write e for meaning $|e^\circ|$ in this case.

We write Γ_0 for meaning $\Gamma_0(x) = C^{0\mathsf{c}}$ for each $x \in \Gamma_0$. The idea is to annotate the types of the variables that are not used in the given expression with views with potential 0.

Moreover, let Γ_1 and Γ_2 be two RAJA contexts with $|\Gamma_1| = |\Gamma_2|$, then $\Gamma_1 \oplus \Gamma_2$ and $\Gamma_1 \wedge \Gamma_2$ are RAJA contexts defined as follows:

$$\begin{array}{lcl} (\Gamma_1 \oplus \Gamma_2)(x) & = & |\Gamma_1|^{\langle \Gamma_1(x)\rangle \oplus \langle \Gamma_2(x)\rangle} \\ (\Gamma_1 \wedge \Gamma_2)(x) & = & |\Gamma_1|^{\langle \Gamma_1(x)\rangle \wedge \langle \Gamma_2(x)\rangle} \end{array}$$

In summary, we define the partial function $\mathsf{typecheck}(|\Gamma|, e^\circ, C^r, n, n')$ as follows.

$$\mathsf{typecheck}(|\Gamma|, e^\circ, C^r, n, n') = \begin{cases} (\langle\!\langle \Gamma \rangle\!\rangle) & \text{if } \Gamma \vdash^{n}_{n'} e^\circ \leftleftarrows C^r \\ \mathsf{fail} & \text{otherw.} \end{cases}$$

In Fig. 4.3 we also define the algorithmic judgement $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$ for checking that the method's body of m admits all the types in $\mathsf{Inst}(C, m)$. The type checking algorithm returns a greater context than the declared one. This has to be checked. Moreover, the annotation p represents the amount of items of potential that we take from the potential of the refined type of this in the type T for using in the method's body. Thus, we need to check that the potential of the refined type of this is at least p.

Algorithmic RAJA Typing $\boxed{\Gamma \mid\!\!\frac{n}{n'}\ e^\circ \leftleftarrows C^r}$

$$\frac{\begin{array}{c}\forall a \in \mathsf{A}(D)\ .\ \mathsf{A}^{\mathrm{set}}(D^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^r, a) \\ D^r <: C^r \qquad n \geq \langle\!(D^r)\rangle\!+1 \qquad n' \leq n - \langle\!(D^r)\rangle\!-1\end{array}}{\Gamma_0 \mid\!\!\frac{n}{n'}\ \mathsf{new}\ D \leftleftarrows C^r}\ (\triangledown New)$$

$$\frac{n' \leq n + \min\{\langle\!(D^r)\rangle\! \mid D <: C\} + 1}{\Gamma_0, x{:}C^r \mid\!\!\frac{n}{n'}\ \mathsf{free}\,(x^r) \leftleftarrows E^s}\ (\triangledown Free)$$

$$\frac{D <: E \qquad D^r <: C^s \qquad n' \leq n}{\Gamma_0, x{:}E^r \mid\!\!\frac{n}{n'}\ (D)x^r \leftleftarrows C^s}\ (\triangledown Cast) \qquad \frac{n' \leq n}{\Gamma_0 \mid\!\!\frac{n}{n'}\ \mathsf{null} \leftleftarrows C^s}\ (\triangledown Null)$$

$$\frac{E^r <: C^s \qquad n' \leq n}{\Gamma_0, x{:}E^r \mid\!\!\frac{n}{n'}\ x^r \leftleftarrows C^s}\ (\triangledown Var)$$

$$\frac{\forall F <: C\ .\ \mathsf{A}^{\mathrm{get}}(F^r, a) \sqsubseteq s \qquad C.a = E \qquad E <: D \qquad n' \leq n}{\Gamma_0, x{:}C^r \mid\!\!\frac{n}{n'}\ x^r.a \leftleftarrows D^s}\ (\triangledown Access)$$

$$\frac{\forall G <: E\ .\ s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r, a) \qquad E.a = D \qquad F <: D \qquad E^r <: C^q \qquad n' \leq n}{\Gamma_0, x{:}E^r, y{:}F^s \mid\!\!\frac{n}{n'}\ x^r.a \leftarrow y^s \leftleftarrows C^q}\ (\triangledown Update)$$

$$\frac{\Gamma_1 \mid\!\!\frac{n}{n'}\ e_1^\circ \leftleftarrows D^s \quad \Gamma_2, x{:}D^s \mid\!\!\frac{n'}{n''}\ e_2^\circ \leftleftarrows C^r}{\Gamma_1 \oplus \Gamma_2 \mid\!\!\frac{n}{n''}\ \mathsf{let}\ D\,x^{n'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r}\ (\triangledown Let)$$

$$\frac{x \in \Gamma \quad \Gamma_1 \mid\!\!\frac{n}{n'}\ e_1^\circ \leftleftarrows C^r \quad \Gamma_2 \mid\!\!\frac{n}{n'}\ e_2^\circ \leftleftarrows C^r}{\Gamma_1 \wedge \Gamma_2 \mid\!\!\frac{n}{n'}\ \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r}\ (\triangledown Cond.)$$

$$\frac{\begin{array}{c}\left(G^{s_0}; \vec{E}^{\vec{s}\,m/m'} \longrightarrow H^{s'}\right) \in \mathsf{Inst}(G, m) \\ G^{r_0} <: G^{s_0} \qquad F_i^{r_i} <: E_i^{s_i} \qquad H^{s'} <: C^{r'} \qquad n \geq m \qquad n' \leq m' + n - m\end{array}}{\Gamma_0, x{:}G^{r_0}, y_1{:}F_1^{r_1}, \ldots, y_n{:}F_n^{r_n} \mid\!\!\frac{n}{n'}\ x^{r_0}.m\,(y_1^{r_1}, \ldots, y_n^{r_n}) \leftleftarrows C^{r'}}\ (\triangledown Inv.)$$

Algorithmic RAJA Method Typing $\boxed{\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}}$

$$\frac{\begin{array}{c}T = C^{r_0}; \vec{E}^{\vec{\mathsf{r}}\,n/n'} \longrightarrow H^{r_{n+1}} \in \mathsf{Inst}(C, m) \qquad (r_0 \doteq \mathsf{p})_{\mathsf{C}} \sqsubseteq s_0 \quad r_i \sqsubseteq s_i \qquad \langle\!(C^{r_0})\rangle\! \geq p \\ \mathsf{A}_{\mathsf{body}}(C, m, T) = (e^\circ, p) \qquad \mathtt{this}{:}C^{s_0},\ x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \mid\!\!\frac{n+p}{n'}\ e^\circ \leftleftarrows H^{r_{n+1}}\end{array}}{\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}}$$

Figure 4.3: Algorithmic RAJA Typing.

RAJA programs with explicit types are well-typed if for each method m in a class C holds $m : \mathsf{Inst}(C, m)\ \mathsf{ok}$ and the set of types of m in C is a subtype of the set of types of m in the super-class of C.

Definition 4.4.1 (Well-typed RAJA-program with explicit types)

A RAJA-program $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *is* well-typed *if the following conditions are satisfied:*

1. $\forall C \in \mathscr{C}, m \in \mathsf{Meth}(C)\,.\ \vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$
2. $\forall C, D \in \mathscr{C}$ *with* $S(C) = D \Rightarrow \mathsf{Inst}(C, m) <: \mathsf{Inst}(D, m)$.

4.4.2 Verification of correctness of typing

In this section we will show that a RAJA program is well-typed iff it can be extended to a complete well-typed RAJA program with explicit types.

Verification of soundness

In the following we show that the syntax-directed typing judgement defined in Section 4.4.1 is sound w.r.t. the declarative typing judgement from Chapter 3. Notice that this holds even for RAJA programs with explicit types that are not complete.

In the proof we will use the fact that $\Gamma <: \varnothing$ for any context Γ without explicit notice.

Lemma 4.4.2 (Soundness of algorithmic RAJA typing)
Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *be a RAJA program with explicit types. If* $\mathcal{D} :: \Gamma \vdash^{n}_{n'} e^{\circ} \leftleftarrows C^r$ *then* $\Gamma \vdash^{n}_{n'} |e^{\circ}| : C^r$.

Proof. By induction on $\mathcal{D}$.

Case $(\triangledown New)$. We have $D^r <: C^r$ and $D^r\ \mathsf{main}$ and $n \geq \Diamond\!(D^r) + 1$ and $n' \leq n - \Diamond\!(D^r) - 1$. Then, by $(\Diamond New)$ followed by $(\Diamond Waste)$, we obtain the desired goal.

$$\dfrac{\dfrac{D^r\ \mathsf{main}}{\varnothing \vdash^{\Diamond\!(D^r)+1}_{0} \mathtt{new}\ D : D^r \qquad D^r <: C^r}\,(\Diamond New) \qquad \Gamma_0 <: \varnothing \qquad n \geq \Diamond\!(D^r) + 1 \qquad n' \leq n - \Diamond\!(D^r) - 1}{\Gamma_0 \vdash^{n}_{n'} \mathtt{new}\ D : C^r}\,(\Diamond Waste)$$

Case $(\triangledown Free)$. Similarly, by $(\Diamond Free)$ followed by $(\Diamond Waste)$.

Case $(\triangledown Cast)$. We have $D^r <: C^s$ and $D <: E$ and $n' \leq n$. Then, by $(\Diamond Cast)$ followed by $(\Diamond Waste)$, we obtain the desired goal.

$$\frac{D <: E}{\dfrac{x{:}E^r \;\vdash^0_0\; (D)x : D^r \qquad D^r <: C^s \qquad \Gamma_0, x{:}E^r <: x{:}E^r \qquad n' \leq n}{\Gamma_0, x{:}E^r \;\vdash^n_{n'}\; (D)x : C^s}}$$

Case ($\triangledown Var$). Follows similarly, by ($\lozenge Var$) followed by ($\lozenge Waste$).

Case ($\triangledown Access$). We have $C.a = E$ and $\forall F <: C \,.\, \mathsf{A}^{\mathrm{get}}(F^r, a) \sqsubseteq s$ and $E <: D$ and $n' \leq n$. Then, by ($\lozenge Access$) followed by ($\lozenge Waste$), we obtain the desired goal.

$$\frac{\forall F <: C \,.\, \mathsf{A}^{\mathrm{get}}(F^r, a) \sqsubseteq s \qquad C.a = E}{\dfrac{x{:}C^r \;\vdash^0_0\; x.a : E^s \qquad E^s <: D^s \qquad \Gamma_0, x{:}C^r <: x{:}C^r \qquad n' \leq n}{\Gamma_0, x{:}C^r \;\vdash^n_{n'}\; x.a : D^s}}$$

Case ($\triangledown Update$). We have $E.a = D$ and $\forall G <: E \,.\, s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r, a)$ and $F <: D$ and $E^r <: C^q$. Then, by ($\triangledown Update$) followed by ($\lozenge Waste$), we obtain the desired goal.

$$\frac{\forall G <: E \,.\, s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r, a) \qquad E.a = D}{\dfrac{x{:}E^r, y{:}D^s \;\vdash^0_0\; x.a \leftarrow y{:}E^r \qquad E^r <: C^q \qquad F^s <: D^s \qquad n' \leq n}{\Gamma_0, x{:}E^r, y{:}F^s \;\vdash^n_{n'}\; x.a \leftarrow y : C^q}}$$

Case ($\triangledown Let$). For ease of notation let us assume w.l.o.g. $\Gamma_1 = y{:}C^{r_1}$ and $\Gamma_2 = y{:}C^{r_2}$. By induction hypothesis we obtain $y{:}C^{r_1} \;\vdash^n_{n'}\; e_1{:}D^s$ and $y{:}C^{r_2}, x{:}D^s \;\vdash^{n'}_{n''}\; e_2{:}C^r$. After substituting y by y_1 in e_1 and y by y_2 in e_2 we obtain $y_1{:}C^{r_1} \;\vdash^n_{n'}\; e_1\,[y_1/y]{:}D^s$ and $y_2{:}C^{r_2}, x{:}D^s \;\vdash^{n'}_{n''}\; e_2\,[y_2/y]{:}C^r$. Then, by ($\lozenge Let$), we obtain

$$\frac{y_1{:}C^{r_1} \;\vdash^n_{n'}\; e_1\,[y_1/y]{:}D^s \qquad y_2{:}C^{r_2}, x{:}D^s \;\vdash^{n'}_{n''}\; e_2\,[y_2/y]{:}C^r}{y_1{:}C^{r_1}, y_2{:}C^{r_2} \;\vdash^n_{n''}\; \mathsf{let}\ D\,x = e_1\,[y_1/y]\ \mathsf{in}\ e_2\,[y_2/y]{:}C^r}\ (\triangledown Let)$$

Then, by ($\lozenge Share$), we obtain the desired goal

$$y{:}C^{r_1 \oplus r_2} \;\vdash^n_{n''}\; \mathsf{let}\ D\,x = e_1\ \mathsf{in}\ e_2{:}C^r$$

since, by Lemma 3.2.25, we obtain $\curlyvee(r_1 \oplus r_2 \,|\, r_1, r_2)$.

Case ($\triangledown Cond.$). Let us assume again for ease of notation w.l.o.g. $\Gamma_1 = y{:}C^{s_1}$ and $\Gamma_2 = y{:}C^{s_2}$. By induction hypothesis we obtain $y{:}C^{s_1} \;\vdash^n_{n'}\; e_1{:}C^r$ and $y{:}C^{s_2} \;\vdash^n_{n'}\; e_2{:}C^r$. We notice that $s_1 \wedge s_2 \sqsubseteq s_i$ by Lemma 3.2.3. Then we obtain the desired goal by ($\lozenge Waste$) and ($\lozenge Conditional$).

$$\frac{\dfrac{y{:}C^{s_1} \;\vdash^n_{n'}\; e_1{:}C^r}{y{:}C^{s_1 \wedge s_2} \;\vdash^n_{n'}\; e_1{:}C^r}\ (\lozenge Waste) \qquad \dfrac{y{:}C^{s_2} \;\vdash^n_{n'}\; e_2{:}C^r}{y{:}C^{s_1 \wedge s_2} \;\vdash^n_{n'}\; e_2{:}C^r}\ (\lozenge Waste)}{y{:}C^{s_1 \wedge s_2} \;\vdash^n_{n'}\; \mathsf{if}\ x\ \mathsf{instanceof}\ C\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2{:}C^r}\ (\lozenge Conditional)$$

Case ($\triangledown Inv.$). We have $G^{s_0}; \vec{E}^{\vec{s}}\xrightarrow{m/m'} H^{s'} \in \mathsf{Inst}(G, m)$ and $G^{r_0} <: G^{s_0}$ and $F_i^{r_i} <: E_i^{s_i}$ and $H^{s'} <: C^{r'}$ and $n \geq m$ and $n' \leq m' + n - m$. Then, the desired goal follows by ($\Diamond Invocation$) and ($\Diamond Waste$).

$$\dfrac{\dfrac{G^{s_0}; \vec{E}^{\vec{s}}\xrightarrow{m/m'} H^{s'} \ \mathsf{instanceof}\ \mathsf{M}(G,m)}{x{:}G^{s_0}, y_1{:}E_1^{s_1}, \ldots, y_n{:}E_n^{s_n} \ \vdash^{m}_{m'}\ {:}H^{s'} \qquad G^{r_0} <: G^{s_0} \qquad F_i^{r_i} <: E_i^{s_i} \quad \ldots}}{\Gamma_0, x{:}G^{r_0}, y_1{:}F_1^{r_1}, \ldots, y_n{:}F_n^{r_n} \ \vdash^{n}_{n'}\ {:}C^{r'}}$$

□

Next we show that if $\mathscr{R}^+$ is a well-typed complete RAJA program with explicit types then its underlying RAJA program is well-typed as well.

Lemma 4.4.3 (Soundness of algorithmic RAJA method typing) *Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A_{body}})$ be a complete RAJA program with explicit types. Let $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$ such that $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$. Then $\vdash m : \mathsf{M}(C, m)\ \mathsf{ok}$.*

Proof. Let $T = \vec{E}^{\vec{r}}\xrightarrow{n/n'} H^{r_{n+1}} \in \mathsf{Inst}(C, m)$ and $\mathsf{A_{body}}(C, m, T) = (e^\circ, p)$. Then, $\mathsf{M_{body}}(C, m) = |e^\circ|$. We have

$$\mathtt{this}{:}C^{s_0}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \ \vdash^{n+p}_{n'}\ e^\circ \leftleftarrows H^{r_{n+1}}$$

$\Diamond(C^{r_0}) \geq p$ and $r_i \sqsubseteq s_i$ and $(r_0 \dot{-} \mathsf{p})_\mathsf{C} \sqsubseteq s_0$. Then, by Lemma 4.3.3 we get $\curlyvee\big(r_0 \,\big|(r_0 \dot{-} \mathsf{p})_\mathsf{C}, \mathsf{p_C}\big)$. We then get the desired goal by soundness of algorithmic typing (Lemma 4.4.2) and ($\Diamond Waste$):

$$\dfrac{\begin{array}{c}\mathtt{this}{:}C^{s_0}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \ \vdash^{n+p}_{n'}\ |e^\circ| : H^{r_{n+1}} \\ C^{(r_0 \dot{-} \mathsf{p})_\mathsf{C}} <: C^{s_0} \qquad E_i^{r_i} <: E_i^{s_i}\end{array}}{\mathtt{this}{:}C^{(r_0 \dot{-} \mathsf{p})_\mathsf{C}}, x_1{:}E_1^{r_1}, \ldots, x_n{:}E_n^{r_n} \ \vdash^{n+p}_{n'}\ |e^\circ| : H^{r_{n+1}}}\ (\Diamond Waste)$$

□

Theorem 4.4.4 *Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A_{body}})$ be a complete RAJA program with explicit types. Then $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ is well-typed.*

Proof. Let $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$ and $\mathsf{M}(C, m) = \phi$ and $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$. Then $\vdash m : \phi\ \mathsf{ok}$ by Lemma 4.4.3. Let further $\mathsf{S}(C) = D$. Then $\mathsf{M}(C, m) <: \mathsf{M}(D, m)$ follows by $\mathsf{Inst}(C, m) <: \mathsf{Inst}(D, m)$ because $\mathscr{R}^+$ is complete. □

Verification of completeness

In this section we show that given a well-typed RAJA program, we can extend it to a complete RAJA program with explicit types that is well-typed as well.

First, we show that the syntax-directed typing judgement defined in Section 4.4.1 is complete w.r.t. the declarative typing judgement from Chapter 3. The completeness proof is a bit more complicated than the soundness proof. The reason for this is that we have eliminated the rules $(\Diamond Share)$ and $(\Diamond Waste)$ and we have to show that typing derivations that use these rules are still admissible in the syntax-directed typing system.

The following lemma states the admissibility of sharing in the syntax-directed typing system.

Lemma 4.4.5 (Share)
Let $\mathcal{D} :: \Gamma, y_1 : D^{s_1}, \ldots, y_n : D^{s_n} \vdash^n_{n'} e^\circ \leftleftarrows C^r$ *and* $\curlyvee(s\,|s_1, \ldots, s_n)$. *Then* $\Gamma, x : D^{\bar{s}} \vdash^n_{n'} e[x/y_1, \ldots, x/y_n]^\circ \leftleftarrows C^r$ *and* $s \sqsubseteq \bar{s}$.

Proof. By induction on $\mathcal{D}$. For ease of notation let us assume $\Gamma = \varnothing$ and $n = 2$.

Case $(\triangledown Let)$. We have

$$\frac{y_1 : D^{s_1}, y_2 : D^{s_2} \vdash^n_{n'} e_1^\circ \leftleftarrows D^q \qquad y_1 : D^{r_1}, y_2 : D^{r_2}, x : D^q \vdash^n_{n'} e_2^\circ \leftleftarrows C^r}{y_1 : D^{s_1 \oplus r_1}, y_2 : D^{s_2 \oplus r_2} \vdash^n_{n'} \mathsf{let}\ D\,x^{n'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r}\ (\triangledown Let)$$

and $\curlyvee(s\,|s_1 \oplus r_1, s_2 \oplus r_2)$. Also notice that $\curlyvee(s_1 \oplus s_2\,|s_1, s_2)$ and $\curlyvee(r_1 \oplus r_2\,|r_1, r_2)$, so by induction hypothesis we obtain $s_1 \oplus s_2 \sqsubseteq \hat{s}$ and $r_1 \oplus r_2 \sqsubseteq \hat{r}$ and

$$\frac{z : D^{\hat{s}} \vdash^n_{n'} e_1[z/y_1, z/y_2] \leftleftarrows D^q \quad z : D^{\hat{r}}, x : D^q \vdash^{n'}_{n''} e_2[z/y_1, z/y_2] \leftleftarrows C^r}{z : D^{\hat{s} \oplus \hat{r}} \vdash^n_{n''} (\,\mathsf{let}\ D\,x = e_1^\circ\ \mathsf{in}\ e_2^\circ)[z/y_1, z/y_2] \leftleftarrows C^r}\ (\triangledown Let)$$

and we show $s \sqsubseteq \hat{s} \oplus \hat{r}$. We compute as follows:

$$\begin{array}{ll} \curlyvee(s\,|s_1 \oplus r_1, s_2 \oplus r_2) & \Longleftrightarrow \\ \curlyvee(s\,|s_1, r_1, s_2, r_2) & \Longleftrightarrow \\ \curlyvee(s\,|s_1 \oplus s_2, r_1 \oplus r_2) & \Longleftrightarrow \text{ Lemma 3.2.24} \\ \curlyvee(s\,|\hat{s}, \hat{r}) & \Longleftrightarrow \text{ Lemma 3.2.25} \\ s \sqsubseteq \hat{s} \oplus \hat{r} & \end{array}$$

Case $\dfrac{y_1 : D^{s_1}, y_2 : D^{s_2} \vdash^n_{n'} e_1^\circ \leftleftarrows C^r \quad y_1 : D^{r_1}, y_2 : D^{r_2} \vdash^n_{n'} e_2^\circ \leftleftarrows C^r}{y_1 : D^{s_1 \wedge r_1}, y_2 : D^{s_2 \wedge r_2} \vdash^n_{n'} \mathsf{if}\ y_1\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r}\ (\triangledown Cond.)$

and $\curlyvee(s\,|s_1 \wedge r_1, s_2 \wedge r_2)$. Also notice that $\curlyvee(s\,|s_1, s_2)$ and $\curlyvee(s\,|r_1, r_2)$, so by induction hypothesis we obtain $s \sqsubseteq \hat{s}$ and $s \sqsubseteq \hat{r}$ and

$$\frac{z : D^{\hat{s}} \vdash^n_{n'} e_1^\circ \leftleftarrows C^r \quad z : D^{\hat{r}} \vdash^n_{n'} e_2^\circ \leftleftarrows C^r}{z : D^{\hat{s} \wedge \hat{r}} \vdash^n_{n'} (\,\mathsf{if}\ z\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ)[z/y_1, z/y_2] \leftleftarrows C^r}\ (\triangledown Cond.)$$

and we show $s \sqsubseteq \hat{s} \wedge \hat{r}$ which follows by Lemma 3.2.4.

□

The next lemma is the main technical lemma of this proof; it states the admissibility of the rule ($\Diamond Waste$) in the syntax-directed typing system, which means admissibility of subtyping and weakening of the potential annotations.

Lemma 4.4.6 (Waste) *Let $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ be a well-typed RAJA program and $\mathsf{Inst}(C, m)$ be the set of instances of $\mathsf{M}(C, m)$ for all $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$.*

If $\mathcal{D} :: \Theta \vdash^{u}_{u'} e^{\circ} \leftleftarrows D^s$ and $D^s <: C^r$ and $|\Gamma| <: |\Theta|$ and $n \geq u$ and $n+u' \geq n'+u$ then there exists an annotation for e such that $\Gamma \vdash^{n}_{n'} e^{\circ} \leftleftarrows C^r$.

Proof. By induction on $\mathcal{D}$.

Case ($\triangledown New$) We have $\Theta_0 \vdash^{u}_{u'} \mathsf{new}\ D \leftleftarrows C^r$ and $D^r <: C^r <: E^p$ and $\forall a \in \mathsf{A}(D)\ .\ \mathsf{A}^{\mathrm{set}}(D^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^r, a)$ and

$$n \geq u \tag{4.4.1}$$
$$n + u' \geq n' + u \tag{4.4.2}$$
$$u \geq \Diamond(D^r) + 1 \tag{4.4.3}$$
$$u' \leq u - (\Diamond(D^r) + 1) \tag{4.4.4}$$

Then $n \geq \Diamond(D^r) + 1$ follows by (4.4.1) and (4.4.3) with transitivity. Moreover, $n' \leq n - (\Diamond(D^r) + 1)$ follows by (4.4.2) and (4.4.4). Thus, we conclude by ($\triangledown New$):

$$\Gamma_0 \vdash^{n}_{n'} \mathsf{new}\ D \leftleftarrows E^p$$

Case ($\triangledown Free$) We have $\Theta_0, x : C^r \vdash^{u}_{u'} \mathtt{free}(x^r) \leftleftarrows E^s$ and $E^s <: F^p$ and $G <: C$ and $m = \min\{\Diamond(D^r) \mid D <: C\}$ and

$$n + u' \geq n' + u \tag{4.4.5}$$
$$u' \leq u + m + 1 \tag{4.4.6}$$

$n' \leq n + m + 1$ follows by (4.4.5) and (4.4.6). Then we conclude with ($\triangledown Free$):

$$\Gamma_0, x : G^r \vdash^{n}_{n'} \mathtt{free}(x^r) \leftleftarrows F^p$$

Case ($\triangledown Cast$) We have $\Theta_0, x : E^r \vdash^{u}_{u'} (D)x^r \leftleftarrows C^s$ and $D <: E$ and $D^r <: C^s <: G^q$ and $F <: E$ and

$$n + u' \geq n' + u \tag{4.4.7}$$
$$u' \leq u \tag{4.4.8}$$

Then $n' \leq n$ follows by (4.4.7) and (4.4.8). Then we finish with ($\triangledown Cast$):

$$\Gamma_0, x : F^r \vdash^{n}_{n'} (D)x^r \leftleftarrows G^q$$

Case $(\triangledown Access)$

$$\frac{\forall I <: C \,.\, \mathsf{A}^{\mathrm{get}}(I^r, a) \sqsubseteq s \qquad C.a = E \qquad E <: D}{\Theta_0, x : C^r \;\vdash^{u}_{u'}\; x^r.a \leftleftarrows D^s} \; (\triangledown Access)$$

and $D^s <: F^p$ and $G <: C$ and

$$n + u' \geq n' + u \tag{4.4.9}$$
$$u' \leq u \tag{4.4.10}$$

Then $n' \leq n$ follows by (4.4.9) and (4.4.10). We obtain by $(\triangledown Access)$:

$$\frac{\forall H <: G \,.\, \mathsf{A}^{\mathrm{get}}(H^r, a) \sqsubseteq s \sqsubseteq p \qquad C.a = E \qquad E <: D}{\Gamma_0, x : G^r \;\vdash^{n}_{n'}\; x^r.a \leftleftarrows F^p} \; (\triangledown Access)$$

Case $(\triangledown Update)$

$$\frac{\forall G <: E \,.\, s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r, a) \qquad E.a = D \qquad F <: D \qquad E^r <: C^q}{\Theta_0, x : E^r, y : F^s \;\vdash^{u}_{u'}\; x^r.a \texttt{<-} y^s \leftleftarrows C^q}$$

and $C^q <: C'^{q'}$ and $E' <: E$ and $F' <: F$ and

$$n + u' \geq n' + u \tag{4.4.11}$$
$$u' \leq u \tag{4.4.12}$$

Then $n' \leq n$ follows by (4.4.11) and (4.4.12). Then we finish with $(\triangledown Update)$:

$$\frac{\forall G <: E' \,.\, s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r, a) \quad E'.a = D \quad F' <: F <: D \quad E^r <: C^q <: C'^{q'}}{\Gamma_0, x : E'^r, y : F'^s \;\vdash^{n}_{n'}\; x^r.a \texttt{<-} y^s \leftleftarrows C'^{q'}}$$

Case $(\triangledown Inv.)$ We have

$$\frac{\begin{array}{c} T' = G^{p_0}; \vec{E}^{\vec{\mathsf{p}}} \xrightarrow{m_1/m_2} H^{p_{n+1}} \in \mathsf{Inst}(G\,, m) \qquad G^{r_0} <: G^{p_0} \\ F_i^{r_i} <: E_i^{p_i} \qquad H^{p_{n+1}} <: C^{r'} \qquad u_1 \geq m_1 \qquad u_2 \leq m_2 + u_1 - m_1 \end{array}}{\Gamma_0, x : G^{r_0}, y_1 : F_1^{r_1}, \ldots, y_n : F_n^{r_n} \;\vdash^{u_1}_{u_2}\; x^{r_0}.m\,(y_1^{r_1}, \ldots, y_n^{r_n}) \leftleftarrows C^{r'}} \; (\triangledown Inv.)$$

Moreover we have $\bar{G} <: G$ and $\bar{F}_i <: F_i$ and $C^{r'} <: \bar{C}^{\bar{r}}$ and $w_1 \geq u_1$ and $w_2 - w_1 \leq u_2 - u_1$.

By definition of subtyping of sets of monomorphic RAJA types we get that there exists $T = s_0; \vec{\mathsf{s}} \xrightarrow{n_1/n_2} s_{n+1} \in \mathsf{Inst}(\bar{G}, m)$ with $T <: T'$, i.e. $n_1 \leq m_1$ and $n_2 \geq m_2$ and $p_0 = s_0$ and $p_i \sqsubseteq s_i$ and $s_{n+1} \sqsubseteq p_{n+1}$.

Next, we check that the following subtyping judgements and inequalities hold:

- $\bar{G}^{r_0} <: G^{p_0} = G^{s_0}$
- $\bar{F}_i^{r_i} <: F_i^{r_i} <: E_i^{p_i} <: E_i^{s_i}$
- $H^{s_{n+1}} <: H^{p_{n+1}} <: C^{r'} <: \bar{C}^{\bar{r}}$
- $w_1 \geq u_1 \geq m_1 \geq n_1$
- $w_2 - w_1 \leq u_2 - u_1 \leq m_2 - m_1 \leq n_2 - m_1 \leq n_2 - n_1$

Then, we can finish by $(\triangledown Inv.)$:

$$\frac{(\bar{G}^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{n_1/n_2} H^{s_{n+1}}) \in \mathsf{Inst}(\bar{G}, m) \qquad \bar{G}^{r_0} <: G^{s_0} \qquad \bar{F}_i^{r_i} <: E_i^{s_i} \qquad H^{s_{n+1}} <: \bar{C}^{\bar{r}} \qquad w_1 \geq n_1 \qquad w_2 - w_1 \leq n_2 - n_1}{\Gamma_0, x{:}\bar{G}^{r_0}, y_1{:}\bar{F}_1^{r_1}, \ldots, y_n{:}\bar{F}_n^{r_n} \vdash^{w_1}_{w_2} x^{r_0}.m\,(y_1^{r_1}, \ldots, y_n^{r_n}) \leftleftarrows \bar{C}^{\bar{r}}} \; (\triangledown Inv.)$$

Case $(\triangledown Cond.)$ We have

$$\frac{x \in \Theta \quad \Theta_1 \vdash^{u}_{u'} e_1^\circ \leftleftarrows C^r \quad \Theta_2 \vdash^{u}_{u'} e_2^\circ \leftleftarrows C^r}{\Theta_1 \wedge \Theta_2 \vdash^{u}_{u'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r}$$

and $C^r <: E^p$ and $|\Gamma| <: |\Theta|$ and $n \geq u$ and $n + u' \geq n' + u$. Then by induction hypothesis and $(\triangledown Cond.)$ we obtain:

$$\frac{x \in \Gamma \quad \Gamma_1 \vdash^{n}_{n'} e_1^\circ \leftleftarrows E^p \quad \Gamma_2 \vdash^{n}_{n'} e_2^\circ \leftleftarrows E^p}{\Gamma_1 \wedge \Gamma_2 \vdash^{n}_{n'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows E^p}$$

Case $(\triangledown Let)$

$$\frac{\Theta_1 \vdash^{u}_{u'} e_1^\circ \leftleftarrows D^s \quad \Theta_2, x{:}D^s \vdash^{u'}_{u''} e_2^\circ \leftleftarrows C^r}{\Theta_1 \oplus \Theta_2 \vdash^{u}_{u''} \mathsf{let}\ D\,x^{u'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r}\ (\triangledown Let)$$

and $C^r <: E^p$ and $|\Gamma| <: |\Theta|$ and

$$n \geq u \tag{4.4.13}$$

$$n'' \leq u'' + n - u \tag{4.4.14}$$

We wish to apply the induction hypothesis on e_1 with $n, u' + n - u$ and on e_2 with $u' + n - u$ and n''. Hence we need to check that the following inequalities holds:

- $n \geq u$ which follows by (4.4.13).
- $u' + n - u \leq u' + n - u$ which follows trivially.
- $u' + n - u \geq u'$ which follows by (4.4.13).
- $n'' \leq u'' + u' + n - u - u'$ which follows by (4.4.14).

Thus, we can apply the induction hypothesis and $(\triangledown Let)$ and obtain:

$$\frac{\Gamma_1 \vdash^{n}_{u'+n-u} e_1^\circ \leftleftarrows E^p \quad \Gamma_2, x{:}D^s \vdash^{u'+n-u}_{n''} e_2^\circ \leftleftarrows E^p}{\Gamma_1 \oplus \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ D\,x^{u'+n-u} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows E^p}\ (\triangledown Let)$$

□

Lemma 4.4.7 (Completeness of algorithmic RAJA typing)
Let $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ be a well-typed RAJA program and $\mathsf{Inst}(C, m)$ be the set of instances of $\mathsf{M}(C, m)$ for each $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$.
If $\Gamma \vdash^{n}_{n'} e : C^r$ then there exists an annotation for e such that $\Delta \vdash^{n}_{n'} e^\circ \leftleftarrows C^r$ and $|\Gamma| = |\Delta|$ and $\Gamma <: \Delta$.

Proof. By induction on typing derivations.

Case
$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ D\,x = e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

Let us assume for ease of notation w.l.o.g. $\Gamma_1 = y{:}C^{r_1}$ and $\Gamma_2 = z{:}E^{r_2}$. Then, by induction hypothesis, we obtain

$$x{:}C^{\hat{r}_1} \vdash^{n}_{n'} e_1^\circ \leftleftarrows D^s \tag{4.4.15}$$

and

$$z{:}E^{\hat{r}_2}, x{:}D^{s'} \vdash^{n'}_{n''} e_2^\circ \leftleftarrows C^r \tag{4.4.16}$$

and $r_i \sqsubseteq \hat{r}_i$ and $D^s <: D^{s'}$. Then, we apply the Waste Lemma (Lemma 4.4.6) to (4.4.15) and obtain

$$x{:}C^{\hat{r}_1}, z{:}E^{0_\mathsf{E}} \vdash^{n}_{n'} e_1^\circ \leftleftarrows D^{s'} \tag{4.4.17}$$

and we apply Lemma 4.4.6 again to (4.4.16) and obtain

$$x{:}C^{0_\mathsf{C}}, z{:}E^{\hat{r}_2}, x{:}D^{s'} \vdash^{n'}_{n''} e_2^\circ \leftleftarrows C^r \tag{4.4.18}$$

Next, by applying rule ($\triangledown Let$) to (4.4.17) and (4.4.18) we obtain

$$x{:}C^{\hat{r}_1 \oplus 0_\mathsf{C}}, z{:}E^{0_\mathsf{E} \oplus \hat{r}_2} \vdash^{n}_{n''} \mathsf{let}\ D\,x^{n'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r$$

which, by Lemma 4.3.1, is equivalent to

$$x{:}C^{\hat{r}_1}, z{:}E^{\hat{r}_2} \vdash^{n}_{n''} \mathsf{let}\ D\,x^{n'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r$$

Case
$$\frac{x \in \Gamma \qquad \Gamma \vdash^{n}_{n'} e_1 : C^r \qquad \Gamma \vdash^{n}_{n'} e_2 : C^r}{\Gamma \vdash^{n}_{n'} \mathtt{if}\ x\ \mathsf{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : C^r}\ (\Diamond Conditional)$$

For ease of notation w.l.o.g. let us assume $\Gamma = x : D^s$. By induction hypothesis we get $x : D^{s_1} \vdash^{n}_{n'} e_1^\circ \leftleftarrows C^r$ and $x : D^{s_2} \vdash^{n}_{n'} e_2^\circ \leftleftarrows C^r$ and $D^s <: D^{s_i}$. We then apply ($\triangledown Cond.$) and obtain

$$x{:}D^{s_1 \wedge s_2} \vdash^{n}_{n'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r$$

Then $D^s <: D^{s_1 \wedge s_2}$ follows by Lemma 3.2.3.

Case $(\Diamond Invocation)$. We have

$$\frac{\mathsf{M}(C,m) = \phi \quad (C^{s_0}; \vec{E^{\vec{s}}} \xrightarrow{n_1/n_2} H^{s_{n+1}}) \text{ instanceof } \phi}{x{:}C^{s_0}, y_1{:}E_1^{s_1}, \ldots, y_n{:}E_n^{s_n} \ \vdash^{n}_{n'} \ x.m(y_1,\ldots,y_n) : H^{s_{n+1}}} \ (\Diamond Invocation)$$

Since $\mathscr{R}^+$ is complete, we know that $(C^{s_0}; \vec{E^{\vec{s}}} \xrightarrow{n_1/n_2} H^{s_{n+1}}) \in \mathsf{Inst}(C,m)$ and can apply $(\triangledown Inv.)$.

$$\frac{(C^{s_0}; \vec{E^{\vec{s}}} \xrightarrow{n_1/n_2} H^{s_{n+1}}) \in \mathsf{Inst}(C,m)}{x{:}C^{s_0}, y_1{:}E_1^{s_1}, \ldots, y_n{:}E_n^{s_n} \ \vdash^{n_1}_{n_2} \ x^{s_0}.m(y_1^{s_1},\ldots,y_n^{s_n}) : H^{s_{n+1}}} \ (\triangledown Inv.)$$

Case

$$\frac{\curlyvee(s\,|s_1,\ldots,s_n) \qquad \Gamma, y_1{:}D^{s_1}, \ldots, y_n{:}D^{s_n} \ \vdash^{n}_{n'} \ e : C^r}{\Gamma, x{:}D^{s} \ \vdash^{n}_{n'} \ e[x/y_1,\ldots,x/y_n] : C^r} \ (\Diamond Share)$$

By induction hypothesis we get $\Delta, y_1 : D^{\hat{s}_1}, \ldots, y_n : D^{\hat{s}_n} \ \vdash^{n}_{n'} \ e^\circ \leftleftarrows C^r$ with $\Gamma <: \Delta$ and $D^{s_i} <: D^{\hat{s}_i}$ for $i \in \{1,\ldots,n\}$. Then, by Lemma 3.2.24 we obtain $\curlyvee(s\,|\hat{s}_1,\ldots,\hat{s}_n)$. Then, by the Share Lemma (Lemma 4.4.5) we obtain $\Delta, x{:}D^{\hat{s}} \ \vdash^{n}_{n'} \ e[x/y_1,\ldots,x/y_n]^\circ \leftleftarrows C^r$ with $s \sqsubseteq \hat{s}$.

Case

$$\frac{\Theta \ \vdash^{m}_{m'} \ e : D^s \quad n \geq m \quad m'-m \geq n'-n \quad \Gamma <: \Theta \quad D^s <: C^r}{\Gamma \ \vdash^{n}_{n'} \ e : C^r} \ (\Diamond Waste)$$

Let us assume for ease of notation w.l.o.g. $\Theta = x{:}E^p$ and $\Gamma = x{:}F^q$. Thus, we have by assumption $F^q <: E^p$. By induction hypothesis we get $x : E^{p'} \ \vdash^{m}_{m'} \ e^\circ \leftleftarrows D^s$ with $E^p <: E^{p'}$. By the Waste Lemma (Lemma 4.4.6) we get an annotation for e such that $x{:}F^{p'} \colon \vdash^{n}_{n'} \ e^\circ \leftleftarrows C^r$. Moreover, $F^q <: F^{p'}$ follows from $q \sqsubseteq p'$ which follows from $F^q <: E^p <: E^{p'}$ with transitivity.

□

Next we show that if $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ is a well-typed RAJA program then we can find $\mathscr{V}$, Inst and $\mathsf{A}_{\mathsf{body}}$ such that $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ is well-typed.

Lemma 4.4.8 (Completeness of algorithmic RAJA method typing) *Let $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ be a well-typed RAJA program. Further let $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$ and let $\mathsf{Inst}(C,m)$ be the set of instances of $\mathsf{M}(C,m)$. If $\vdash m : \mathsf{M}(C,m)$ ok then there exists a map $\mathsf{A}_{\mathsf{body}}(C,m,\cdot)$ such that $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C,m)$ ok.*

Proof. Let $T = C^{s_0}; \vec{E^{\vec{s}}} \xrightarrow{n/n'} H^{s_{n+1}}$ instanceof $\mathsf{M}(C,m)$ and $\curlyvee(s_0\,|q_1,q_2)$ and let $\mathsf{M}_{\mathsf{body}}(C,m) = e$. We have

$$\texttt{this}{:}C^{q_1}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \ \vdash^{n + \Diamond(C^{q_2})}_{n'} \ e : H^{s_{n+1}}$$

By Lemma 4.4.7 we get an annotation for e such that

$$\texttt{this}\!:\!C^{r_0}, x_1\!:\!E_1^{r_1}, \ldots, x_n\!:\!E_n^{r_n} \;\vdash^{\underline{n + \Diamond(C^{q_2})}}_{n'}\; e^\circ \Leftarrow H^{s_{n+1}}$$

and $C^{q_1} <: C^{r_0}$ and $E_i^{s_i} <: E_i^{r_i}$ for all $i = 1, \ldots, n$. Let $p = \Diamond(C^{q_2})$. Then, we set $\mathsf{A}_{\mathsf{body}}(C, m, T) = (e^\circ, p)$. We need to show $(s_0 \dot{-} \mathsf{p})_{\mathsf{C}} \sqsubseteq r_0$. By Lemma 4.3.4 we get $(s_0 \dot{-} \mathsf{p})_{\mathsf{C}} \sqsubseteq q_1$ and we are done by transitivity. □

Theorem 4.4.9 *Let $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ be a well-typed RAJA program. Then there exists a set of views $\mathscr{V}$ and maps Inst and $\mathsf{A}_{\mathsf{body}}$ such that $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ is well-typed.*

Proof. We set $\mathscr{V} = \mathscr{V}^{\mathscr{C}}$ and for each $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$ we set

$$\mathsf{Inst}(C, m) = \{T \in \mathsf{MonoType} \mid T \text{ } \mathsf{instanceof} \text{ } \mathsf{M}(C, m)\}$$

Then, by Lemma 4.4.8, we get $\mathsf{A}_{\mathsf{body}}(C, m, \cdot)$ such that $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)$ ok. Moreover, if $\mathsf{S}(C) = D$, then $\mathsf{Inst}(C, m) <: \mathsf{Inst}(D, m)$ follows from $\mathsf{M}(C, m) <: \mathsf{M}(D, m)$. □

4.5 Decidability of typing

In the previous section we provided syntax-directed rules for type checking RAJA programs with explicit types that are equivalent to the declarative and non-deterministic typing rules from Chapter 3. These rules will be useful for proving the correctness of the constraint generation rules for type inference that we shall describe in Chapter 5.

Moreover, we have described a deterministic type-checking algorithm that can be implemented for checking correctness of RAJA programs with explicit types, provided that the range of the map Inst is finite and that the set of views $\mathscr{V}$ is finite and the views are regular.

Definition 4.5.1 (Finite RAJA programs with explicit types)
Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ be a RAJA program with explicit types. We say that $\mathscr{R}^+$ is finite *if the following conditions are satisfied:*

1. *$\mathscr{V}$ is a finite set of regular views.*

2. *For each $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$ holds $\mathsf{Inst}(C, m)$ is finite.*

4.5.1 Decidability of subtyping

For checking that RAJA programs with explicit types are well-typed, we need to check subtyping between RAJA types and between sets of monomorphic RAJA method types, both of which can be reduced to checking subtyping between views, i.e. checking $s_1 \sqsubseteq s_2$ when s_1 and s_2 are views, which is defined coinductively. Hence we can implement subtyping between views using an algorithm for computing membership in greatest fixpoints. Concretely, we use an algorithm defined and proved correct in [Pie02, Ch. 21], which works for coinductive definitions that fall into a specific scheme, i.e. a goal is *supported* by a set of sub-goals in a deterministic way. In our case, a goal $s_1 \sqsubseteq s_2$ is supported by the set of sub-goals $\mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^{s_2}, a)$ and $\mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^{s_1}, a)$ for each class $C \in \mathscr{C}$ and $a \in \mathsf{A}(C)$.

The idea of the algorithm is to maintain a list of assumptions. Every goal is kept in this list, unless some condition is not fulfilled. The condition in our case is $\Diamond\!(C^{s_1}) \geq \Diamond\!(C^{s_2})$ for each class C. Then, the algorithm is called recursively with all the sub-goals. If a given sub-goal is an element of the list of assumptions (which means it has been a goal before) then we conclude that the sub-goal is in the coinductive defined relation.

As described in [Pie02, Ch. 21], the algorithm terminates if the set of reachable states from a given goal (a pair of views) is finite. Since views are infinite trees, the set of reachable states can be infinite. Hence, we must ensure that we check subtyping only between views for which the set of reachable states is finite and this is the case for regular views.

This is the reason why we require the views that appear in a finite RAJA program to be regular. However, in the syntax-directed typing rules from Section 4.4.1, we use not only views from the given set $\mathscr{V}$, but also computed views like $s_1 \oplus s_2$ or n_C. Fortunately, these computed views are regular as well, as the following fact shows.

Fact 4.5.2 *Let $s, s_1, s_2 \in \mathscr{V}^{\mathscr{C}}$ and $n \in \mathbb{D}$ and $C \in \mathscr{C}$ and let $* \in \{\oplus, \boxplus, \wedge, \vee\}$. Then:*

1. *n_C and n_C^- are regular.*
2. *$(s \dot{-} \mathsf{n})_C$ is regular, if s is regular.*
3. *$s_1 * s_2$ is regular, if s_1 and s_2 are regular.*

4.5.2 Efficiency of typing

In the following we prove that given a finite RAJA program with explicit types $\mathscr{R}^+$, it can be decided in polynomial time whether $\mathscr{R}^+$ is well-typed.

Lemma 4.5.3 (Efficiency of syntax-directed typing)
$\Gamma \vdash^{n}_{n'} e^{\circ} \leftleftarrows C^{r}$ *is decidable in polynomial time.*

Proof sketch. The backwards application of the syntax-directed typing rules produces a linear number of subtyping constraints. Furthermore, the algorithmic view expressions occurring in these constraints are themselves of linear size. It then suffices to restrict attention to the views that occur as sub-expressions of the ones appearing in the constraints. Their number is therefore polynomial in the size of the program. A complete table of the subtyping judgements for this relevant subset can then be computed iteratively in polynomial time. In practice, a goal-directed implementation performs even better. □

Theorem 4.5.4 (Efficiency of typing finite RAJA programs)
Given a finite RAJA program $\mathscr{R}^{+} = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *with explicit types, its well-typedness is decidable in polynomial time.*

Proof. Let $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$. Then $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$ is decidable in polynomial time because

$$\mathtt{this}{:}C^{s_0}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \vdash^{u}_{u'} \mathsf{M}_{\mathsf{body}}(C, m)^{\circ} \leftleftarrows H^{r_{n+1}}$$

is decidable in polynomial time by Lemma 4.5.3. Let moreover $\mathsf{S}(C) = D$. Then $\mathsf{Inst}(C, m) <: \mathsf{Inst}(D, m)$ is decidable in polynomial time, as discussed in Section 4.5.1. □

In [HR09] we provided a similar type checking algorithm for RAJA programs and we also gave an implementation. The main difference between that system and the system presented here is that here we require more annotations and the rules are considerably simpler. The reason for this is that, when we described the system in [HR09], we had not developed the type inference algorithm yet, and thus we wanted to reduce the amount of annotations that a programmer had to write to a minimum. Here we have more annotations because we also have a type inference algorithm that will provide them, that we shall describe in the next chapter.

Chapter 5

Type Inference for RAJA

5.1 Overview

In this chapter we present an algorithm that, when given an FJEU program $\mathcal{P} = (\mathcal{C}, \mathsf{main})$, generates a RAJA program based on $\mathcal{P}$ by giving polymorphic RAJA method types to the methods of $\mathcal{P}$. Moreover, we give an algorithm for generating a monomorphic RAJA method type for the main method of $\mathcal{P}$, by solving the constraints of main's polymorphic type. The heap-space consumption of $\mathcal{P}$ follows from the potential given to main's arguments by that type, as discussed in Chapter 3.

We are only able to analyse a subset of the programs that are typeable in the RAJA system. We have the following restrictions:

First, we do not analyse programs with polymorphic recursion; we analyse programs that are typeable in a subset of RAJA that allows only monomorphic recursion, called RAJA^{m}.

Second, the algorithm for solving subtyping constraints that we describe in Chapter 6, is capable of solving only a subset of the constraints that arise during the analysis, i.e. a subset of the constraints that correspond to programs whose heap-space consumption can be described as a linear function. Finding an algorithm for solving all the subtyping constraints remains an open problem for future work.

When we are able to obtain a well-typed RAJA program for $\mathcal{P}$, we can also extend it to a finite RAJA program with explicit types, that can be type checked with the efficient algorithm described in Chapter 4.

This chapter is organised as follows. In Section 5.2 we describe the system RAJA^{m}. Then, in Section 5.3 we present an algorithm for generating RAJA^{m} programs. Section 5.4 discusses the algorithm for constraint solving, which is described in Chapter 6. Finally, Section 5.5 describes how we can build a finite RAJA program with explicit types.

Notation. In this chapter we use the notation $C.m$ for referring to the method m of class C, when we wish to distinguish it from the method of the same name in another class.

5.2 RAJA programs with monomorphic recursion

In type systems with polymorphic types and recursion, polymorphic recursion is possible. Polymorphic recursion means that, in recursive calls, any instance of the polymorphic type can be used, whereas in monomorphic recursion only one instance can be used: the same instance that the polymorphic type is being typechecked with.

In this section we shall define a subset of the RAJA system, called RAJA$^{\mathrm{m}}$, where only monomorphic recursion is allowed. The reason for not treating polymorphic recursion is that type inference in the presence of polymorphic recursion is difficult, in particular we would need to compute a fixpoint when generating constraints for recursive functions. We decided to develop a simpler type inference algorithm, that does not require a fixpoint computation, because RAJA$^{\mathrm{m}}$ is expressive enough for coding interesting programs, as we shall see in our experimental evaluation in Chapter 7.

RAJA$^{\mathrm{m}}$ programs are RAJA programs with explicit types with slightly modified typing rules. In Fig. 5.1 we define typing rules that are sound with respect to the typing rules for RAJA programs with explicit types from Chapter 4, but that forbid programs that use polymorphic recursion.

We implement monomorphic recursion with the help of a map

$$\Xi : \forall C \in \mathscr{C} \,.\, \mathsf{Meth}(C) \to \mathsf{MonoType}$$

For type checking a method body, we check that it is typeable with each instance T of its polymorphic RAJA method type ϕ. The map Ξ keeps track of those instances during type checking, i.e. we set $\Xi(C, m) = T$. Then, in the method invocation rule, we check that the methods that appear in $\mathrm{dom}(\Xi)$ can only be called with the same instance T.

Another important aspect of the implementation of monomorphic recursion is the order in which methods are type checked. We need to distinguish between recursive and non-recursive method calls. With non-recursive methods calls, we can use any instance of the polymorphic type of the called method. That is why there are two rules for method invocation: ($\nabla PolyInv.$) for polymorphic method invocation and ($\nabla MonInv.$) for monomorphic method invocation. In the rule ($\nabla PolyInv.$) we assume that the called method is not mutually recursive with the method we are currently analysing, and consequently, we can use any instance of its polymorphic type. On the other hand, we apply the rule ($\nabla MonInv.$) when the called method appears in the map Ξ, which means that this method and the method whose body we are analysing are mutually recursive.

The judgement for typing the body of a method ($\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$) shall mean that all the methods in the domain of the map M are well-typed.

RAJA^m Typing $\boxed{\mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} e \leftleftarrows C^r}$

$$\frac{\begin{array}{c}\forall a \in \mathsf{A}(D)\,.\,\mathsf{A}^{\mathrm{set}}(D^r,a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^r,a) \\ D^r <: C^r \qquad n \geq \Diamond(D^r)+1 \qquad n' \leq n - \Diamond(D^r) - 1\end{array}}{\mathsf{M};\Xi;\Gamma_0 \vdash^{n}_{n'} \mathsf{new}\ D \leftleftarrows C^r}\ (\nabla New)$$

$$\frac{n' \leq n + \min\{\Diamond(D^r) \mid D <: C\} + 1}{\mathsf{M};\Xi;\Gamma_0, x{:}C^r \vdash^{n}_{n'} \mathsf{free}\,(x^r) \leftleftarrows E^s}\ (\nabla Free) \qquad \frac{D <: E \quad D^r <: C^s \quad n' \leq n}{\mathsf{M};\Xi;\Gamma_0, x{:}E^r \vdash^{n}_{n'} (D)x^r \leftleftarrows C^s}\ (\nabla Cast)$$

$$\frac{n' \leq n}{\mathsf{M};\Xi;\Gamma_0 \vdash^{n}_{n'} \mathsf{null} \leftleftarrows C^s}\ (\nabla Null) \qquad \frac{E^r <: C^s \qquad n' \leq n}{\mathsf{M};\Xi;\Gamma_0, x{:}E^r \vdash^{n}_{n'} x^r \leftleftarrows C^s}\ (\nabla Var)$$

$$\frac{\forall F <: C\,.\,\mathsf{A}^{\mathrm{get}}(F^r,a) \sqsubseteq s \qquad C.a = E \qquad E <: D \qquad n' \leq n}{\mathsf{M};\Xi;\Gamma_0, x{:}C^r \vdash^{n}_{n'} x^r.a \leftleftarrows D^s}\ (\nabla Access)$$

$$\frac{\forall G <: E\,.\,s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r,a) \qquad E.a = D \qquad F <: D \qquad E^r <: C^q \qquad n' \leq n}{\mathsf{M};\Xi;\Gamma_0, x{:}E^r, y{:}F^s \vdash^{n}_{n'} x^r.a \leftarrow y^s \leftleftarrows C^q}\ (\nabla Update)$$

$$\frac{\mathsf{M};\Xi;\vec{y}{:}\vec{F}^{\vec{p}} \vdash^{n}_{n'} e_1^\circ \leftleftarrows D^s \quad \mathsf{M};\Xi;\vec{y}{:}\vec{F}^{\vec{q}}, x{:}D^s \vdash^{n'}_{n''} e_2^\circ \leftleftarrows C^r \quad r_i \sqsubseteq p_i \oplus q_i}{\mathsf{M};\Xi;\vec{y}{:}\vec{F}^{\vec{r}} \vdash^{n}_{n''} \mathsf{let}\ D\,x^{n'} = e_1^\circ\ \mathsf{in}\ e_2^\circ \leftleftarrows C^r}\ (\nabla Let)$$

$$\frac{x \in \Gamma \quad \mathsf{M};\Xi;\Gamma_1 \vdash^{n}_{n'} e_1^\circ \leftleftarrows C^r \quad \mathsf{M};\Xi;\Gamma_2 \vdash^{n}_{n'} e_2^\circ \leftleftarrows C^r}{\mathsf{M};\Xi;\Gamma_1 \wedge \Gamma_2 \vdash^{n}_{n'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r}\ (\nabla Conditional)$$

$$\frac{\begin{array}{c}\left(G^{s_0}; \vec{E}^{\vec{s}} \xrightarrow{m/m'} H^{s'}\right)\ \mathsf{instanceof}\ \mathsf{M}(G\,,m) \qquad G^{r_0} <: G^{s_0} \\ F_i^{r_i} <: E_i^{s_i} \qquad H^{s'} <: C^{r'} \qquad n \geq m \qquad n' \leq m' + n - m\end{array}}{\mathsf{M};\Xi;\Gamma_0, x{:}G^{r_0}, y_1{:}F_1^{r_1}, \ldots, y_n{:}F_n^{r_n} \vdash^{n}_{n'} x^{r_0}.m\,(y_1^{r_1}, \ldots, y_n^{r_n}) \leftleftarrows C^{r'}}\ (\nabla PolyInv.)$$

$$\frac{\begin{array}{c}\left(G^{s_0}; \vec{E}^{\vec{s}\ m/m'} H^{s'}\right) \in \Xi(G\,,m) \qquad G^{r_0} <: G^{s_0} \\ F_i^{r_i} <: E_i^{s_i} \qquad H^{s'} <: C^{r'} \qquad n \geq m \qquad n' \leq m' + n - m\end{array}}{\mathsf{M};\Xi;\Gamma_0, x{:}G^{r_0}, y_1{:}F_1^{r_1}, \ldots, y_n{:}F_n^{r_n} \vdash^{n}_{n'} x^{r_0}.m\,(y_1^{r_1}, \ldots, y_n^{r_n}) \leftleftarrows C^{r'}}\ (\nabla MonInv.)$$

RAJA^m Method Typing $\boxed{\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\vdash_{\mathsf{m}} \mathsf{M}'\ \mathsf{ok} \qquad \forall\,(C,m) \in \mathsf{M}'' \qquad \forall\,T = (C^{r_0}; \vec{E}^{\vec{\mathsf{r}}\ n/n'} H^{r_{n+1}})\ \mathsf{instanceof}\ \mathsf{M}''(C,m) \\ \mathrm{dom}(\Xi) = \mathrm{dom}(\mathsf{M}'') \quad \Xi(C,m) = T \qquad (r_0 \dot{-} \mathsf{p})_{\mathsf{C}} \sqsubseteq s_0 \quad r_i \sqsubseteq s_i \qquad \Diamond(C^{r_0}) \geq p \\ \mathsf{A}_{\mathsf{body}}(C,m,T) = (e^\circ, p) \quad \mathsf{M}';\Xi;\mathtt{this}{:}C^{s_0}, x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \vdash^{n+p}_{n'} e^\circ \leftleftarrows H^{r_{n+1}}\end{array}}{\vdash_{\mathsf{m}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}}$$

Figure 5.1: RAJA$^{\mathrm{m}}$ Typing.

When analysing M, we partition its domain according to the call graph of the program: If the methods $m_1, \ldots, m_j$ in the classes $C_1, \ldots, C_j$ are mutually recursive and $(C_i, m_i) \in \mathrm{dom}(\mathsf{M})$ for each $i \in \{1, \ldots, j\}$, and all the methods called by the methods m_1 to m_j are in

$$\mathrm{dom}(\mathsf{M}') = \mathrm{dom}(\mathsf{M}) \setminus \bigcup_{i \in \{1,\ldots,j\}} (C_i, m_i)$$

then we proceed as follows. First, we check $\vdash_{\mathsf{m}} \mathsf{M}' \ \mathsf{ok}$. Then, for checking m_1 to m_j, we set $\mathrm{dom}(\Xi) = \bigcup_{i \in \{1,\ldots,j\}} (C_i, m_i)$, we check each instance T_i of the polymorphic type of m_i, and at the same time we set $\Xi(C_i, m_i) = T_i$. Notice that we have described the general case, when we are in the presence of j mutually recursive methods, but clearly j can be equal 1 in the case of standard recursion or no recursion at all. We also remark that we call the class where m_i appears C_i for ease of notation. It is by all means possible that m_i and m_j are in the same class; in that case $C_i = C_j$.

Definition 5.2.1 (Well-typed RAJA$^{\mathbf{m}}$-program)
A RAJA^m-program $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *is* well-typed *if the following conditions are satisfied:*

1. $\vdash_{\mathsf{m}} \mathsf{M} \ \mathsf{ok}$

2. $\forall C, D \in \mathscr{C}$ *with* $S(C) = D \Rightarrow \mathsf{M}(C, m) <: \mathsf{M}(D, m)$.

Lemma 5.2.2 (Soundness of RAJA$^{\mathbf{m}}$ typing)
Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *be a complete RAJA^m-program and let* $\mathsf{M} = \mathsf{M}' \uplus \mathsf{M}''$. *If* $\Xi(C, m)$ $\mathsf{instanceof}$ $\mathsf{M}''(C, m)$ *for each* $(C, m) \in \mathrm{dom}(\mathsf{M}'')$ *and* $\mathcal{D} :: \mathsf{M}'; \Xi; \Gamma \stackrel{n}{\vdash}_{n'} e \leftrightarrows C^r$ *then* $\Delta \stackrel{n}{\vdash}_{n'} e \leftrightarrows C^r$ *and* $\Gamma <: \Delta$.

Proof. By induction on $\mathcal{D}$.

Case ($\nabla MonInv$.). We can finish with ($\triangledown Inv$.) because of the assumption $\Xi(C, m)$ $\mathsf{instanceof}$ $\mathsf{M}''(C, m)$ for each $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$.

Case (∇Let). By the induction hypothesis and Lemma 3.2.8.

Case ($\nabla Conditional$). By the induction hypothesis and Lemma 3.2.6.

□

Lemma 5.2.3 (Soundness of RAJA$^{\mathrm{m}}$ method typing)
Let $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ *be a complete RAJA^m-program. If* $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$ *then for all* $(C, m) \in \mathrm{dom}(\mathsf{M})$ *holds* $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$.

Proof. By induction on $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$. Let $\mathsf{M} = \mathsf{M}' \uplus \mathsf{M}''$ and $\vdash_{\mathsf{m}} \mathsf{M}'\ \mathsf{ok}$. Then, by induction hypothesis, for all $(D, m') \in \mathrm{dom}(\mathsf{M}')$ holds $\vdash_{\mathsf{a}} m : \mathsf{Inst}(D, m')\ \mathsf{ok}$. Moreover let:

$$(C, m) \in \mathrm{dom}(\mathsf{M}'') \tag{5.2.1}$$

$$T = (C^{r_0}; \vec{E}^{\vec{\mathfrak{r}}\, n_1/n_2} \!\!\longrightarrow H^{r_{n+1}})\ \mathsf{instanceof}\ \mathsf{M}''(C, m) \tag{5.2.2}$$

$$\Xi(C, m) = T \tag{5.2.3}$$

$$(r_0 \dot{-} \mathsf{p})_{\mathsf{C}} \sqsubseteq s_0 \tag{5.2.4}$$

$$r_i \sqsubseteq s_i \tag{5.2.5}$$

$$\Diamond\!(C^{r_0}) \geq p \tag{5.2.6}$$

$$\mathsf{A}_{\mathsf{body}}(C, m, T) = (e^\circ, p) \tag{5.2.7}$$

$$\mathsf{M}'; \Xi; \mathtt{this}{:}C^{s_0},\ x_1{:}E_1^{s_1}, \ldots, x_n{:}E_n^{s_n} \ \vdash^{n+p}_{n'}\ e^\circ \leftleftarrows H^{r_{n+1}} \tag{5.2.8}$$

since $\Xi(C, m)\ \mathsf{instanceof}\ \mathsf{M}''(C, m)$ for each $(C, m) \in \mathrm{dom}(\mathsf{M}'')$, we can apply Lemma 5.2.2 and obtain

$$\mathtt{this}{:}C^{q_0},\ x_1{:}E_1^{q_1}, \ldots, x_n{:}E_n^{q_n} \ \vdash^{n+p}_{n'}\ e^\circ \leftleftarrows H^{r_{n+1}}$$

with $s_i \sqsubseteq q_i$. Since $r_i \sqsubseteq q_i$ by (5.2.5) and transitivity, we obtain the desired $\vdash_{\mathsf{a}} m : \mathsf{Inst}(C, m)\ \mathsf{ok}$. □

5.3 Generation of RAJA$^{\mathrm{m}}$ programs

Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be an FJEU program. In this section we present an algorithm that generates subtyping and arithmetic constraints for the methods in $\mathcal{P}$. This gives us polymorphic RAJA$^{\mathrm{m}}$ method types that represent the heap-space consumption of the methods.

It is important to remark that the generation of a polymorphic RAJA$^{\mathrm{m}}$ method type for a method can be performed modularly because the type does not depend on the method's callers. This implies that when programmers add new classes and methods to the program, the polymorphic type for the method is still valid, except for the case when the method is redefined in a newly added subclass. The reason for this will be explained in detail later.

5.3.1 Constraint generation rules

In the following we present rules for generating subtyping and arithmetic constraints from FJEU programs that are sound and complete with respect

to the typing rules from the system RAJA$^{\mathrm{m}}$. The rules (Fig. 5.2) describe a constraint generation judgement

$$\mathsf{M}; \Xi; \Gamma \vdash^{p}_{p'} e \leftleftarrows C^{v} \;\&\; \mathcal{C}$$

where e is an expression, Γ maps variables to FJEU types refined with view variables, C^v is an FJEU type refined with a view variable, p and p' are arithmetic variables and $\mathcal{C}$ is a conjunction of subtyping and arithmetic constraints. Further, Ξ is a map from classes and methods with n arguments to $n+2$ view variables and two arithmetic variables.

The judgement reads: expression e has type C^v in the context Γ, subject to the constraints $\mathcal{C}$. Moreover, the judgement defines a *total* function $\mathsf{generateConstraints}$ that generates constraints for an expression:

$$\mathsf{generateConstraints}(\mathsf{M}, \Xi, \Gamma, p, p', e, C^v) \;=\; \mathcal{C} \text{ if } \mathsf{M}; \Xi; \Gamma \vdash^{p}_{p'} e : C^{v} \;\&\; \mathcal{C}$$

Our notation and methodology has been partially inspired by Knowles and Flanagan's type reconstruction algorithm for refinement types [KF07].

We sometimes write $\vec{y} : \vec{F^{v}}$ to mean the context $y_1 : F_1^{v_1}, \ldots, y_n : F_n^{v_n}$. The subtyping constraints are of the form $C^v <: D^u$ where C and D are classes and v and u are view variables. We also create constraints of the form $u \sqsubseteq v \oplus w$ in the rule $(\triangle Let)$, where $v \oplus w$ is a view expression.

There are two rules for method invocation: $(\triangle PolyInv)$ for polymorphic method invocation and $(\triangle MonInv)$ for monomorphic method invocation.

In the rule $(\triangle PolyInv)$ we assume that the called method has already been analysed and so its polymorphic RAJA method type is available. The constraints generated by this rule consist of the method's constraints, where we substitute the view and arithmetic variables with fresh ones, in conjunction with subtyping and arithmetic constraints needed for the integration of the $(\Diamond Waste)$ rule.

We apply the rule $(\nabla MonInv.)$ when the called method appears in the map Ξ, which means, as we discussed earlier, that the method and the method whose body we are analysing are mutually recursive. In that case the constraints for the method are not yet available. Thus, we only generate the standard subtyping and arithmetic constraints.

The judgement $\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$ returns RAJA$^{\mathrm{m}}$ polymorphic method types for the methods in M by generating the constraints for the methods' bodies. We perform the analysis on the basis of the call graph of the program, which we modify slightly by adding the inheritance relations to it. Concretely, let $(C_1, m_{1,1}), \ldots, (C_1, m_{1,j_1}), \ldots, (C_n, m_{n,1}), \ldots, (C_n, m_{n,j_n})$ be an enumeration of all the pairs of classes and methods in a program $\mathcal{P}$, such that $C_i \in \mathscr{C}$ and $m_{i,k} \in \mathsf{Meth}(C_i)$.

$\mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} e : C^v \;\&\; \mathcal{C}$

$$\frac{\mathcal{C} = (E^v <: C^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x{:}E^v \vdash^{p}_{p'} x : C^u \;\&\; \mathcal{C}}\;(\triangle Var) \qquad \frac{\mathcal{C} = (p' \leq p)}{\mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} \mathsf{null} : C^v \;\&\; \mathcal{C}}\;(\triangle Null)$$

$$\frac{\mathcal{C} = (D^v <: E^v \;\wedge\; D^v <: C^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x{:}E^v \vdash^{p}_{p'} (D)\, x : C^u \;\&\; \mathcal{C}}\;(\triangle Cast)$$

$$\mathcal{E} = \textstyle\bigwedge_{a \in \mathsf{A}(D)} \mathsf{A}^{\mathrm{set}}(D^v, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^v, a)$$
$$\frac{\mathcal{D} = D^v <: C^v \;\wedge\; p \geq \Diamond(D^v) + 1 \;\wedge\; p' \leq p - \Diamond(D^v) - 1 \qquad \mathcal{C} = \mathcal{D} \wedge \mathcal{E}}{\mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} \mathsf{new}\ D : C^v \;\&\; \mathcal{C}}\;(\triangle New)$$

$$\frac{\mathcal{C} = \textstyle\bigwedge_{D <: C} p' \leq p + \Diamond(D^v) + 1}{\mathsf{M};\Xi;\Gamma, x{:}C^v \vdash^{p}_{p'} \mathsf{free}\,(x) : E^u \;\&\; \mathcal{C}}\;(\triangle Free)$$

$$\frac{\mathcal{C} = \textstyle\bigwedge_{E <: C} (C.a)^{\mathsf{A}^{\mathrm{get}}(E^v, a)} <: D^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x{:}C^v \vdash^{p}_{p'} x.a : D^u \;\&\; \mathcal{C}}\;(\triangle Access)$$

$$\frac{\mathcal{C} = (\textstyle\bigwedge_{E <: C} F^w <: (C.a)^{\mathsf{A}^{\mathrm{set}}(E^v, a)} \;\wedge\; C^v <: D^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x{:}C^v, y{:}F^w \vdash^{p}_{p'} x.a \leftarrow y : D^u \;\&\; \mathcal{C}}\;(\triangle Update)$$

$$\frac{\mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} e_1 : C^v \;\&\; \mathcal{C}_1 \qquad \mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} e_2 : C^v \;\&\; \mathcal{C}_2 \qquad \mathcal{C} = (\mathcal{C}_1 \;\wedge\; \mathcal{C}_2)}{\mathsf{M};\Xi;\Gamma \vdash^{p}_{p'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : C^v \;\&\; \mathcal{C}}\;(\triangle Cond.)$$

$$\mathcal{C} = (\mathcal{C}_1 \;\wedge\; \mathcal{C}_2 \;\wedge\; \textstyle\bigwedge_i u_i \sqsubseteq v_i \oplus w_i)$$
$$\frac{\mathsf{M};\Xi;\vec{y} : \vec{F}^{\vec{v}} \vdash^{p}_{p'} e_1 : D^u \;\&\; \mathcal{C}_1 \qquad \mathsf{M};\Xi;\vec{y} : \vec{F}^{\vec{w}}, x{:}D^u \vdash^{p'}_{p''} e_2 : C^v \;\&\; \mathcal{C}_2}{\mathsf{M};\Xi;\vec{y} : \vec{F}^{\vec{u}} \vdash^{p}_{p''} \mathsf{let}\ D\, x = e_1\ \mathsf{in}\ e_2 : C^v \;\&\; \mathcal{C}}\;(\triangle Let)$$

$$\Xi(G, m) = G^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{q_1/q_2} H^{v_{n+1}} \qquad \mathcal{AC} = p \geq q_1 \;\wedge\; p' \leq q_2 + p - q_1$$
$$\frac{\mathcal{SC} = G^u <: G^{v_0} \;\wedge\; F_i^{u_i} <: E_i^{v_i} \;\wedge\; H^{v_{n+1}} <: C^{u'} \qquad \mathcal{C} = \mathcal{SC} \;\wedge\; \mathcal{AC}}{\mathsf{M};\Xi;\Gamma, x{:}G^u, \vec{y}{:}\vec{F}^{\vec{u}} \vdash^{p}_{p'} x.m\,(\vec{y}) : C^{u'} \;\&\; \mathcal{C}}\;(\triangle MonInv)$$

$$\mathsf{M}(G, m) = \forall \vec{v}, \vec{q} \exists \vec{v'}, \vec{q'}\ .\ G^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{q_1/q_2} H^{v_{n+1}} \;\&\; \mathcal{D} \qquad \mathcal{D}' = \mathcal{D}[\vec{w}/\vec{v}, \vec{w'}/\vec{v'}, \vec{t}/\vec{q}, \vec{t'}/\vec{q'}]$$
$$\mathcal{SC} = G^u <: G^{w_0} \;\wedge\; F_i^{u_i} <: E_i^{w_i} \;\wedge\; H^{w_{n+1}} <: C^{u'}$$
$$\frac{\mathcal{AC} = p \geq t_1 \;\wedge\; p' \leq t_2 + p - t_1 \qquad \mathcal{C} = \mathcal{SC} \;\wedge\; \mathcal{AC} \;\wedge\; \mathcal{D}'}{\mathsf{M};\Xi;\Gamma, x{:}G^u, \vec{y}{:}\vec{F}^{\vec{u}} \vdash^{p}_{p'} x.m\,(\vec{y}) : C^{u'} \;\&\; \mathcal{C}}\;(\triangle PolyInv)$$

Figure 5.2: Constraint generation rules.

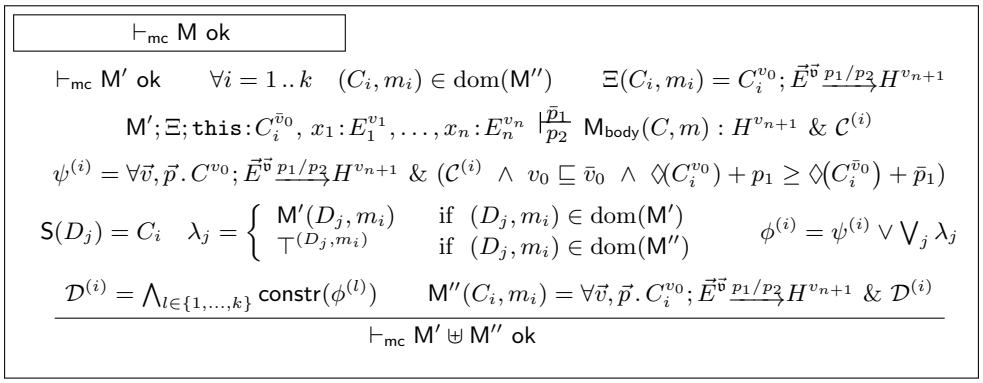


Figure 5.3: Generation of RAJA$^{\mathrm{m}}$ polymorphic types.

Then, we build a directed graph $G = (V, E)$ as follows. We set

$$V = \bigcup_{i \in \{1,\ldots,n\}, k \in \{1,\ldots,j_i\}} (C_i, m_{i,k})$$

and we build the set E of edges by:

$$\begin{array}{lll} ((C_i, m_{i,k}), (C_j, m_{j,t})) \in E & \iff & m_{i,k} \ \text{ calls } \ m_{j,t} \\ ((C_i, m_{i,k}), (C_j, m_{j,t})) \in E & \iff & m_{i,k} = m_{j,t} \ \text{ and } \ \mathsf{S}(C_j) = C_i \end{array}$$

For example, the graph corresponding to the program for copying lists defined in Fig. 3.1, can be represented as in Fig. 5.4.

After we have built the graph, we decompose it in its strongly connected components to obtain the acyclic component graph G^{SCC}. Afterwards, we sort the obtained dag G^{SCC} topologically and call the constraint generation algorithm in that order, with the strongly connected components being analysed together. The decomposition in strongly connected components and the topological sorting of graphs are based on the algorithm for depth-first search of graphs, and are discussed, for instance, in Cormen's "Introduction to Algorithms" [CLRS01].

If we apply these algorithms to our example graph from Fig. 5.4, we obtain the following order:

$$(\,\mathsf{Nil}, \mathsf{copy}), [(\,\mathsf{Cons}, \mathsf{copy}), (\mathsf{List}, \mathsf{copy})], (\,\mathsf{Main}, \mathsf{main})$$

where $(\,\mathsf{Cons}, \mathsf{copy})$ and $(\mathsf{List}, \mathsf{copy})$ are analysed together.

Now, why do we need to extend the call graph with inheritance relations? The reason for this is that, before we analyse a method m in a class C, we would like to analyse the same method m in each subclass D of C. For proving soundness of the constraint generation algorithm we need to show $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$, and this follows trivially when we add the constraints of $D.m$ to the polymorphic type of $C.m$. For example, the method

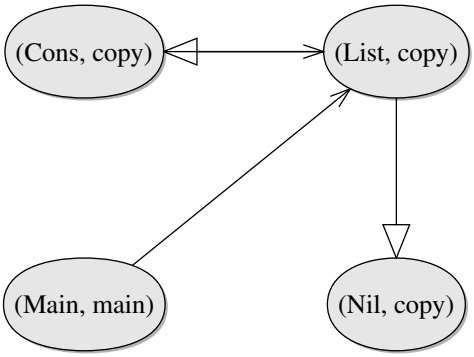


Figure 5.4: Call graph for the program for copying lists extended with inheritance relations.

List.copy should contain the constraints generated by the body of the methods Cons.copy and Nil.copy. Otherwise, a variable of type list could be used for calling the method copy, and this call would always be possible since there are no constraints for the method body of List.copy. However, during runtime the variable could point to a Cons object in the heap, causing the method Cons.copy to be executed, and this would lead to unpredictable resource consumption.

Let us analyse in detail the following lines from the rule for constraint generation for methods:

$$\mathsf{S}(D_j) = C_i \quad \lambda_j = \begin{cases} \mathsf{M}'(D_j, m_i) & \text{if } (D_j, m_i) \in \mathrm{dom}(\mathsf{M}') \\ \top^{(D_j, m_i)} & \text{if } (D_j, m_i) \in \mathrm{dom}(\mathsf{M}'') \end{cases} \tag{5.3.1}$$

$$\phi^{(i)} = \psi^{(i)} \vee \bigvee_j \lambda_j \tag{5.3.2}$$

We are interested in adding the constraints of $D_j.m_i$ to $\mathsf{M}''(C_i, m_i)$, because D_j is a subclass of C_i. Because of the order in which we call the algorithm, there are two possible cases: either the method $D_j.m_i$ has already been analysed at this moment, or the method $D_j.m_i$ is being analysed together with the method $C_i.m_i$. In the first case we set $\lambda_j = \mathsf{M}'(D_j, m_i)$ and in the second case we set $\lambda_j = \top^{(D_j, m_i)}$ because the polymorphic type of $D_j.m_i$ is not available yet. In either case, we create the least upper bound of $\psi^{(i)}$ and λ_1 to λ_n, if C_i has n subclasses. $\psi^{(i)}$ is the polymorphic method type for $C_i.m_i$ that contains the constraints that arise by analysing the method's body. The operation $\psi^{(i)} \vee \bigvee_j \lambda_j$ combines the constraints of the types in such a way that $\psi^{(i)} \vee \bigvee_j \lambda_j$ is the least upper bound of $\psi^{(i)}$ and λ_j, as we proved in Lemma 3.2.19.

If we are analysing together the methods $C_1.m_1, \ldots, C_k.m_k$, we create the types $\phi^{(1)}, \ldots, \phi^{(k)}$ as described so far. Then, in a last step, we add

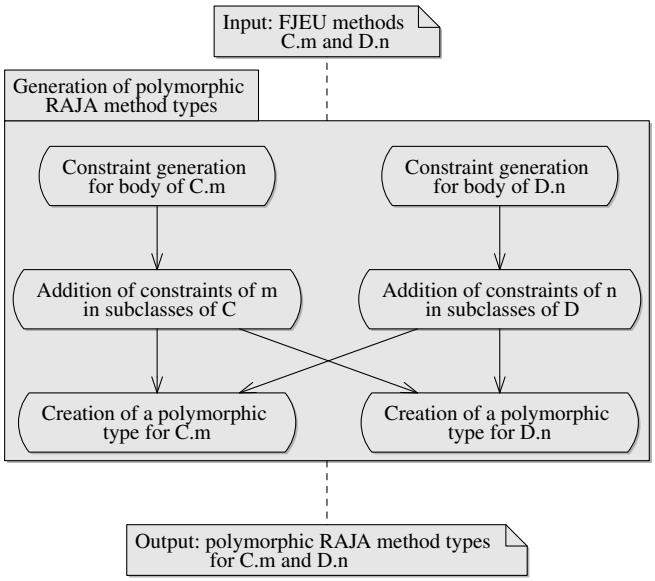


Figure 5.5: Schematic structure of the algorithm for generating polymorphic RAJA method types in the simplified case of analysing the two mutually recursive methods $C.m$ and $D.n$.

to each polymorphic method type $\phi^{(i)}$ the constraints of the other methods $\phi^{(1)}, \ldots, \phi^{(i-1)}, \phi^{(i+1)}, \ldots, \phi^{(k)}$:

$$\mathcal{D}^{(i)} = \bigwedge_{l \in \{1,\ldots,k\}} \mathsf{constr}(\phi^{(l)}) \tag{5.3.3}$$

We can think of this step as adding the constraints retrospectively, which we needed to add in the rule ($\nabla MonInv.$), or when adding the constraints of the method m_i in the subclasses of C_i. Fig. 5.5 shows a schematic structure of the judgement $\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$ in the simplified case where $\mathrm{dom}(\mathsf{M}) = \{(C, m), (D, n)\}$.

5.3.2 Verification of correctness of constraint generation

In this section we shall prove soundness and completeness of the constraint generation rules with respect to the typing rules for $\mathrm{RAJA}^{\mathrm{m}}$ programs.

Notation. If Ξ is a map from classes and methods with n arguments to $n+2$ view variables and 2 arithmetic variables, we write $\pi(\Xi)$ to mean the map from classes and methods to monomorphic RAJA method types that is obtained after substituting every view and arithmetic variable in Ξ with its value in the valuation π. Similarly, $\pi(\Gamma)$ means the context that we obtain after substituting the view variables in Γ with their values in π.

In addition, we use the notations $|\Xi|$ and $|\Gamma|$ for meaning the following. If Ξ is a map from classes and method names to monomorphic RAJA types, then $|\Xi|$ denotes a map from classes and method names to view and arithmetic variables with $\mathrm{dom}(|\Xi|) = \mathrm{dom}(\Xi)$. Similarly, if Γ is an FJEU context, then $|\Gamma|$ is a context from program variables to FJEU types refined with view variables with the same domain as Γ.

Soundness proof

In the following we prove that, if the constraints generated for the expression e are satisfiable, then the expression is typeable in the RAJA$^{\mathrm{m}}$ system with the result type, context and effect given by the solution to the constraints.

Lemma 5.3.1 (Soundness of constraint generation)
If $\mathcal{D} :: \mathsf{M}; \Xi; \Gamma \vdash^{q_1}_{q_2} e : C^v \;\&\; \mathcal{C}$ *and* $\pi \models \mathcal{C}$ *then there exists an annotation for* e *with* $\mathsf{M}; \pi(\Xi); \pi(\Gamma) \vdash^{\pi(q_1)}_{\pi(q_2)} e^\circ : C^{\pi(v)}$.

Proof. By induction on $\mathcal{D}$.

Case ($\triangle Free$) We have $\mathcal{C} = \bigwedge_{D<:C} q_2 \leq q_1 + \Diamond(D^v) + 1$. Let $\pi(q_i) = n_i$ and $\pi(v) = r$. Then, we have for each $D <: C$, $n_2 \leq n_1 + \Diamond(D^r) + 1$, thus, also $n_2 \leq n_1 + \min\{\Diamond(D^r) \mid D <: C\} + 1$. Moreover, since the view variables in Γ are not used in the constraints $\mathcal{C}$, we can assume w.l.o.g. $\pi(u_i) = \widehat{0}_{C_i}$ if $\Gamma = \vec{y} : \vec{C}^{\vec{u}}$. Further, we get the annotation $\mathtt{free}(x^r)$ for e. Thus, we can finish with ($\nabla Free$).

Case ($\triangle Cond.$). For ease of notation let us assume $\Gamma = y : D^u$. We have

$$\mathsf{M}; \Xi; y : D^u \vdash^{q_1}_{q_2} \mathsf{if}\ x\ \mathsf{instanceof}\ D\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : C^v \;\&\; \mathcal{C}_1 \wedge \mathcal{C}_2$$

and $\pi \models \mathcal{C}_1 \wedge \mathcal{C}_2$. Then, also $\pi \models \mathcal{C}_1$ and $\pi \models \mathcal{C}_2$. Let $\pi(v) = r$ and $\pi(u) = s$ and $\pi(p_i) = n_i$. By induction hypothesis and ($\nabla Conditional$) we get

$$\frac{x \in \Gamma \quad \mathsf{M}; \Xi; x : D^s \vdash^{n_1}_{n_2} e_1^\circ \leftleftarrows C^r \quad \mathsf{M}; \Xi; x : D^s \vdash^{n_1}_{n_2} e_2^\circ \leftleftarrows C^r}{\mathsf{M}; \Xi; x : D^{s \wedge s} \vdash^{n_1}_{n_2} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^\circ\ \mathsf{else}\ e_2^\circ \leftleftarrows C^r}$$

and the goal follows since $s = s \wedge s$.

□

Lemma 5.3.2 *Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be an FJEU program and let* $\mathrm{dom}(\mathsf{M}) = \{(C, m) \mid C \in \mathscr{C}, m \in \mathsf{Meth}(C)\}$ *and let* $\mathcal{D} :: \vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$ *and let* $\mathsf{M}(C, m)$ *be non-empty for each* $(C, m) \in \mathrm{dom}(\mathsf{M})$*. Then:*

1. *There exists* $\mathsf{A_{body}}$ *with* $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$*.*
2. $\mathsf{S}(D) = C$ *implies* $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$*.*

Proof.

1. By induction on $\mathcal{D}$. We have $\vdash_{\mathsf{mc}} \mathsf{M}'\ \mathsf{ok}$ and

$$\frac{\begin{array}{c}\Xi(C_i, m_i) = C_i^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}} \\ \mathsf{M}'; \Xi; \mathtt{this}{:}C_i^{\bar{v}_0},\ \vec{x}{:}\vec{E}^{\vec{\mathfrak{v}}} \vdash_{p_2}^{\bar{p}_1} \mathsf{M_{body}}(C, m) \Leftarrow H^{v_{n+1}}\ \&\ \mathcal{C}^{(i)} \\ \mathcal{D}^{(i)} = (\mathcal{C}^{(i)}\ \wedge\ v_0 \sqsubseteq \bar{v}_0\ \wedge\ \Diamond(C_i^{v_0}) + p_1 \geq \Diamond(C_i^{\bar{v}_0}) + \bar{p}_1)\ \wedge\ \mathcal{E} \\ \mathsf{M}''(C_i, m_i) = \forall \vec{v}, \vec{p}.\, C_i^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}}\ \&\ \mathcal{D}^{(i)}\end{array}}{\vdash_{\mathsf{mc}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}}$$

   By induction hypothesis $\vdash_{\mathsf{m}} \mathsf{M}'\ \mathsf{ok}$. We have by assumption

$$T = (C_i^{r_0}; \vec{E}^{\vec{\mathfrak{r}}} \xrightarrow{m_1/m_2} H^{r_{n+1}})\ \mathsf{instanceof}\ \mathsf{M}''(C_i, m_i) \text{ i.e.} \tag{5.3.4}$$

$$\pi = \{v_0 \mapsto r_0, v_i \mapsto r_i, \bar{v}_0 \mapsto \bar{r}_0\}, \{p_i \mapsto m_i, \bar{p}_1 \mapsto \bar{n}_1\} \models \mathcal{D}^{(i)} \tag{5.3.5}$$

   Let $n = \bar{n}_1 \dot{-} n_1$. Then we have $r_0 \sqsubseteq \bar{r}_0$ and $\Diamond(C_i^{r_0}) + n_1 \geq \Diamond(C_i^{\bar{r}_0}) + \bar{n}_1$, i.e. $\Diamond(C_i^{r_0}) \geq \Diamond(C_i^{\bar{r}_0}) + n$. Then, by Lemma 4.3.4, we get $(r_0 \dot{-} \mathsf{n})_{\mathsf{C_i}} \sqsubseteq \bar{r}_0$. Let $\mathsf{M_{body}}(C, m) = e$. Then, we obtain an annotation for e by Lemma 5.3.1 such that:

$$\mathsf{M}'; \pi(\Xi); \mathtt{this}{:}C_i^{\bar{r}_0}, \vec{x}{:}\vec{E}^{\vec{\mathfrak{r}}} \vdash_{n_2}^{\bar{n}_1} e^\circ \Leftarrow H^{r_{n+1}} \tag{5.3.6}$$

   Moreover, we set $\mathsf{A_{body}}(C, m, T) = (e^\circ, n)$. We conclude $\vdash_{\mathsf{m}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}$.

2. Follows by the design of the judgement $\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$, as discussed earlier. □

Theorem 5.3.3 (Soundness of generation of a RAJA$^{\mathbf{m}}$ program) *Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be an FJEU program. Then, there exists* M *such that, for each* $C \in \mathscr{C}$ *and* $m \in \mathsf{Meth}(C)$*, if* $\mathsf{M}(C, m)$ *is a non-empty polymorphic type, then there exists* Inst*,* $\mathsf{A_{body}}$ *and* $\mathscr{V}$ *such that* $\mathscr{R}^+ = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A_{body}})$ *is a complete well-typed RAJA^m program.*

Proof. Let $\mathrm{dom}(\mathsf{M}) = \mathrm{dom}(\mathsf{Inst}) = \{(C, m) \mid C \in \mathscr{C}, m \in \mathsf{Meth}(C)\}$. By Lemma 5.3.2 we obtain values for M and a map $\mathsf{A_{body}}$ such that $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$ and $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$ if $\mathsf{S}(D) = C$. Moreover, we set $\mathscr{V} = \mathscr{V}^{\mathscr{C}}$ and $\mathsf{Inst}(C, m) = \{T \mid T\ \mathsf{instanceof}\ \mathsf{M}(C, m)\}$. □

Corollary 5.3.4 (Soundness of generation of a RAJA program) *Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be an FJEU program. Then, there exists M such that if $\mathsf{M}(C, m)$ is a non-empty polymorphic type for each $C \in \mathscr{C}$ and $m \in \mathsf{Meth}(C)$, then $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ is a well-typed RAJA program.*

Proof. By Theorem 5.3.3 we obtain a complete well-typed RAJA$^{\mathrm{m}}$ program $\mathscr{R}^{+} = (\mathscr{C}, \mathsf{main}, \mathscr{V}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$. By Lemma 5.2.3 $\mathscr{R}^{+}$ is also a complete well-typed RAJA program with explicit types. Finally, by Theorem 4.4.4, the underlying RAJA program of $\mathscr{R}^{+}$, $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ is also well-typed. □

Completeness proof

Next, we show that, when applied to a typeable expression, the constraint generation rules emit a satisfiable constraint set.

Lemma 5.3.5 (Completeness of constraint generation) *If $\mathsf{M}; \Xi; \Gamma \vdash^{n_1}_{n_2} e \leftleftarrows C^r$ and $\mathsf{M}; |\Xi|; |\Gamma| \vdash^{p_1}_{p_2} |e^{\circ}| : C^v \,\&\, \mathcal{C}$ then there exists π with $\pi(p_i) = n_i$, $\pi(v) = r$, $\pi(|\Gamma|) = \Gamma$, $\pi(|\Xi|) = \Xi$ such that $\pi \models \mathcal{C}$.*

Proof. By induction on typing derivations.

Case (∇Var) We have $\mathsf{M}; \Xi; \Gamma_0, x : E^r \vdash^{n_1}_{n_2} x^r \leftleftarrows C^r$. Then, by $(\triangle Var)$ we obtain

$$\mathsf{M}; |\Xi|; |\Gamma_0|, x : E^v \vdash^{p_1}_{p_2} x : C^u \,\&\, (E^v <: C^u \;\wedge\; p_2 \leq p_1)$$

Then, let $|\Gamma_0| = \vec{y} : \vec{F}^{\vec{w}}$. Then

$$\pi = \{v \mapsto r, u \mapsto s, w_i \mapsto 0_{\mathsf{F_i}}\}, \{p_i \mapsto n_i\} \models (E^v <: C^u \;\wedge\; p_2 \leq p_1)$$

by assumption.

Case $(\nabla MonInv.)$ We have $\mathsf{M}; \Xi; x : G^{r_0}, \vec{y} : \vec{F}^{\vec{\mathfrak{r}}} \vdash^{n_1}_{n_2} x.m(\vec{y}) : C^{r_{n+1}}$ and $\Xi(G, m) = G^{s_0}; \vec{E}^{\vec{\mathfrak{s}}} \xrightarrow{n_1/n_2} H^{s_{n+1}}$ and let $|\Xi|(G, m) = G^{u_0}; \vec{E}^{\vec{\mathfrak{u}}} \xrightarrow{q_1/q_2} H^{u_{n+1}}$. By $(\triangle MonInv)$ we get

$$\mathsf{M}; |\Xi|; x : G^{v_0}, \vec{y} : \vec{F}^{\vec{\mathfrak{v}}} \vdash^{p_1}_{p_2} x.m\,(\vec{y}) : C^{v_{n+1}} \,\&\, \mathcal{C}$$

and $\mathcal{C} = (G^{v_0} <: G^{u_0} \;\wedge\; F_i^{v_i} <: E_i^{u_i} \;\wedge\; H^{u_{n+1}} <: C^{v_{n+1}} \;\wedge\; p_1 \geq q_1 \;\wedge\; p_1 + q_2 \geq p_2 + q_1)$. Set $\pi = \{v_i \mapsto r_i\}, \{p_i \mapsto n_i\}$ and $\pi(|\Xi|) = \Xi$. Then $\pi \models \mathcal{C}$ follows by assumption.

Case ($\nabla PolyInv.$) We have $\mathsf{M}; \Xi; x : G^{r_0}, \vec{y} : \vec{F}^{\vec{\mathfrak{r}}} \vdash^{n_1}_{n_2} x.m(\vec{y}) : C^{r_{n+1}}$ and $\mathsf{constr}(\mathsf{M}(G,m)) = \mathcal{D}(\vec{u}, \vec{q}, \vec{u'}, \vec{q'})$ and we have a valuation $\pi \models \mathcal{D}$ with $\pi(u_i) = s_i$ and $\pi(q_i) = m_i$. By ($\triangle PolyInv$) we get

$$\begin{aligned}\mathsf{M}; |\Xi|; x : G^{v_0}, \vec{y} : \vec{F}^{\vec{\mathfrak{v}}} \vdash^{p_1}_{p_2} x.m(\vec{y}) : H^{v_{n+1}} \;\&\; \\ \mathcal{D}[\vec{w}/\vec{u}, \vec{t}/\vec{q}, \vec{w'}/\vec{u'}, \vec{t'}/\vec{q'}] \;\wedge\; G^{v_0} <: G^{w_0} \;\wedge\; F_i^{v_i} <: E_i^{w_i} \;\wedge \\ H^{w_{n+1}} <: C^{v_{n+1}} \;\wedge\; p_1 \geq t_1 \;\wedge\; p_1 + t_2 \geq p_2 + t_1 \quad (5.3.7)\end{aligned}$$

Then we set $\pi' = \pi(\{v_i \mapsto r_i, w_i \mapsto s_i\}, \{p_i \mapsto n_i, t_i \mapsto m_i\})$. Then $\pi' \models \mathcal{C}$ follows by assumption.

□

Lemma 5.3.6 (Completeness of constraint generation)
Let $\mathscr{R}^+ = (\mathscr{C}, \mathscr{V}^{\mathscr{C}}, \mathsf{M}, \mathsf{Inst}, \mathsf{A}_{\mathsf{body}})$ be a well-typed RAJA^m program and let N *be a map with* $\mathrm{dom}(\mathsf{N}) = \mathrm{dom}(\mathsf{M})$ *and* $\vdash_{\mathsf{mc}} \mathsf{N}\ \mathsf{ok}$. *Then for all* $(C,m) \in \mathsf{M}$ *holds* $\mathsf{N}(C,m) <: \mathsf{M}(C,m)$.

Proof. By induction on the derivation of $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$. We have $\vdash_{\mathsf{m}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}$ and $\vdash_{\mathsf{mc}} \mathsf{N}' \uplus \mathsf{N}''\ \mathsf{ok}$. W.l.o.g. we assume that $\mathrm{dom}(\mathsf{M}') = \mathrm{dom}(\mathsf{N}')$ and $\mathrm{dom}(\mathsf{M}'') = \mathrm{dom}(\mathsf{N}'')$. By induction hypothesis $\forall (C,m) \in \mathsf{M}\,.\,\mathsf{N}(C,m) <: \mathsf{M}(C,m)$. Next we show $\forall (C_i, m_i) \in \mathrm{dom}(\mathsf{N}')\,.\,\mathsf{N}'(C_i,m_i) <: \mathsf{M}'(C_i,m_i)$. Let

$$\mathsf{M}''(C_i,m_i) = \delta^{(i)} = \forall \vec{u}, \vec{q}\,.\, C_i^{u_0}; \vec{E}^{\vec{\mathfrak{u}}} \xrightarrow{q_1/q_2} H^{u_{n+1}} \;\&\; \mathcal{E}$$

$$\Xi(C_i,m_i) = T = (C_i^{s_0}; \vec{E}^{\vec{\mathfrak{s}}} \xrightarrow{n_1/n_2} H^{s_{n+1}})\ \mathsf{instanceof}\ \delta^{(i)}$$

$$\psi^{(i)} = \forall \vec{v}, \vec{p}\,.\, C_i^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; \mathcal{D}^{(i)}$$

$$\mathsf{S}(D_j) = C_i \quad \lambda_j = \begin{cases} \mathsf{N}(D_j,m_i) & \text{if } (D_j,m_i) \in \mathrm{dom}(\mathsf{N}') \\ \top^{(D_j,m_i)} & \text{if } (D_j,m_i) \in \mathrm{dom}(\mathsf{N}'') \end{cases}$$

$$\phi^{(i)} = \psi^{(i)} \vee \bigvee_j \lambda_j$$

$$\mathsf{N}'(C_i,m_i) = \xi^{(i)} = \forall \vec{v}, \vec{p}\,.\, C^{v_0}; \vec{E}^{\vec{\mathfrak{v}}} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; \bigwedge_{l \in \{1,\ldots,k\}} \mathsf{constr}(\phi^{(l)})$$

We show $\xi^{(i)} <: \delta^{(i)}$. First, we show $\phi^{(i)} <: \delta^{(i)}$ and we notice that this follows from $\psi^{(i)} <: \delta^{(i)}$ and for all j, $\lambda_j <: \delta^{(i)}$ with the l.u.b. property. We know by I.H. that for $(D_j,m_i) \in \mathrm{dom}(\mathsf{N}')$ holds $\overbrace{\mathsf{N}'(D_j,m_i)}^{\lambda_j} <: \mathsf{M}'(D_j,m_i)$ and $\mathsf{M}'(D_j,m_i) <: \overbrace{\mathsf{M}''(C_i,m_i)}^{\delta^{(i)}}$ because $\mathscr{R}^+$ is well-typed. Thus, by transitivity we get $\lambda_j <: \delta^{(i)}$. For $(D_j,m_i) \in \mathrm{dom}(\mathsf{N}'')$ is $\lambda_j = \top^{(D_j,m_i)}$ and $\top^{(D_j,m_i)} <: \delta^{(i)}$ holds trivially. Next, we show $\psi^{(i)} <: \delta^{(i)}$.
From $T\ \mathsf{instanceof}\ \delta^{(i)}$ we get $\pi = \{u_i \mapsto s_i\}, \{q_i \mapsto n_i\} \models \mathcal{E}$. We show that there exists π'' such that $\pi''|_{\mathrm{dom}(\pi)} = \pi$ and, moreover, $\pi'' \models \mathcal{D}^{(i)}$ and

$s_0 = \pi''(v_0)$ and $s_i \sqsubseteq \pi''(v_i)$ and $\pi''(v_{n+1}) \sqsubseteq s_{n+1}$ and $\pi''(p_1) \leq n_1$ and $\pi''(p_2) \geq n_2$, where

$$\mathcal{D}^{(i)} = \mathcal{C} \ \wedge \ v_0 \sqsubseteq \bar{v}_0 \ \wedge \ \Diamond\!\!\!\!\Diamond(C_i^{v_0}) + p_1 \geq \Diamond\!\!\!\!\Diamond\big(C_i^{\bar{v}_0}\big) + \bar{p}_1 \tag{5.3.8}$$

By assumption we have $\mathsf{A}_{\mathsf{body}}(C_i, m_i, T) = (e^\circ, p)$ and

$$\mathsf{M}; \Xi; \mathtt{this}\!:\!C_i^{r_0}, x_1\!:\!E_1^{r_1}, \ldots, x_j\!:\!E_j^{r_j} \ \vdash^{\frac{n_1+p}{n_2}} \ e^\circ : H^{s_{n+1}} \tag{5.3.9}$$

$$(s_0 \dot{-} \mathsf{p})_{\mathsf{C_i}} \sqsubseteq r_0 \tag{5.3.10}$$

$$s_i \sqsubseteq r_i \tag{5.3.11}$$

$$\Diamond\!\!\!\!\Diamond(C_i^{s_0}) \geq p \tag{5.3.12}$$

By Lemma 5.3.5 we obtain

$$\begin{aligned} &\mathsf{M}; |\Xi|; \mathsf{this}\!:\!C_i^{\bar{v}_0}, x_1\!:\!E_1^{v_1}, \ldots, x_n\!:\!E_n^{v_n} \ \vdash^{\bar{p}_1}_{p_2} \ |e^\circ|\!:\!H^{v_{n+1}} \ \&\ \mathcal{C} \\ &\quad \pi' \supseteq (\{\bar{v}_0 \mapsto r_0, v_i \mapsto r_i, v_{n+1} \mapsto s_{n+1}\}, \{\bar{p}_1 \mapsto n_1 + p, p_2 \mapsto n_2\}) \models \mathcal{C} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{and} \ \ \pi'(|\Xi|) = \Xi \end{aligned} \tag{5.3.13}$$

Now we set

$$\tilde{\pi} = \pi\pi'(\{v_0 \mapsto s_0\}, \{p_1 \mapsto n_1\}) \tag{5.3.14}$$

We show the following items:

- $\tilde{\pi} \models \mathcal{C}$ follows by (5.3.13) and (5.3.14).
- $$\begin{array}{rlll} \tilde{\pi} \models & v_0 & \sqsubseteq \ \bar{v}_0 & \iff_{(5.3.14)} \\ & s_0 & \sqsubseteq \ r_0 & \Leftarrow_{\text{by Lemma 4.3.4}} \\ & (s_0 \dot{-} \mathsf{p})_{\mathsf{C_i}} & \sqsubseteq \ r_0 & \Leftarrow_{(5.3.10)} \end{array}$$
- $$\begin{array}{rlll} \tilde{\pi} \models & \Diamond\!\!\!\!\Diamond(C_i^{v_0}) + p_1 & \geq \ \Diamond\!\!\!\!\Diamond\big(C_i^{\bar{v}_0}\big) + \bar{p}_1 & \iff_{(5.3.13)} \\ & \Diamond\!\!\!\!\Diamond(C_i^{s_0}) + n_1 & \geq \ \Diamond\!\!\!\!\Diamond(C_i^{r_0}) + n_1 + p & \iff \\ & \Diamond\!\!\!\!\Diamond(C_i^{s_0}) \dot{-} p & \geq \ \Diamond\!\!\!\!\Diamond(C_i^{r_0}) & \text{follows by (5.3.10)} \end{array}$$
- $s_0 = \tilde{\pi}(v_0)$ follows by (5.3.14).
- $s_i \sqsubseteq \tilde{\pi}(v_i)$ follows by (5.3.11) and (5.3.13).
- $\tilde{\pi}(v_{n+1}) \sqsubseteq s_{n+1}$ follows by (5.3.13).
- $\tilde{\pi}(p_1) \leq n_1$ follows by (5.3.14).
- $\tilde{\pi}(p_2) \geq n_2$ follows by (5.3.13).

Altogether we have shown $\tilde{\pi} \models \mathcal{D}^{(i)}$ and we have shown that $\tilde{\pi}$ is the desired π''. Finally we notice $\forall l \neq i \,.\, \tilde{\pi} \models \mathsf{constr}(\phi^{(l)})$ since $\tilde{\pi}(|\Xi|)(C_l, m_l) = \Xi(C_l, m_l)$ and $\Xi(C_l, m_l)$ $\mathsf{instanceof}$ $\mathsf{N}''(C_l, m_l)$. Thus, $\xi^{(i)} <: \delta^{(i)}$ follows. $\square$

5.4 Analysing the heap-space requirements of FJEU programs

In this section we shall investigate how to apply a RAJA program to the analysis of the heap-space requirements of the underlying FJEU program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$. In particular, we wish to compute an upper bound on the amount of heap-cells needed for executing the method main as a function of main's arguments. By the results in Chapter 3, this follows from the potential given to main's arguments by its RAJA type. We have already seen in the previous section how to obtain a polymorphic RAJA type for main, but, for being able to read off the potential from that type, we need a concrete instance of the type, which we can obtain by solving the type's constraints.

Figure 5.6 shows the schematic structure of the algorithm for building a monomorphic RAJA method type for main.

The first step of the algorithm is the generation of a polymorphic RAJA method type for main which was described in Section 5.3. The next step is solving the constraints, i.e. obtaining a valuation mapping view variables to views and arithmetic variables to numbers that satisfies the constraints, and building the monomorphic RAJA method type based on those views and real numbers.

Whereas solving linear arithmetic constraints is easily achieved by an LP-Solver, solving subtyping constraints is more challenging. The task of solving a constraint $C^u <: D^v$ can be reduced to the tasks of solving $C <: D$ and $u \sqsubseteq v$, by the definition of subtyping. $C <: D$ can be checked easily by analysing the inheritance relations in the program. Thus, the real challenge is solving $u \sqsubseteq v$. Solving these kind of constraints is difficult for various reasons.

First, the views are infinite objects, and the subtyping relation over views is defined coinductively. Thus, unfolding the definition of subtyping; that is, trying to solve the constraints $\Diamond\!(C^u) \geq \Diamond\!(C^v)$ for each $C \in \mathcal{C}$ and $\mathsf{A}^{\mathrm{get}}(C^u, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^v, a)$ and $\mathsf{A}^{\mathrm{set}}(C^v, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^u, a)$ for each $a \in \mathsf{A}(C)$ would lead to more unfolding steps and this process would not terminate.

Second, subtyping over views is covariant in the get views and contravariant in the set views. The contravariance also brings difficulties. For this reason, we study in Chapter 6 a simpler type of infinite trees than views, that have nonnegative real numbers in the nodes. For these trees, we define an inequality relation that is covariant in all cases. Because these trees are simpler objects, solving constraints over them is simpler than solving constraints over views. Thus, we solve the inequalities over views by reducing them to inequalities over infinite trees.

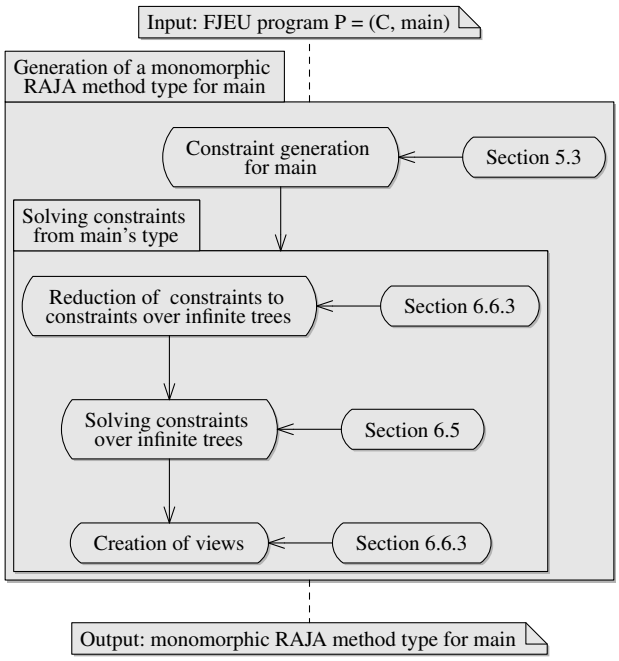


Figure 5.6: Schematic structure of the algorithm for building a monomorphic RAJA method type for the main method of an FJEU program.

In Chapter 6 we describe the infinite trees. For solving constraints over infinite trees, we still have the problem that unfolding the inequality relation would not terminate. This is why, to ensure termination of unfolding, we present a heuristic algorithm for solving these constraints in Section 6.5 that assumes that the solutions to the constraints are regular infinite trees. This implies, however, that the algorithm is not able to solve all the constraints but only a subset of them that admit regular solutions. Therefore, when using this algorithm, we can solve only subtyping constraints that admit regular views as a solution, which correspond to programs whose heap-space consumption is a linear function of its input.

Hence, the algorithm that we present in this thesis can compute only linear bounds on the heap-space requirements of programs. However, we remark that this is because no better algorithm for solving the constraints over infinite trees is known at the moment. In fact, it is unknown whether the problem of solving those constraints is decidable. If the problem was decidable, and an algorithm for solving the constraints is found, we could, with the same general mechanism, analyse programs whose heap-space consumption can be described as an non-linear function.

We present the reduction from subtyping and arithmetic constraints to inequalities over trees and arithmetic constraints in Section 6.6.3. The re-

duction delivers sets of tree variables that correspond to view variables. Then, by solving the constraints over infinite trees by the previously mentioned algorithm, we obtain a solution to the constraints, i.e. a valuation that maps those tree variables to regular infinite trees and arithmetic variables to numbers. When we combine those trees together to build views, we obtain a map from view variables to regular views that, together with the map from arithmetic variables to numbers, represents a solution to the original subtyping and arithmetic constraints.

5.5 Generating a finite RAJA program with explicit types

We have seen in the previous section how to compute bounds on the heap-space requirements of an FJEU program by giving a monomorphic RAJA method type to the program's main method. Here we show how to generate all the monomorphic RAJA method types for all the methods that are necessary to justify the RAJA typing of main. If we are able to do this, and we obtain a finite set of monomorphic RAJA method types for each method, then we can implement the efficient type checking algorithm described in Chapter 4. This way, we would obtain efficiently verifiable certificates that validate the prediction of the heap-space requirements of FJEU programs given by the analysis. In the following we show how we can achieve this.

First, we notice that the algorithm for generating constraints from Section 5.3 generates a finite amount of constraints for main. This follows by the fact that each rule generates a finite amount of constraints and the constraint generation algorithm is not recursive. This is the main advantage of not allowing polymorphic recursion.

Next, we notice that the polymorphic RAJA method type of main contains the constraints of all the methods that are relevant to main's execution. For instance, if a method m is called n times during main's execution, then its constraints appear in main's type n times, each time containing different variables. Let $\mathcal{D}(\vec{w}, \vec{t})$ be the constraints from m's type and let $\mathcal{C}_i = \mathcal{D}[\vec{v}^{(i)}/\vec{w}, \vec{p}^{(i)}/\vec{t}]$. Further, let $\mathcal{C} = \mathcal{E} \wedge \bigwedge_{i=1\ldots n} \mathcal{C}_i(\vec{v}^{(i)}, \vec{p}^{(i)})$ be the constraints of main's type, and let π be a valuation that satisfies $\mathcal{C}$, which we can find in some cases by the algorithm mentioned earlier. Then $\pi_i = \pi|_{\vec{v}^{(i)}, \vec{p}^{(i)}}$ satisfies $\mathcal{C}_i$. Thus, we can build a monomorphic RAJA method type T_i for m based on π_i. That way, we build a set of n monomorphic RAJA method types for the method m.

The soundness of the so created finite RAJA program with explicit types follows by the soundness of the algorithms for constraint generation and solving, described in Section 5.3 and Section 6.6.4, respectively.

Chapter 6

Linear Constraints over Infinite Trees

6.1 Overview

In this chapter we present a new algorithmic problem related to linear arithmetic over $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$. Indeed, it can be seen as a special case of linear arithmetic with infinitely many variables (with some schematic notation so as to make instances of the problem finite objects).

While in general linear arithmetic with infinitely many variables is easily seen to be undecidable (introduce a variable x_{it} for every position i and time t of a computation on a Turing machine) the question of decidability for our special case remains open. We do, however, provide a heuristic solution for an important subcase motivated by the type inference algorithm for RAJA, which we discuss in Chapter 5.

We begin with an informal description of our constraint systems. We have arithmetic variables that take on values in $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ and *tree variables* whose values are infinite trees whose nodes are labelled with elements of $\mathbb{D}$. We fix a finite set $\mathcal{L} = \{l_1, \ldots, l_n\}$ of labels to address the children of a node, e.g. $\mathcal{L} = \{L, R\}$ for infinite binary trees and $\mathcal{L} = \{tl\}$ for infinite lists.

Such trees can be added, scaled, and compared componentwise; furthermore, we have an operation $\Diamond(.)$ that extracts the root label of a tree, thus if t is a tree expression then $\Diamond(t)$ is an arithmetic expression. Finally, if t is a tree expression and $l \in \mathcal{L}$ then $l(t)$ is a tree expression denoting the l-labelled immediate subtree of t.

Given a system of constraints built from these constructions we can ask for satisfiability and for values of selected arithmetic variables. Asking for values of tree variables makes no sense in general as these are infinite objects. We can also ask for the optimum value of some linear combination of the arithmetic variables.

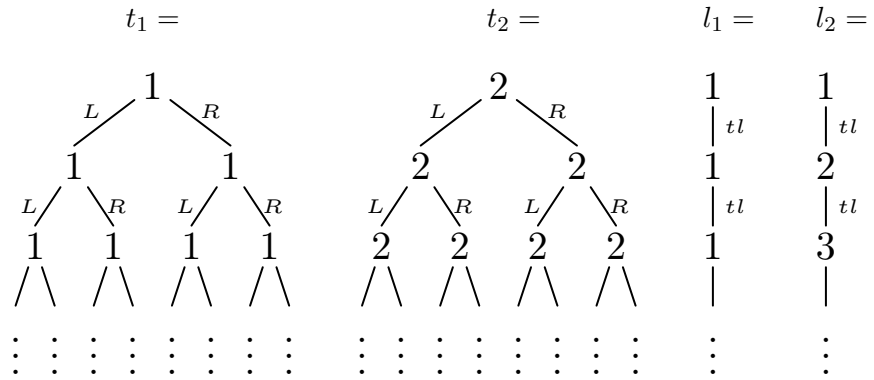


Figure 6.1: Some infinite trees.

In Fig. 6.1 two infinite trees t_1, t_2 over label set $\mathcal{L} = \{L, R\}$ are defined. It also contains two infinite trees over label set $\mathcal{L} = \{tl\}$ which are effectively infinite lists. Within one and the same constraint system we can only use trees over one and the same label set. These trees satisfy for example: $L(t_1) = R(t_1) = t_1$, $t_1 \sqsubseteq t_2$, $l_2 \sqsubseteq l_1$, $\Diamond(t_1) = 1$, $\Diamond(t_2) = 2$. We also have $t_1 + t_1 = t_2$ and $2t_1 = t_2$ and $tl(l_1) = l_1 + l_2$.

Now, the constraint system $tl(x) \sqsupseteq x \wedge \Diamond(x) \geq 1$ is satisfiable, for example with $x = l_1$ and its optimum value with respect to the objective $c = \Diamond(x)$ to be minimised equals 1. The constraint system $L(x) \sqsupseteq x \wedge R(x) \sqsupseteq x \wedge \Diamond(x) \geq 6$ is satisfiable, for example with $x = 6t_1$.

The constraint system $\Diamond(x) \geq 1 \wedge\ 2tl(x) = tl(x)$ is also satisfiable, namely by $x = 10^\omega$, but $\Diamond(x) \geq 1 \wedge\ 2tl(x) = tl(x) \wedge\ x = tl(x)$, however, is unsatisfiable. As already mentioned, we currently do not know whether satisfiability of such constraint systems is in general decidable, but the heuristic method we shall present covers all the constraint systems given so far. This is because, the trees witnessing satisfiability were *regular* in the sense that their set of subtrees is finite. So, t_1, t_2, l_2 are regular, but l_1 is not. Accordingly, a constraint system like $\Diamond(x) \geq 1 \wedge\ tl(x) \sqsupseteq x \wedge\ tl(y) \sqsupseteq x + y$ is not amenable to our heuristic as it does not admit a regular solution.

In order to decide satisfiability of constraints in general it is tempting to use Büchi tree automata; however, in order to represent our "arithmetic" trees as a tree whose nodes are labelled with letters from a finite alphabet, we would have to represent the numerical annotations using extra branches and then primitive predicates such as equality cannot be recognised by a Büchi tree automaton. Indeed, we conjecture that the algebraic structure of arithmetic trees is not "automatic" in the sense of [BG00].

Nevertheless, we believe that satisfiability of our constraint systems is decidable; in support of this conjecture, we can enlist the fact that the set of solutions to a constraint system is *convex* in the sense that if $\vec{t_1}$ and $\vec{t_2}$ are both solutions then so is $(1-\lambda)\vec{t_1} + \lambda\vec{t_2}$ for $\lambda \in [0, 1]$. Furthermore, constraint

systems can be reduced by algebraic manipulations and elimination steps to canonical forms from which solutions can be read off.

We encountered these constraint systems as part of our endeavour of developing an automatic type inference for the object-oriented resource type system presented in this thesis. We hope, though, that due to their compact and general formulation our arithmetic tree constraint systems will find other applications beyond type inference as well.

We were surprised to find practically no directly related work. One notable exception is [DV07] where constraint satisfaction problems with infinitely many variables are introduced and studied. The difference to our work is twofold: first, the range of individual variables in *loc.cit.* is finite, e.g. Boolean in contrast to $\mathbb{D}$ in our case; secondly, the access policy is much more general and leads to undecidability in general. Interestingly, the near absence of related work has also been noted in *loc.cit.*

This chapter is organised as follows. Section 6.2 describes the infinite trees formally. Then, Section 6.3 describes constraints over infinite trees and arithmetic constraints containing tree expressions. This section also lists important problems regarding these constraints like satisfiability. Then, Section 6.4 shows an algorithm that partially solves the problem of eliminating variables from constraints. Further, Section 6.5 describes a heuristic algorithm for solving satisfiability. Finally, Section 6.6 presents a reduction from subtyping and arithmetic constraints to constraints over trees and arithmetic constraints. This section finishes with an algorithm for solving the subtyping and arithmetic constraints, which uses the mentioned reduction and the heuristic algorithm described in Section 6.5.

6.2 Infinite trees

In this section we present infinite trees labelled with nonnegative real numbers. Fix a finite set of labels $\mathcal{L} = \{l_1, \ldots, l_n\}$.

Definition 6.2.1 *The set* $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ *of infinite trees is given by* $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} = \{t \mid t : \mathcal{L}^* \to \mathbb{D}\}$ *where* $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ *with* $0 \in \mathbb{R}^+$.

We will refer to elements $w \in \mathcal{L}^*$ as *paths*. We denote the empty word by ϵ. We write $\bar{w}$ for the reverse of w and $|w|$ for the length of w, where $|\epsilon| = 0$ and $|lw| = |w| + 1$.

A tree t' is a *sub-tree* of a tree t if there exists $w \in L^*$ so that $t'(p) = t(w\,p)$ for all $p \in L^*$. Further, we say that an infinite tree is *regular* if it contains a finite number of different sub-trees. The set $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ carries a final coalgebra structure consisting of the function

$$\begin{array}{lcl} \langle \Diamond, \mathsf{step} \rangle & : & \mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \to \mathbb{D} \times (\mathcal{L} \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \\ t & \mapsto & \langle t(\epsilon), \lambda l\, w\,.\, t(l\,w) \rangle \end{array}$$

where $\mathsf{step}\, l_i$ returns the ith subtree, and $\Diamond$ gives the label of the root node tree. In [SR10], Silva and Rutten give a similar final coalgebra structure for binary infinite trees. We write l_i as a short notation for $\mathsf{step}\, l_i$. Let U be a domain. Every family of functions $lt_i : U \to U$ and $o : U \to \mathbb{D}$ defines a function $h : U \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, such that:

$$\Diamond(h(x)) = o(x) \qquad l_i(h(x)) = h(lt_i(x))$$

We define a preorder $\sqsubseteq$ between trees as follows:

Definition 6.2.2 *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. We define $t \sqsubseteq t'$ coinductively by $t \sqsubseteq t' \iff$*

$$\Diamond(t) \leq \Diamond(t') \tag{6.2.1}$$
$$l_i(t) \sqsubseteq l_i(t') \quad \textit{for all } l_i \in \mathcal{L} \tag{6.2.2}$$

Alternatively, we can define the same preorder pointwise by:

Definition 6.2.3 *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then $t \sqsubseteq_{\mathsf{ind}} t' \iff$ for all paths $w \in L^*$:*

$$t(w) \leq t'(w) \tag{6.2.3}$$

Lemma 6.2.4 *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then:*

1. *$l_i(t)(w') = t(l_i w')$ for all $l_i \in \mathcal{L}$ and $w' \in \mathcal{L}^*$.*
2. *$t \sqsubseteq_{\mathsf{ind}} t' \iff \Diamond(t) \leq \Diamond(t')$ and $l_i(t) \sqsubseteq_{\mathsf{ind}} l_i(t')$ for all $l_i \in \mathcal{L}$.*
3. *$t \sqsubseteq_{\mathsf{ind}} t' \iff t \sqsubseteq t'$.*

Proof.

1. Follows by unfolding definitions:

$$\begin{array}{rcl} l_i(t)(w') & = & (\mathsf{step}\, l_i\, t)(w') \\ & = & \lambda w \,.\, t(l_i\, w)(w') \\ & = & t(l_i\, w') \end{array}$$

2. By definition,

$$\begin{array}{rcll} t \sqsubseteq_{\mathsf{ind}} t' & \iff & \text{for each } \; w \in \mathcal{L}^* \,.\, t(w) \leq t'(w) & \\ & \iff & \text{for each } \; l_i \in \mathcal{L}, w' \in \mathcal{L}^* \,. & \\ & & t(\epsilon) \leq t'(\epsilon) \;\text{ and }\; t(l_i w') \leq t'(l_i w') & \\ & \iff_{1.} & \text{for each } \; l_i \in \mathcal{L}, w' \in \mathcal{L}^* \,. & \\ & & \Diamond(t) \leq \Diamond(t') \;\text{ and }\; l_i(t)(w') \leq l_i(t')(w') & \\ & \iff & \text{for each } \; l_i \in \mathcal{L}, w' \in \mathcal{L}^* \,. & \\ & & \Diamond(t) \leq \Diamond(t') \;\text{ and }\; l_i(t) \sqsubseteq_{\mathsf{ind}} l_i(t') & \end{array}$$

3. Follows by 2.

□

We extend the inequality relation over infinite trees to inequality over pairs of trees in the straightforward manner, i.e. if $t_1, t_2, t_1', t_2' \in \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$ then

$$(t_1, t_2) \sqsubseteq (t_1', t_2') \iff t_1 \sqsubseteq t_2 \text{ and } t_1' \sqsubseteq t_2'$$

We define addition of trees ($+ : \mathsf{T}_{\mathbb{D}}^{\mathcal{L}} \times \mathsf{T}_{\mathbb{D}}^{\mathcal{L}} \to \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$) by:

$$\Diamond(t + t') = \Diamond(t) + \Diamond(t') \qquad l_i(t + t') = l_i(t) + l_i(t')$$

and multiplication of trees with a nonnegative scalar ($\cdot : \mathbb{R}^+ \times \mathsf{T}_{\mathbb{D}}^{\mathcal{L}} \to \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$) by:

$$\Diamond(c \cdot t') = c \cdot \Diamond(t') \qquad l_i(c \cdot t') = c \cdot l_i(t')$$

Lemma 6.2.5 *Let $t_1, t_2 \in \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$. For all $w \in \mathcal{L}^*$ holds $(t_1 + t_2)(w) = t_1(w) + t_2(w)$.*

Proof. By induction on w, using Lemma 6.2.4, item 1. □

We say that a function $f : A \to B$ where $A, B \in \{\mathsf{T}_{\mathbb{D}}^{\mathcal{L}}, \mathsf{T}_{\mathbb{D}}^{\mathcal{L}} \times \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}\}$ is order-preserving or monotone if $t \sqsubseteq t'$ implies $f(t) \sqsubseteq f(t')$.

Lemma 6.2.6 (Order-preserving functions)

1. *For every $l \in \mathcal{L}$ holds l is order preserving.*
2. *$+$ is order preserving.*

Proof.

1. Let $t, t' \in \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$ with $t \sqsubseteq t'$. Then $l(t) \sqsubseteq l(t')$ follows by definition.
2. By coinduction. Let $t_1, t_2, t_1', t_2' \in \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$ and let $(t_1, t_2) \sqsubseteq (t_1', t_2')$. We show $t_1 + t_2 \sqsubseteq t_1' + t_2'$, which follows from
   (a) $\Diamond(t_1) + \Diamond(t_2) \leq \Diamond(t_1') + \Diamond(t_2')$, which follows by assumption and the monotonicity of $+$ over $\mathbb{D}$.
   (b) $l(t_1) + l(t_2) \sqsubseteq l(t_1') + l(t_2')$, which follows by coinduction hypothesis, since $l(t_1) \sqsubseteq l(t_2)$ and $l(t_1') \sqsubseteq l(t_2')$, by assumption and 1.

□

Definition 6.2.7 *Let $T, T' \subseteq \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$ and $l \in \mathcal{L}$. Then we define $l(T), T + T'$ pointwise by:*

$$\begin{aligned} l(T) &= \{l(t) \mid t \in T\} \\ T + T' &= \{t + t' \mid t \in T, t' \in T'\} \end{aligned}$$

Defining a complete lattice

Recall that an ordered set P is a bounded lattice if P is a lattice and it contains a bottom and a top element. Moreover, P is a complete lattice if, for each subset $S \subseteq P$, $\bigvee S$ (the *join* of S) and $\bigwedge S$ (the *meet* of S) exist. For more information on lattices see, for instance, Davey and Priestley's "Introduction to Lattices and Order: Second Edition" [DP02].

The domain $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ is a complete lattice under its usual order by the completeness axiom for $\mathbb{R}$ and because it has a top and a bottom element: ∞ and 0. For each $d \in \mathbb{D}$ we define $\widehat{d} \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ by $\Diamond\left(\widehat{d}\right) = d$ and $l_i(\widehat{d}) = \widehat{d}$ for each $l_i \in \mathcal{L}$. Then, $\widehat{\infty}$ is the top element in $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $\widehat{0}$ the bottom. In the following we will show that $(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}, \sqsubseteq)$ is a bounded complete lattice. For each subset of $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, we define its join and its meet as follows.

- $\bigwedge : \mathcal{P}(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is totally determined by:

$$\Diamond(\bigwedge T) = \min_{t \in T}(\Diamond(t)) \qquad l_i(\bigwedge T) = \bigwedge l_i(T)$$

- $\bigvee : \mathcal{P}(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is totally determined by:

$$\Diamond(\bigvee T) = \max_{t \in T}(\Diamond(t)) \qquad l_i(\bigvee T) = \bigvee l_i(T)$$

Lemma 6.2.8 (Complete lattice) *Let $t \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $T \subseteq \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then:*

1. $\widehat{0} \sqsubseteq t$.
2. $t \sqsubseteq \widehat{\infty}$.
3. $\bigvee T$ *is the least upper bound of* T.
4. $\bigwedge T$ *is the greatest lower bound of* T.
5. $(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}, \sqsubseteq)$ *is a bounded complete lattice.*

Proof.

1. The goal follows by $0 \leq \Diamond(t)$, which follows trivially, and by $\widehat{0} \sqsubseteq l(t)$ for each $l \in \mathcal{L}$, which follows by the coinduction hypothesis.
2. The goal follows by $\Diamond(t) \leq \infty$, which follows trivially, and by $l(t) \sqsubseteq \widehat{\infty}$ for each $l \in \mathcal{L}$, which follows by the coinduction hypothesis.
3. By coinduction. The goal follows by
   (a) $\Diamond(\bigvee T) = \max_{t \in T} \Diamond(t)$ is the l.u.b. of $\{\Diamond(t) \mid t \in T\}$, which follows by the fact that $\mathbb{D}$ is a complete lattice.

(b) $l(\bigvee T)$ is the l.u.b. of $l(T)$. It follows by the coinduction hypothesis, since $l(\bigvee T) = \bigvee l(T)$, by definition.

4. By coinduction. The goal follows by

   (a) $\Diamond\!\!\!|(\bigwedge T) = \min_{t \in T} \Diamond\!\!\!|(t)$ is the g.l.b. of $\{\Diamond\!\!\!|(t) \mid t \in T\}$, which follows by the fact that $\mathbb{D}$ is a complete lattice.

   (b) $l(\bigwedge T)$ is the g.l.b. of $l(T)$. It follows by the coinduction hypothesis, since $l(\bigwedge T) = \bigwedge l(T)$, by definition.

5. It follows by the items 1. to 4.

□

Lemma 6.2.9 (Functions that preserve $\bigwedge, \bigvee$) *Let $l \in L$. The functions l and $+$ preserve joins and meets. Let $T, T' \subseteq \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then:*

1. $l(\bigwedge T) = \bigwedge l(T)$.
2. $l(\bigvee T) = \bigvee l(T)$.
3. $\bigwedge T + \bigwedge T' = \bigwedge(T + T')$.
4. $\bigvee T + \bigvee T' = \bigvee(T + T')$.

Proof.

1. Follows by definition of $\bigwedge T$.
2. Follows by definition of $\bigvee T$.
3. By coinduction. The goal follows by

   (a)

   $$\begin{array}{lcll} \Diamond\!\!\!|(\bigwedge T) + \Diamond\!\!\!|(\bigwedge T') & = & \Diamond\!\!\!|(\bigwedge T + T') & \Longleftrightarrow \\ \min_{t \in T} \Diamond\!\!\!|(t) + \min_{t' \in T'} \Diamond\!\!\!|(t') & = & \min_{t'' \in T + T'} \Diamond\!\!\!|(t'') & \\ & = & \min_{t \in T, t' \in T'} \Diamond\!\!\!|(t) + \Diamond\!\!\!|(t') & \end{array}$$

   which follows trivially.

   (b) Let $l \in L$. We show:

   $$\begin{array}{lcll} l(\bigwedge T) + l(\bigwedge T') & = & l(\bigwedge T + T') & \Longleftrightarrow \text{ 1.} \\ \bigwedge(l(T)) + \bigwedge(l(T')) & = & \bigwedge l(T + T') & \\ & = & \bigwedge(l(T) + l(T')) & \end{array}$$

   which follows by the coinduction hypothesis.

4. Similar to 3.

□

6.3 Constraints

In this section we consider a system of inequalities among tree expressions and a system of linear arithmetic constraints. Let X be a fixed, countably infinite set of tree variables and Λ be a fixed countably infinite set of arithmetic variables where $X \cap \Lambda = \emptyset$. We write TAExp to denote the set of tree expressions that represent a path. We call these expressions *atomic*. The set TExp denotes expressions that represent either a path or a sum of paths. We call expressions in TExp, that are not atomic, *compound*. Moreover, we write AExp to denote linear arithmetic expressions. An arithmetic expression is either a number n, an arithmetic variable λ, an expression representing a potential found at some path $\Diamond(\mathsf{tae})$ or a sum of two expressions $\mathsf{ae}_1 + \mathsf{ae}_2$. We build the sets of *valid* expressions TExp and AExp by the following grammar, where $x \in X$, $n \in \mathbb{D}$, $\lambda \in \Lambda$ and $l \in \mathcal{L}$.

$$\begin{array}{lcll} \mathsf{tae} & ::= & x \mid l(\mathsf{tae}) & \in \mathsf{TAExp} \\ \mathsf{te} & ::= & \mathsf{tae} \mid \mathsf{te} + \mathsf{te} & \in \mathsf{TExp} \\ \mathsf{ae} & ::= & n \mid \lambda \mid \Diamond(\mathsf{tae}) \mid \mathsf{ae} + \mathsf{ae} & \in \mathsf{AExp} \\ \mathsf{tc} & ::= & \mathsf{te} \sqsubseteq \mathsf{te} & \in \mathsf{TConstr} \\ \mathsf{ac} & ::= & \mathsf{ae} \leq \mathsf{ae} & \in \mathsf{AConstr} \end{array}$$

A system of constraints is a set of valid tree constraints and arithmetic constraints, i.e. a pair $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ where $\mathcal{TC}$ and $\mathcal{AC}$ are finite subsets of $\mathsf{TConstr}$ and $\mathsf{AConstr}$ respectively. We write $\mathsf{Vars}(\mathsf{te}) \subseteq X$ for the set of tree variables that occur in the tree expression te and $\mathsf{Vars}(\mathsf{ae}) \subseteq X \cup \Lambda$ for the set of tree and arithmetic variables that appear in the arithmetic expression ae. Moreover, we write $\mathsf{Vars}(\mathcal{C})$ for the set of tree and arithmetic variables that appear in $\mathcal{C}$. Sometimes we write $\mathcal{C}(\vec{x}, \vec{\lambda})$ as a short notation for $\mathsf{Vars}(\mathcal{C}) = \vec{x}, \vec{\lambda}$.

Meaning of constraints

Let $\pi = (\pi_t, \pi_a)$ where $\pi_t : X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $\pi_a : \Lambda \to \mathbb{D}$. The meaning of arithmetic expressions $\pi(\mathsf{ae}) : \mathbb{D}$ is defined by

$$\begin{array}{lcl} \pi(n) & = & n \\ \pi(\lambda) & = & \pi_a(\lambda) \\ \pi(\Diamond(\mathsf{tae})) & = & \Diamond(\pi(\mathsf{tae})) \\ \pi(\mathsf{ae}_1 + \mathsf{ae}_2) & = & \pi(\mathsf{ae}_1) + \pi(\mathsf{ae}_2) \end{array}$$

The meaning of tree expressions $\pi(\mathsf{te}) : \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is defined by

$$\begin{array}{lcl} \pi(x) & = & \pi_t(x) \\ \pi(l(\mathsf{tae})) & = & l(\pi(\mathsf{tae})) \\ \pi(\mathsf{te}_1 + \mathsf{te}_2) & = & \pi(\mathsf{te}_1) + \pi(\mathsf{te}_2) \end{array}$$

Then, π satisfies a tree constraint $\mathsf{te} \sqsubseteq \mathsf{te}'$ (written $\pi \models \mathsf{te} \sqsubseteq \mathsf{te}'$) if $\pi(\mathsf{te}) \sqsubseteq \pi(\mathsf{te}')$. Similarly, π satisfies an arithmetic constraint $\mathsf{ae}_1 \leq \mathsf{ae}_2$ ($\pi \models \mathsf{ae}_1 \leq \mathsf{ae}_2$) if $\pi(\mathsf{ae}_1) \leq \pi(\mathsf{ae}_2)$. Finally, we say π satisfies a system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ if $\pi \models \mathsf{tc}$ for each $\mathsf{tc} \in \mathcal{TC}$ and $\pi \models \mathsf{ac}$ for each $\mathsf{ac} \in \mathcal{AC}$.

We say that a variable x occurs *projected* in a set of tree constraints when x appears in (sub)expressions $l(\mathsf{tae})$, $\Diamond(\mathsf{tae})$. If x appears as a variable (sub)expression "x" we say that x occurs *as a whole*. We write $\mathcal{C}(x^{\mathsf{proj}})$ for the subset of $\mathcal{C}$ where x occurs projected and we write $\mathcal{C}(x^{\mathsf{whole}})$ for the subset of $\mathcal{C}$ where x appears as a whole.

Given a tree expression te and a path w we define $\mathsf{te}_w : \mathsf{AExp}$ inductively by

$$\begin{array}{lcl} \mathsf{te}_\epsilon & = & \Diamond(\mathsf{te}) \\ \mathsf{te}_{lw} & = & l(\mathsf{te})_w \end{array} \tag{6.3.1}$$

te_w represents the arithmetic expression we obtain starting from te after following the path w. The resulting expression may not be valid, but it can be transformed easily into an equivalent valid one with the following transformations:

$$\begin{array}{lcl} \Diamond(\mathsf{tae}_1 + \mathsf{tae}_2) & = & \Diamond(\mathsf{tae}_1) + \Diamond(\mathsf{tae}_2) \\ l(\mathsf{tae}_1 + \mathsf{tae}_2) & = & l(\mathsf{tae}_1) + l(\mathsf{tae}_2) \end{array} \tag{6.3.2}$$

For example, $(x+y)_l = \Diamond(l(x+y))$ is not valid but it is equivalent to $\Diamond(l(x))+\Diamond(l(y))$.

Moreover, we define substitution of tree variables with tree expressions in constraints $\mathcal{C}\,[\mathsf{te}/x]$, substitution of tree expressions with tree variables $\mathcal{C}\,[x/\mathsf{te}]$ and substitution of arithmetic expressions with arithmetic variables $\mathcal{C}\,[\mathsf{ae}/\lambda]$ as usual and ensure that the resulting constraints are valid, again by the transformations (6.3.2).

Next, we define the maximal nesting depth of expressions that contain a given variable x in a system of constraints $\mathcal{C}$.

Definition 6.3.1 (Nesting depth of expressions containing x) *Let $\mathcal{C}$ be a system of constraints and $x \in X$. We define the maximal* nesting depth *of the (sub)expressions in $\mathcal{C}$ that contain x, written $\mathsf{nd}_x(\mathcal{C})$, inductively as shown in Fig. 6.2*

The following function $\mathsf{unfold}(\mathcal{TC})$ unrolls the definition of inequality ($\sqsubseteq$) in the constraints once. The validity of the resulting constraints is ensured by applying the transformations (6.3.2).

Definition 6.3.2 (Unfold constraints) *Let $\mathcal{TC}$ be a set of tree constraints. We define a function* $\mathsf{unfold}(\mathcal{TC})$ *by unfolding the definition of $\sqsubseteq$ and obtaining the respective system of constraints.*

$$\mathsf{unfold}(\mathcal{TC}) = \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \bigcup_{l \in \mathcal{L}} \{l(\mathsf{te}) \sqsubseteq l(\mathsf{te}')\}, \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \{\Diamond(\mathsf{te}) \leq \Diamond(\mathsf{te}')\}$$

Nesting depth $\mathsf{nd}_x(\mathcal{C})$

$$
\begin{array}{lcl}
\mathsf{nd}_x(y) = \mathsf{nd}_x(x) & = & 0 \\
\mathsf{nd}_x(l(\mathsf{tae})) & = & \text{if } x \in \mathsf{Vars}(\mathsf{tae}) \text{ then } 1 + \mathsf{nd}_x(\mathsf{tae}) \text{ else } 0 \\
\mathsf{nd}_x(\mathsf{te}_1 + \mathsf{te}_2) & = & \mathsf{max}(\mathsf{nd}_x(\mathsf{te}_1), \mathsf{nd}_x(\mathsf{te}_2)) \\
 & & \\
\mathsf{nd}_x(n) = \mathsf{nd}_x(\lambda) & = & 0 \\
\mathsf{nd}_x(\Diamond(\mathsf{tae})) & = & \text{if } x \in \mathsf{Vars}(\mathsf{tae}) \text{ then } 1 + \mathsf{nd}_x(\mathsf{tae}) \text{ else } 0 \\
\mathsf{nd}_x(\mathsf{ae}_1 + \mathsf{ae}_2) & = & \mathsf{max}(\mathsf{nd}_x(\mathsf{ae}_1), \mathsf{nd}_x(\mathsf{ae}_2)) \\
 & & \\
\mathsf{nd}_x(\mathsf{tc}) & = & \mathsf{max}(\mathsf{nd}_x(\mathsf{te}_1), \mathsf{nd}_x(\mathsf{te}_2)) \quad \text{where } \mathsf{tc} = \mathsf{te}_1 \sqsubseteq \mathsf{te}_2 \\
\mathsf{nd}_x(\mathsf{ac}) & = & \mathsf{max}(\mathsf{nd}_x(\mathsf{ae}_1), \mathsf{nd}_x(\mathsf{ae}_2)) \quad \text{where } \mathsf{ac} = \mathsf{ae}_1 \leq \mathsf{ae}_2 \\
\mathsf{nd}_x(\mathcal{C}) & = & \max_{ij}(\mathsf{nd}_x(\mathsf{tc}_i), \mathsf{nd}_x(\mathsf{ac}_j)) \quad \text{where } \mathcal{C} = (\bigcup_i \mathsf{tc}_i, \bigcup_j \mathsf{ac}_j)
\end{array}
$$

Figure 6.2: Nesting depth of a variable x in a system of constraints $\mathcal{C}$.

Lemma 6.3.3 *Let $\mathcal{TC}$ be a set of constraints and let $(\mathcal{TC}', \mathcal{AC}') = \mathsf{unfold}(\mathcal{TC})$. Then $\mathcal{TC} \iff (\mathcal{TC}', \mathcal{AC}')$.*

Proof. Follows by the definition of inequality (Def. 6.2.2). □

In the following we define when variables occur positively or negatively in expressions.

Definition 6.3.4 ($\mathcal{C}(x^+), \mathcal{C}(x^-), \mathcal{C}(x^+, x^-)$)

1. *Let $\mathsf{tc} = \mathsf{te}_1 \sqsubseteq \mathsf{te}_2$ be an inequality among tree expressions.*
   (a) *We say that x* occurs positively in tc *if x occurs in te_2.*
   (b) *We say that x* occurs negatively in tc *if x occurs in te_1.*
2. *Let $\mathsf{ac} = \mathsf{ae}_1 \leq \mathsf{ae}_2$ be a linear arithmetic inequality.*
   (a) *We say that x* occurs positively in ac *if x occurs in ae_2.*
   (b) *We say that x* occurs negatively in ac *if x occurs in ae_1.*
3. *Let $\mathcal{C} = (\bigcup_i \mathsf{tc}_i, \bigcup_j \mathsf{ac}_j)$ be a system of constraints.*
   (a) *We say that x* occurs positively (negatively) in $\mathcal{C}$ *and write $\mathcal{C}(x^+)$ ($\mathcal{C}(x^-)$) if x occurs positively and does not occur negatively (occurs negatively and does not occur positively) in all tc_i and ac_j.*
   (b) *We say that x* occurs positively and negatively in $\mathcal{C}$ *and write $\mathcal{C}(x^+, x^-)$ if x occurs both positively and negatively in tc_i and ac_j.*

Lemma 6.3.5 *Let $t, \hat{t} \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ with $t \sqsubseteq \hat{t}$ and let π be a valuation and $x \in X$.*

1. *Let te be a tree expression and let $x \in \mathsf{Vars}(\mathsf{te}) \subseteq \mathrm{dom}(\pi)$. Then, $\pi[x \mapsto t](\mathsf{te}) \sqsubseteq \pi[x \mapsto \hat{t}](\mathsf{te})$.*

2. *Let ae be an arithmetic expression and let $x \in \mathsf{Vars}(\mathsf{ae}) \subseteq \mathrm{dom}(\pi)$. Then, $\pi[x \mapsto t](\mathsf{ae}) \sqsubseteq \pi[x \mapsto \hat{t}](\mathsf{ae})$.*

Proof.

1. By induction on te, using monotonicity of the function l, when $l \in L$, and $+$.

2. Follows by 1.

□

Lemma 6.3.6 *Let $\mathcal{C}(x^+)$ and $\mathcal{D}(x^-)$ be systems of constraints and $t, \hat{t} \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ with $t \sqsubseteq \hat{t}$. Then*

1. *If $\pi[x \mapsto t] \models \mathcal{C}(x^+)$ then $\pi[x \mapsto \hat{t}] \models \mathcal{C}$.*

2. *If $\pi[x \mapsto \hat{t}] \models \mathcal{D}(x^-)$ then $\pi[x \mapsto t] \models \mathcal{D}$.*

Proof.

1. Let $\mathcal{C} = (\mathcal{TC}, AC)$ and let $\mathsf{tc} \in \mathcal{TC}$. We have $\pi[x \mapsto t] \models \mathsf{te}_1 \sqsubseteq \mathsf{te}_2(x)$, i.e. $\pi[x \mapsto t](\mathsf{te}_1) \sqsubseteq \pi[x \mapsto t](\mathsf{te}_2(x))$. Moreover, by Lemma 6.3.5, we obtain $\pi[x \mapsto t](\mathsf{te}_2) \sqsubseteq \pi[x \mapsto \hat{t}](\mathsf{te}_2(x))$. Thus, by transitivity, we obtain the desired $\pi[x \mapsto \hat{t}] \models \mathsf{te}_1 \sqsubseteq \mathsf{te}_2(x)$.

   Let moreover $\mathsf{ac} = \mathsf{ae}_1 \sqsubseteq \mathsf{ae}_2(x) \in \mathcal{AC}$. Then, $\pi[x \mapsto \hat{t}] \models \mathsf{ae}_1 \sqsubseteq \mathsf{ae}_2(x)$ follows similarly.

2. Similar to the previous case.

□

Lemma 6.3.7 *Let $\mathcal{C}$ be a system of constraints and $x \in X$ and $\mathsf{te}_1, \ldots, \mathsf{te}_n \in \mathsf{TExp}$ and $\mathsf{te}_1, \ldots, \mathsf{te}_m \in \mathsf{TExp}$ and π be a valuation. Then*

1. *If $\mathcal{C}(x^+)$ and $\pi \models \mathcal{C}\,[\mathsf{te}_i/x]$ then $\pi \cup \{x \mapsto \bigwedge\{\pi(\mathsf{te}_1), \ldots, \pi(\mathsf{te}_n)\}\} \models \mathcal{C}$.*

2. *If $\mathcal{C}(x^-)$ and $\pi \models \mathcal{C}\,[\mathsf{te}_i/x]$ then $\pi \cup \{x \mapsto \bigvee\{\pi(\mathsf{te}_1), \ldots, \pi(\mathsf{te}_m)\}\} \models \mathcal{C}$.*

Proof.

1. Let $\mathcal{C} = (\mathcal{TC}, AC)$ and let $\mathsf{te} \sqsubseteq \mathsf{te}'(x) \in \mathcal{TC}$ and let $\pi(\mathsf{te}_i) = t_i$ and let te' stand for the function f, where f is possibly a composition of the functions l for some $l \in \mathcal{L}$ or $+$ or the identity function and let $\pi \cup \{x \mapsto \bigwedge\{t_1, \ldots, t_n\}\} = \hat{\pi}$. We have $\pi(\mathsf{te}) \sqsubseteq f(t_i)$, then:

$$\begin{array}{rll} \pi(\mathsf{te}) & \sqsubseteq & f(t_i) \iff \text{g.l.b.} \\ & \sqsubseteq & \bigwedge\{f(t_1), \ldots, f(t_n)\} \iff \text{Lemma 6.2.9} \\ & \sqsubseteq & f(\bigwedge\{t_1, \ldots, t_n\}) \iff \\ & \sqsubseteq & \hat{\pi}(f(x)) \end{array}$$

   Moreover let let $\mathsf{ae} \sqsubseteq \mathsf{ae}'(x) \in \mathcal{AC}$. Then $\hat{\pi} \models \mathsf{ae} \sqsubseteq \mathsf{ae}'$ follows similarly.

2. Similar to the previous case, using the l.u.b. property and Lemma 6.2.9.

□

6.3.1 Algorithmic problems

In this section we discuss algorithmic problems regarding a system of constraints $\mathcal{C}$ whose study would be of interest.

Satisfiability.

One important problem, with a direct application to type inference for the RAJA typing system (Chapter 5), is satisfiability. That is, if we have given a system of constraints, we would like to know whether it is satisfiable. Moreover, we would like to obtain a valuation π that satisfies the constraints. Here we give a slightly weaker definition of the satisfiability problem. We are interested in a *finite* set of arithmetic constraints that is satisfiable iff the system of constraints $\mathcal{C}$ is satisfiable. Since the trees we are studying are infinite, it is not possible to obtain a valuation $\pi_t : X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ in general. However, we will see in Section 6.5 that we can effectively deliver a valuation π_t when all the values in $\mathrm{ran}(\pi_t)$ are regular trees.

Reducing the satisfiability problem to the problem of satisfying a finite set of arithmetic constraints is advantageous because there are effective ways of solving linear arithmetic constraints. Moreover, we remark that the problem of obtaining an *infinite* set of arithmetic constraints equivalent to $\mathcal{C}$ is trivial. If we follow the definition of inequality ($\sqsubseteq_{\mathsf{ind}}$) we notice that a set of inequalities over trees $\mathcal{TC} = \bigcup_i \mathsf{te}_i \sqsubseteq \mathsf{te}'_i$ is satisfiable iff the following set of arithmetic constraints is satisfiable: $\Gamma(\mathcal{TC}) = \{\mathsf{te}_{iw} \leq \mathsf{te}'_{iw} \mid w \in \mathcal{L}^*\}$. In Section 6.5 we provide an algorithm for solving satisfiability that is sound in all cases and complete for constraints systems of a restricted form.

Example 6.3.8 *Let $\mathcal{L} = \{l\}$ and $\mathcal{TC} = \{x \sqsubseteq l(x), l(x) + l(x) \sqsubseteq z\}$ and $\mathcal{AC} = \{1 \leq \Diamond(x)\}$. The set $\mathcal{AC}' = \{1 \leq \lambda, \lambda + \lambda \leq \delta\}$ is equivalent to $(\mathcal{TC}, \mathcal{AC})$. This example can be analysed by our algorithm.*

Satisfiability

Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$.

Wanted: A finite set of linear arithmetic constraints $\mathcal{AC}'$ such that: there is π_a with $\pi_a \models \mathcal{AC}'$ iff there is π_t such that $(\pi_t, \pi_a) \models \mathcal{C}$.

Optimisation

Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and a linear objective function f defined on the arithmetic variables.

Wanted: A valuation π_a of the arithmetic variables such that $(\pi_t, \pi_a) \models \mathcal{C}$ for some valuation of the tree variables and whenever $(\pi'_t, \pi'_a) \models \mathcal{C}$ then $f(\pi'_a) \leq f(\pi_a)$.

Elimination of a tree variable

Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and a variable $x \in X$.

Wanted: A finite system of constraints $\mathcal{C}'$ with $x \notin \mathsf{Vars}(\mathcal{C}') \subseteq \mathsf{Vars}(\mathcal{C})$ and $\pi \models \mathcal{C}'$ iff $\exists t.\pi[x \mapsto t] \models \mathcal{C}$.

Figure 6.3: Algorithmic problems regarding systems of constraints.

Elimination of a tree variable.

The problem of eliminating a variable x from a system of constraints $\mathcal{C}$ while keeping the satisfiability of the constraints (Fig. 6.3) is interesting for various reasons. The first one is efficiency. Eliminating variables can reduce significantly the size of a system of constraints. Thus, it is a good idea to eliminate some variables first, and then try to solve the resulting system. On the other hand, eliminating variables can help in bringing constraints in a form that is particularly suitable for applying an algorithm (see Section 6.5.3). In Section 6.4 we give an algorithm for variable elimination that, however, does not succeed in eliminating all variables. If we had an algorithm that solved the elimination problem, the algorithm would solve satisfiability as well, since a finite system of constraints without tree variables is automatically a finite set of arithmetic constraints.

Example 6.3.9 *Assume we want to eliminate y from*

$$\mathcal{C} = \{x \sqsubseteq y, y \sqsubseteq l(x)\}, \{1 \leq \Diamond\langle(y)\}$$

Then our algorithm would return

$$\mathcal{C}' = \{x \sqsubseteq l(x)\}, \{1 \leq \Diamond\langle(l(x))\}$$

which is equivalent to $\mathcal{C}$. However, our algorithm is not able to eliminate x from $\mathcal{C}'$.

6.4 Elimination of tree variables

In this section we define an algorithm for eliminating tree variables from a set of tree constraints while keeping the satisfiability of the constraints. This algorithm is partially inspired by the Fourier-Motzkin elimination procedure for eliminating variables from a system of linear inequalities [Mot36, DE73].

Concretely, we define the algorithm elim, that takes a system of constraints $\mathcal{C}$ and a variable y, and returns a system of constraints $\mathcal{C}'$, which does not contain y, and that is satisfiable iff $\mathcal{C}$ is satisfiable. We present the judgement $\mathcal{C} \to_y \mathcal{C}'$ as a set of inference rules (Fig. 6.4) and define the elimination function elim by:

$$\mathsf{elim}_y(\mathcal{C}) = \begin{cases} \mathcal{C}' & \text{if } \mathcal{C} \to_y \mathcal{C}' \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Depending on whether the variable appears only positively or only negatively in the rules, or whether it appears both positively and negatively, we choose one of the rules.

The rule ($\triangleright\, \mathsf{Prune}$) is used when the variable appears either only positively or only negatively in the constraints. In that case, the variable can be removed safely from the system of constraints. If the variable appears in a constraint such as $\mathsf{tae}_1(y) + \mathsf{tae}_2 \sqsubseteq \mathsf{tae}$, then we return $\mathsf{tae}_2 \sqsubseteq \mathsf{tae}$. Otherwise, when it appears in a constraint $\mathsf{tae}_1 \sqsubseteq \mathsf{tae}_2$, then we remove the whole constraint. Further, if the variable appears in an arithmetic constraint $\Diamond\!\!\!\!-(\mathsf{tae}(y)) + \mathsf{ae}_2 \leq \mathsf{ae}$, we return $\mathsf{ae}_2 \leq \mathsf{ae}$.

If the variable has at least one upper bound and appears otherwise only positively, we use the rule ($\triangleright\, \mathsf{Elim}^+$). With that rule, the elimination takes place by substituting the variable in the constraints with its upper bounds. The rule ($\triangleright\, \mathsf{Elim}^-$) is analogue; we use it when the variable has at least one lower bound and appears otherwise only negatively.

The last and more complicated rule is ($\triangleright\, \mathsf{Elim}^{+/-}$). We use it when the variable appears both positively and negatively. In that case, we calculate $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$. Recall that $\mathcal{C}(y^{\mathsf{whole}})$ is the subset of the constraints where the variable appears as a variable expression or sub-expression. Moreover, $\mathcal{C}(y^{\mathsf{proj}})$ is the subset of the constraints where the variable appears in expressions or sub-expressions $l(\mathsf{tae})$ or $\Diamond\!\!\!\!-(\mathsf{tae})$.

If $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$ are disjoint sets and $\mathsf{nd}_y(\mathcal{C}) > 0$, we unfold $\mathcal{C}(y^{\mathsf{whole}})$ and substitute all the occurrences of $l_i(y)$ and $\Diamond\!\!\!\!-(y)$ with fresh variables z_i and λ, respectively. Finally, we eliminate recursively all the new introduced tree variables z_i.

If, on the other hand, $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$ are not disjoint sets, the variable cannot be eliminated. This restriction is necessary to ensure termination, as we shall see in Theorem 6.4.2. For an example of an elimination problem that would not terminate, suppose we have the constraint $l(x) \sqsubseteq x$ and $\mathcal{L} = \{l\}$ and we want to eliminate the variable x. If we applied the rule

Eliminating y from $\mathcal{C}$ $\quad \boxed{\mathsf{elim}_y(\mathcal{C})}$

$$\frac{\mathcal{C}(y^+) \text{ or } \mathcal{C}(y^-) \qquad \mathcal{C}' = \text{ erase } y \text{ from } \mathcal{C}}{\mathcal{C} \to_y \mathcal{C}'} \ (\triangleright\,\mathsf{Prune})$$

$$\frac{(\bigcup_{i=1..n}\{y \sqsubseteq \mathsf{te}_i\}) \cup \mathcal{D}(y^+), \mathcal{AC}(y^+)}{\mathcal{C} \to_y \bigcup_{i=1..n} (\mathcal{D}, \mathcal{AC})\,[\mathsf{te}_i/y]} \ (\triangleright\,\mathsf{Elim}^+)$$

$$\frac{(\bigcup_{i=1..n}\{\mathsf{te}_i \sqsubseteq y\}) \cup \mathcal{D}(y^-), \mathcal{AC}(y^-)}{\mathcal{C} \to_y \bigcup_{i=1..n} (\mathcal{D}, \mathcal{AC})\,[\mathsf{te}_i/y]} \ (\triangleright\,\mathsf{Elim}^-)$$

$$\frac{\begin{array}{c}\mathcal{C}(y^+, y^-) \qquad \mathcal{C}(y^{\mathsf{proj}}) \cap \mathcal{C}(y^{\mathsf{whole}}) = \emptyset \ \text{ and } \ \mathsf{nd}_y(\mathcal{C}) > 0 \\ l_i \in \mathcal{L}, \vec{z}, \lambda \text{ new} \quad \mathcal{C}' = \mathcal{C}(y^{\mathsf{proj}}) \cup \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}})) \quad \mathcal{C}'' = \mathcal{C}'\,[z_i/l_i(y)][\lambda/\Diamond(y)]\end{array}}{\mathcal{C} \to_y \mathsf{elim}_{\vec{z}}(\mathcal{C}'')} \ (\triangleright\,\mathsf{Elim}^{+/-})$$

Figure 6.4: Elimination of a tree variable from a set of tree constraints.

($\triangleright\,\mathsf{Elim}^{+/-}$), we would obtain $l(l(x)) \sqsubseteq l(x)$ after unfolding and $l(z) \sqsubseteq z$ after substituting $l(x)$ with a fresh variable z. If we now tried to eliminate z, we would go through the same procedure again and would not terminate.

Another condition for the application of the rule ($\triangleright\,\mathsf{Elim}^{+/-}$) is that $\mathsf{nd}_y(\mathcal{C}) > 0$. This is also necessary for termination. If $\mathsf{nd}_y(\mathcal{C}) = 0$, then applying ($\triangleright\,\mathsf{Elim}^{+/-}$) would not succeed in eliminating the variable. The reason for this is that this rule is useful only when different subtrees of the tree that the variable represents appear in different constraints, and so the unfolding and substitution causes that the different subtrees are treated as two independent variables, which often leads to the elimination of those variables. For instance, if we wish to eliminate the variable x in the constraints $l_1(x) \sqsubseteq y, w \sqsubseteq l_2(x)$, after the substitution we would obtain $z_1 \sqsubseteq y, w \sqsubseteq z_2$, and z_1 and z_2 can be eliminated with the rule ($\triangleright\,\mathsf{Prune}$).

Next, we prove that elim satisfies its specification:

$$\mathsf{elim}_y(\mathcal{C}) = \mathcal{C}' \ \text{ implies } \ \mathcal{C}' \iff \exists y\,.\,\mathcal{C}$$

It is important to remark that, given a valuation π for $\mathcal{C}'$, the proof provides a value t for y, based on the values from π, such that $\pi \cup \{y \mapsto t\}$ satisfies $\mathcal{C}$.

Theorem 6.4.1 (Correctness of elim)
Let $\mathcal{C}(\vec{x}, y, \vec{\lambda})$ be a system of constraints. If $\mathsf{elim}_y(\mathcal{C}) = \mathcal{C}'(\vec{x}, \vec{\lambda})$ then for all $\pi : \pi \models \mathcal{C}' \iff$ there exists t with $\pi \cup \{y \mapsto t\} \models \mathcal{C}$.

Proof. By induction on the recursive definition of elim.

Case ($\triangleright\mathsf{Prune}$) If $\mathcal{C}(v^+)$. The goal follows from the fact that we can add the constraint $y \sqsubseteq \widehat{\infty}$ to the system of constraints. Then we could apply case $\mathcal{C} = y \sqsubseteq \widehat{\infty} \cup \mathcal{D}(y^+), \mathcal{AC}(y^+)$ and obtain $(\mathcal{D}, \mathcal{AC})\,[\widehat{\infty}/y]$. Then we could remove again the resulting constraints $\mathsf{tae} \sqsubseteq \widehat{\infty}$ and remove $\widehat{\infty}$ otherwise from compound expressions in the constraints. If $\mathcal{C}(v^-)$. We could add $\widehat{0} \sqsubseteq y$ to the system of constraints and proceed similarly to the previous case.

Case ($\triangleright\,\mathsf{Elim}^+$)

Case "$\Longleftarrow$" We have $\pi' = \pi \cup \{y \mapsto t\}$ and $t \sqsubseteq \pi(\mathsf{te}_i)$ and $\pi \models (\mathcal{D}, \mathcal{AC})$. By Lemma 6.3.6 we get $\pi[y \mapsto \pi(\mathsf{te}_i)] \models (\mathcal{D}, \mathcal{AC})$ and this implies $\pi \models (\mathcal{D}, \mathcal{AC})[\mathsf{te}_i/x]$.

Case "$\Longrightarrow$" Let $\pi \models \mathcal{C}'$ and let $\hat{\pi} = \pi \cup \{y \mapsto t\}$ where

$$t := \bigwedge\{\pi(\mathsf{te}_1), \ldots, \pi(\mathsf{te}_n)\}$$

Then $t \sqsubseteq \pi(\mathsf{te}_i)$ by the g.l.b. property and $\hat{\pi} \models \mathcal{D}$ follows by Lemma 6.3.7.

Case ($\triangleright\mathsf{Elim}^-$) The proof is dual to the previous case. In the direction"$\Longleftarrow$" we use Lemma 6.3.6 and in the direction"$\Longrightarrow$" we set

$$\hat{\pi}(y) := \bigvee\{\pi(\mathsf{te}_1), \ldots, \pi(\mathsf{te}_n)\}$$

and $\pi(\mathsf{te}_i) \sqsubseteq t$ follows by the l.u.b. property and $\hat{\pi} \models \mathcal{D}$ follows by Lemma 6.3.7.

Case ($\triangleright\,\mathsf{Elim}^{+/-}$)

Case "$\Longleftarrow$" Let $\overbrace{\pi \cup \{y \mapsto t\}}^{\hat{\pi}} \models \mathcal{C}$. By Lemma 6.3.3 we get $\hat{\pi} \models \mathcal{C}'$. Moreover, set $\bar{\pi} = \pi \cup \{z_i \mapsto l_i(t), \lambda \mapsto \Diamond(t)\}$. Then $\bar{\pi} \models \mathcal{C}''$ follows trivially. Finally, by induction hypothesis we get a valuation π' with $\pi' \models \mathsf{elim}_{\vec{z}}(\mathcal{C}'')$.

Case "$\Longrightarrow$" Let $\pi \models \mathsf{elim}_{\vec{z}}(\mathcal{C}'')$. By induction hypothesis we obtain $\pi \cup \{z_i \mapsto t_i\} \models \mathcal{C}''$. Let moreover $\hat{\pi} = \pi \cup \{x \mapsto \hat{t}\}$ where $\hat{t}$ is defined by $l_i(\hat{t}) = t_i$ and $\Diamond(\hat{t}) = \pi(\lambda)$. Then $\hat{\pi} \models \mathcal{C}'$ follows trivially. Finally, $\hat{\pi} \models \mathcal{C}$ follows by Lemma 6.3.3.

□

Termination of the elimination of tree variables

Since the algorithm elim is recursive, its termination is not straight forward. Indeed, for proving termination we need a measure that decreases in each recursive call. We will see that the nesting depth of expressions containing the variable to be eliminated is the appropriate measure.

Theorem 6.4.2 *The function* $\mathsf{elim}_y(\mathcal{C})$ *always terminates.*

Proof. We need to find a measure that decreases in each iteration. We can use the maximal nesting depth of the expressions containing the variable y in $\mathcal{C}$ ($\mathsf{nd}_y(\mathcal{C})$). This measure is useful because if $\mathsf{nd}_y(\mathcal{C}) = 0$ then the rule ($\triangleright\,\mathsf{Elim}^{+/-}$) cannot be applied and the algorithm terminates.

The only interesting case is ($\triangleright\,\mathsf{Elim}^{+/-}$). We show then

$$\mathsf{nd}_{z_i}(\mathcal{C}'') < \mathsf{nd}_y(\mathcal{C}) \quad \text{or} \quad \mathsf{nd}_{z_i}(\mathcal{C}'') = 0 \tag{6.4.1}$$

Let $\mathsf{tc} = \mathsf{te} \sqsubseteq \mathsf{te}'$ be a constraint in $\mathcal{C}''$. Let us assume w.l.o.g. $z_i \in \mathsf{Vars}(\mathsf{te})$. Then $\hat{\mathsf{tc}} = (\mathsf{te}[l_i(y)/z_i] \sqsubseteq \mathsf{te}') \in \mathcal{C}(y^{\mathsf{proj}}) \cup \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}}))$. Then either:

Case $\hat{\mathsf{tc}} \in \mathcal{C}(y^{\mathsf{proj}})$. Then $\mathsf{nd}_{z_i}(\mathsf{te}) = \mathsf{nd}_y(\mathsf{te}[l_i(y)/z_i]) - 1$.

Case $\hat{\mathsf{tc}} \in \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}}))$. Then, by the side condition $\mathcal{C}(y^{\mathsf{proj}}) \cap \mathcal{C}(y^{\mathsf{whole}}) = \emptyset$, $\mathsf{nd}_y(\mathsf{te}[l_i(y)/z_i]) = 1$ and $\mathsf{nd}_{z_i}(\mathsf{te}) = 0$.

Next, let $\mathsf{ac} = \mathsf{ae}_1 \leq \mathsf{ae}_2$ be a linear arithmetic constraint in $\mathcal{C}''$. Let us assume w.l.o.g. $z_i \in \mathsf{Vars}(\mathsf{ae}_1)$. Then $\hat{\mathsf{ac}} = (\mathsf{ae}_1[l_i(y)/z_i] \leq \mathsf{ae}_2) \in \mathcal{C}(y^{\mathsf{proj}}) \cup \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}}))$. Then we notice that $\hat{\mathsf{ac}} \in \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}}))$ can not happen by the side condition. Hence, $\hat{\mathsf{ac}} \in \mathcal{C}(y^{\mathsf{proj}})$ and, like in the previous case, $\mathsf{nd}_{z_i}(\mathsf{ae}_1) = \mathsf{nd}_y(\mathsf{ae}_1[l_i(y)/z_i]) - 1$. Since tc and ac were arbitrary constraints in $\mathcal{C}'$, (6.4.1) follows. □

6.5 Solving a system of constraints

In this section we present an algorithm for solving a system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$. The linear arithmetic constraints $\mathcal{AC}$ can be easily solved by an LP-Solver. Thus, the challenge is to deal with constraints over trees. Our goal is to reduce the problem of solving these constraints to the problem of solving a finite set of linear arithmetic constraints.

We noticed in Section 6.3.1 that the canonical set

$$\Gamma(\mathcal{TC}) = \{\mathsf{te}_w \leq \mathsf{te}'_w \mid \mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}, w \in \mathcal{L}^*\}$$

is infinite. But in some particular cases when the constraints admit regular solutions, we can obtain a finite set of arithmetic constraints. Our algorithm seeks solutions to the constraints in the case that the trees must also satisfy some (given) regular structure. When the algorithm is given a regular structure for the tree variables that occur in $\mathcal{TC}$, that we call a *tree schema* Ts, it calculates a finite set of arithmetic constraints $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$. We prove that the algorithm is sound, i.e. if there is a solution for $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$, there is also a solution for $\mathcal{TC}$ and the algorithm delivers it. Clearly, the algorithm is not complete in the general case since not all constraints admit a regular

solution. We also give an upper bound on the size of the resulting set of arithmetic constraints in terms of the sizes of $\mathcal{TC}$ and $\mathsf{Vars}(\mathcal{TC})$.

Tree constraints in normal form. We say that tree expressions are in *normal form* when they are either atomic or a compound expression of the restricted form: $\mathsf{tae} + \mathsf{tae}'$. Moreover, we say that a tree constraint $\mathsf{tc} = \mathsf{te}_1 \sqsubseteq \mathsf{te}_2$ is in normal form if te_1 and te_2 are in normal form and only one of them is compound. Arbitrary tree constraints $\mathsf{tc} \in \mathsf{TConstr}$ can be brought into this form by introducing new variables, for example the tree constraint $x \sqsubseteq y + z + w$ is equivalent to $\{x \sqsubseteq y + v, v = z + w\}$. In the following section we assume that the tree constraints are in normal form. This will simplify our computation of $|\Gamma_{\mathsf{Ts}}(\mathcal{TC})|$ because we will be able to use the fact that $|\mathsf{Vars}(\mathsf{tc})| \leq 3$ for each constraint tc.

6.5.1 Tree schema substitution and $\Delta_{\mathsf{Ts}}(\mathcal{C})$

In the following we define tree schemas: a finite set of tree variables, a finite set of regular trees and a pair of maps, which represent a regular structure for a set of infinite trees.

Definition 6.5.1 (Tree schema) *A* tree schema Ts *consists of*

- *a finite subset* $\mathsf{Ts}.X \subseteq X$.
- *a finite subset* $\mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \subseteq \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ *closed under* $l(.)$ *for every* $l \in \mathcal{L}$.
- *a total map* $\mathsf{Ts}.\mathsf{next} : \mathcal{L} \times \mathsf{Ts}.X \to \mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$.
- *a total injective map* $\mathsf{Ts}.\Diamond : \mathsf{Ts}.X \to \Lambda$.

A valuation $\pi = (\pi_t, \pi_a)$ matches *tree schema* Ts *if the following conditions hold for every* $x \in \mathsf{Ts}.X$*:*

- *if* $\mathsf{Ts}.\Diamond(x) = \lambda \in \Lambda$ *then* $\Diamond(\pi_t(x)) = \pi_a(\lambda)$;
- *if* $\mathsf{Ts}.\mathsf{next}(l, x) = y \in \mathsf{Ts}.X$ *then* $l(\pi_t(x)) = \pi_t(y)$.
- *if* $\mathsf{Ts}.\mathsf{next}(l, x) = t \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ *then* $l(\pi_t(x)) = t$.

Example 6.5.2 (Tree schema)
Assume $x_1, x_2 \in X$ *and* $\lambda_1, \lambda_2 \in \Lambda$ *and* $\mathcal{L} = \{l\}$.

Let Ts *be a tree schema defined by* $\mathsf{Ts}.X = \{x_1, x_2\}$ *and* $\mathsf{Ts}.\Diamond(x_i) = \lambda_i$ *for* $i \in \{1, 2\}$ *and* $\mathsf{Ts}.\mathsf{next}(l, x_1) = x_2$ *and* $\mathsf{Ts}.\mathsf{next}(l, x_2) = x_1$.

Now define the trees t_1 *and* t_2 *by* $\Diamond(t_1) = 1$, $l(t_1) = t_2$ *and* $\Diamond(t_2) = 2$, $l(t_2) = t_1$. *The valuation* π *given by* $\pi_t(x_i) = t_i$ *and* $\pi_a(\lambda_i) = i$ *matches* Ts *(see Fig. 6.5).*

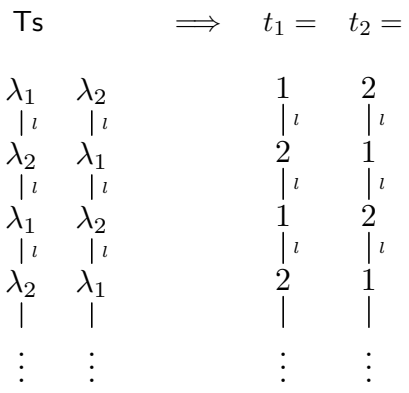


Figure 6.5: Representation of a tree schema Ts and a matching valuation.

The reason why the set $\Gamma(\mathcal{TC})$ is infinite is that it contains expressions te_w for each $w \in L^*$. The main advantage of having a tree schema is that we can eliminate expressions containing labels (like $x_{1ll} = l(l(x_1))$) from a set of constraints. The substitution of such expressions with tree schemas delivers a variable or a constant. In this case $l(l(x_1))[\mathsf{Ts}]$ delivers x_1 because $\mathsf{Ts}.\,\mathsf{next}(l, x_1) = x_2$ and $\mathsf{Ts}.\,\mathsf{next}(l, x_2) = x_1$. We define the functions $\mathsf{tae}[\mathsf{Ts}] : X \cup \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, $\mathsf{te}[\mathsf{Ts}] : \mathsf{TExp}$ and $\mathsf{ae}[\mathsf{Ts}] : \mathsf{AExp}$ formally in Fig. 6.6. These functions simplify the given expressions with respect to a particular tree schema so that $\mathcal{TC}[\mathsf{Ts}]$ returns a set of constraints over trees with no (sub)expressions of the form $l(\mathsf{tae})$, while $\mathcal{AC}[\mathsf{Ts}]$ returns a set of arithmetic constraints that contains no tree expressions.

In Fig. 6.6 we also define the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$, a set of arithmetic constraints whose satisfiability implies satisfiability of $\mathcal{TC}$. We build the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as follows: for each constraint $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$ and each path $w \in \mathcal{L}^*$, we add the arithmetic constraints $\mathsf{te}_w[\mathsf{Ts}] \leq \mathsf{te}'_w[\mathsf{Ts}]$ to the set. The use of tree schema substitution ensures that $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ is finite, in contrast to $\Gamma(\mathcal{TC})$. In the following Lemma we compute an upper bound on the size of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as a function of the sizes of $\mathcal{TC}$ and $\mathsf{Vars}(\mathcal{TC})$.

Lemma 6.5.3 (Cardinality of the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$) *Let $\mathcal{TC}$ be a set of constraints and Ts a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$. Then $|\Gamma_{\mathsf{Ts}}(\mathcal{TC})| \leq |\mathcal{TC}| \cdot |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}|^3$.*

Proof. First, we show, for each $\mathsf{tae} \in \mathsf{TAExp}$:

$$|\{\mathsf{tae}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| \leq |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}| \tag{6.5.1}$$

This follows from the fact that

$$\{\mathsf{tae}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\} \subseteq \mathrm{ran}(\mathsf{Ts}.\langle\!\rangle()) \cup \mathrm{ran}(\langle\!\rangle()|_{\mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}})$$

Tree schema substitution

$\boxed{\mathsf{te}[\mathsf{Ts}]}$

$$
\begin{array}{lcl}
x[\mathsf{Ts}] & = & x \\
l(\mathsf{tae})[\mathsf{Ts}] & = & \begin{cases} \mathsf{Ts}.\mathsf{next}(l, y) & \text{if } \mathsf{tae}[\mathsf{Ts}] = y \in \mathsf{Ts}.X \\ l(t) & \text{if } \mathsf{tae}[\mathsf{Ts}] = t \in \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \end{cases} \\
(\mathsf{te}_1 + \mathsf{te}_2)[\mathsf{Ts}] & = & \mathsf{te}_1[\mathsf{Ts}] + \mathsf{te}_2[\mathsf{Ts}]
\end{array}
$$

$\boxed{\mathsf{ae}[\mathsf{Ts}]}$

$$
\begin{array}{lcl}
n[\mathsf{Ts}] & = & n \\
\lambda[\mathsf{Ts}] & = & \lambda \\
\langle\!(\mathsf{tae})\!\rangle[\mathsf{Ts}] & = & \begin{cases} \mathsf{Ts}.\langle\!(\mathsf{tae}[\mathsf{Ts}])\!\rangle & \text{if } \mathsf{tae}[\mathsf{Ts}] = y \in \mathsf{Ts}.X \\ \langle\!(t)\!\rangle & \text{if } \mathsf{tae}[\mathsf{Ts}] = t \in \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \end{cases} \\
(\mathsf{ae}_1 + \mathsf{ae}_2)[\mathsf{Ts}] & = & \mathsf{ae}_1[\mathsf{Ts}] + \mathsf{ae}_2[\mathsf{Ts}] \\
\mathcal{TC}[\mathsf{Ts}] & = & \bigcup_i \{\mathsf{te}_i[\mathsf{Ts}] \sqsubseteq \mathsf{te}'_i[\mathsf{Ts}]\} \quad \text{for } \mathcal{TC} = \bigcup_i \{\mathsf{te}_i \sqsubseteq \mathsf{te}'_i\} \\
\mathcal{AC}[\mathsf{Ts}] & = & \bigcup_i \{\mathsf{ae}_i[\mathsf{Ts}] \leq \mathsf{ae}'_i[\mathsf{Ts}]\} \quad \text{for } \mathcal{AC} = \bigcup_i \{\mathsf{ae}_i \leq \mathsf{ae}'_i\} \\
\Gamma_{\mathsf{Ts}}(\mathcal{TC}) & = & \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \{\mathsf{te}_w[\mathsf{Ts}] \leq \mathsf{te}'_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\} \\
\Delta_{\mathsf{Ts}}(\mathcal{C}) & = & \Gamma_{\mathsf{Ts}}(\mathcal{TC}) \cup \mathcal{AC}[\mathsf{Ts}]
\end{array}
$$

Figure 6.6: Tree schema substitution and $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$.

Then,

$$
\begin{array}{rcl}
|\{\mathsf{tae}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| & \leq & |\operatorname{ran}(\mathsf{Ts}.\langle\!(\,)\!\rangle) \cup \operatorname{ran}(\langle\!(\,)\!\rangle|_{\mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}})| \\
& \leq & |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}|
\end{array}
$$

Next, we show, for each $\mathsf{te} \in \mathsf{TExp}$ in normal form:

$$|\{\mathsf{te}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| \leq |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}|^2 \tag{6.5.2}$$

Case $\mathsf{te} = \mathsf{tae}$. It follows from (6.5.1).

Case $\mathsf{te} = \mathsf{tae} + \mathsf{tae}'$. First we notice that

$$
\begin{array}{rcl}
\{\mathsf{te}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\} & = & \{\mathsf{tae}_w[\mathsf{Ts}] + \mathsf{tae}'_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\} \text{ Thus,} \\
|\{\mathsf{te}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| & \leq & |\{\mathsf{tae}_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| \times |\{\mathsf{tae}'_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| \\
& \leq & |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}|^2 \qquad \text{by (6.5.1)}
\end{array}
$$

Now, let $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$ with either $\mathsf{te} \in \mathsf{TAExp}$ or $\mathsf{te}' \in \mathsf{TAExp}$. We show, similarly to the previous case,

$$|\{\mathsf{te}_w[\mathsf{Ts}] \sqsubseteq \mathsf{te}'_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*\}| \leq |\mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}|^3 \tag{6.5.3}$$

and the goal follows trivially.

□

We would like to remark that we have given an upper bound to $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$, but that this set is much smaller in practice.

The set of arithmetic constraints $\Delta_{\mathsf{Ts}}(\mathcal{C})$, also defined in Fig. 6.6, is obtained by adding the constraints in $\mathcal{AC}$, after their substitution with the tree schema Ts, to the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$. Thus, $\Delta_{\mathsf{Ts}}(\mathcal{C})$ is a finite set of arithmetic constraints without tree variables. We will show below that satisfiability of $\Delta_{\mathsf{Ts}}(\mathcal{C})$ implies satisfiability of $\mathcal{C}$.

Example 6.5.4 *Let $X, \Lambda, \mathcal{L}$ and Ts be defined as in Example 6.5.2. Moreover, let $\mathcal{C} = \{l(x_1) \sqsubseteq x_2, l(x_2) \sqsubseteq x_1\}, \{1 \leq \Diamond(x_1), 2 \leq \Diamond(x_2)\}$. Then*

$$\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \{ \begin{array}{lcl} \lambda_2 & \leq & \lambda_2, \\ \lambda_1 & \leq & \lambda_1 \end{array} \} \qquad \Delta_{\mathsf{Ts}}(\mathcal{C}) = \{ \begin{array}{lcl} \lambda_2 & \leq & \lambda_2, \\ \lambda_1 & \leq & \lambda_1, \\ 1 & \leq & \lambda_1, \\ 2 & \leq & \lambda_2 \end{array} \}$$

Our next goal is to show the soundness of the algorithm for computing $\Delta_{\mathsf{Ts}}(\mathcal{C})$: if we have a solution for $\Delta_{\mathsf{Ts}}(\mathcal{C})$, we can also find a solution for $\mathcal{C}$. This result is based on the next Lemma, which states the following: given a tree schema Ts and a valuation $\pi = (\pi_t, \pi_a)$ that matches Ts, π satisfies $\mathcal{C}$ iff π_a satisfies $\Delta_{\mathsf{Ts}}(\mathcal{TC})$. Moreover we notice that all the trees in $\mathrm{ran}(\pi_t)$ are regular.

Lemma 6.5.5 *Let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ be a system of constraints and Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and $\pi = (\pi_t, \pi_a)$ be a valuation that matches Ts.*

1. *Let $\mathsf{te} \in \mathsf{TExp}$ and $w \in \mathcal{L}^*$. Then $\pi_a(\mathsf{te}_w[\mathsf{Ts}]) = (\pi(\mathsf{te}))(w)$.*
2. *Let $\mathsf{ae} \in \mathsf{AExp}$. Then $\pi_a(\mathsf{ae}[\mathsf{Ts}]) = \pi(\mathsf{ae})$.*
3. *$\pi_a \models \Gamma_{\mathsf{Ts}}(\mathcal{TC}) \iff \pi \models \mathcal{TC}$.*
4. *$\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C}) \iff \pi \models \mathcal{C}$.*

Proof.

1. By induction on te, with an inner induction on w.
2. By induction on ae.
3. Let $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$. We have

$$\begin{array}{lcll} \pi \models \mathsf{te} \sqsubseteq \mathsf{te}' & \iff & & \forall w \in \mathcal{L}^* \,.\, (\pi(\mathsf{te}))(w) \leq (\pi(\mathsf{te}'))(w) \\ & \iff & {\scriptstyle 1.} & \forall w \in \mathcal{L}^* \,.\, \pi_a(\mathsf{te}_w[\mathsf{Ts}]) \leq \pi_a(\mathsf{te}'_w[\mathsf{Ts}]) \\ & \iff & & \pi_a \models \Gamma_{\mathsf{Ts}}(\mathsf{te} \sqsubseteq \mathsf{te}') \end{array}$$

4. Follows by 2. and 3.

□

Extending tree schema to a valuation $\boxed{\mathsf{Ts}[\pi_a] : X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}}$

$$
\begin{array}{lcll}
\mathsf{subst}(x, \mathsf{Ts}, \pi_a) = t & & & \\
\quad \text{where } \langle\!\langle t \rangle\!\rangle & = & \pi_a(\mathsf{Ts}.\langle\!\langle x \rangle\!\rangle) & \\
\quad \text{and } l(t) & = & \begin{cases} \mathsf{subst}(y, \mathsf{Ts}, \pi_a) \ \text{ for each } l \in \mathcal{L} & \text{if } \ \mathsf{Ts.next}(l, x) = y \in \mathsf{Ts}.X \\ t' & \text{if } \ \mathsf{Ts.next}(l, x) = t' \in \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \end{cases} & \\
\mathsf{Ts}[\pi_a] & = & \{x \mapsto \mathsf{subst}(x, \mathsf{Ts}, \pi_a) \mid x \in \mathsf{Ts}.X\} &
\end{array}
$$

Figure 6.7: Extending tree schema to a valuation.

Given a tree schema Ts and a valuation π_a that satisfies $\Delta_{\mathsf{Ts}}(\mathcal{C})$, we can build a valuation $\mathsf{Ts}[\pi_a] : \mathsf{Ts}.X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, as shown in Fig. 6.7, such that the valuation $(\mathsf{Ts}[\pi_a], \pi_a)$ matches Ts. Thus, by Lemma 6.5.5, $(\mathsf{Ts}[\pi_a], \pi_a)$ satisfies $\mathcal{C}$.

Theorem 6.5.6 (Soundness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$)
Let Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and $\pi_a : \Lambda \to \mathbb{D}$ be a valuation with $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$. Then there exists a valuation $\pi_t : \mathsf{Ts}.X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ such that $(\pi_t, \pi_a) \models \mathcal{C}$ and if $t \in \mathrm{ran}(\pi_t)$ then t is regular.

Proof. Since $(\mathsf{Ts}[\pi_a], \pi_a)$ matches Ts, the goal $(\mathsf{Ts}[\pi_a], \pi_a) \models \mathcal{C}$ follows by Lemma 6.5.5, item 4. Thus, $(\mathsf{Ts}[\pi_a], \pi_a)$ is the desired valuation. □

Lemma 6.5.5 also provides a sufficient condition on $\mathcal{C}$ which guarantees that its satisfiability implies satisfiability of $\Delta_{\mathsf{Ts}}(\mathcal{C})$. If it is possible to construct a tree schema such that there is a satisfying valuation for $\mathcal{C}$ that matches it, then $\Delta_{\mathsf{Ts}}(\mathcal{C})$ is satisfiable. Moreover, it follows that $\mathcal{C}$ must admit regular solutions.

Corollary 6.5.7 (Condition for Completeness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$) *Let Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and let $\pi \models \mathcal{C}$ with π matches Ts. Then $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$.*

Proof. Follows by Lemma 6.5.5, item 4. □

6.5.2 Computation of $\Delta_{\mathsf{Ts}}(\mathcal{C})$

In the previous section we described the set $\Delta_{\mathsf{Ts}}(\mathcal{C})$ and proved that its satisfiability implies the satisfiability of $\mathcal{C}$. The natural question that arises is how to compute $\Delta_{\mathsf{Ts}}(\mathcal{C})$. Computing $\mathcal{AC}[\mathsf{Ts}]$ is simple, the challenge is the computation of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$. Adding constraints to the set for each path $w \in \mathcal{L}^*$ according to the definition is clearly infeasible since there are infinitely many paths. However, we can calculate the desired set by iteration: we build a set

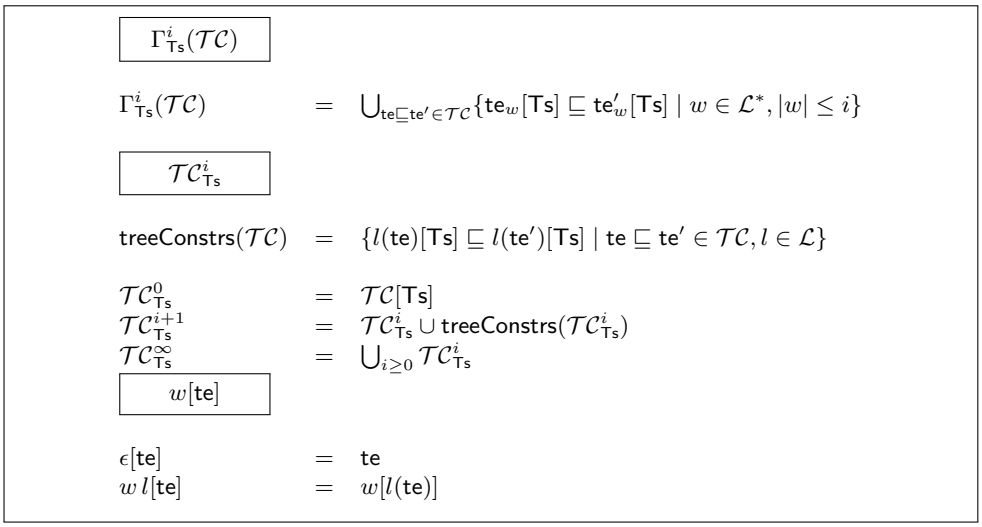


Figure 6.8: $\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ and $\mathcal{TC}^i_{\mathsf{Ts}}$.

$\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ iteratively. In the i-th step of the iteration the set contains exactly the constraints corresponding to the paths w with $|w| \leq i$. We prove that the iteration terminates, i.e. that there is an index j with $\Gamma^j_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^{j+1}_{\mathsf{Ts}}(\mathcal{TC})$ and that this set contains all the constraints in $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$.

The sets $\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ are useful for proving the soundness of the iteration and for understanding how it works. However, actually building the sets in each iteration would be inefficient. Instead, we build a set of tree constraints $\mathcal{TC}^i_{\mathsf{Ts}}$ iteratively (Fig. 6.8), by adding new constraints in each step, so that the following invariant holds: for all i, $\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^i_{\mathsf{Ts}}) = \Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$. In the following, we prove the soundness of the iteration: $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^\infty_{\mathsf{Ts}})$ that follows directly from the invariant. For the proof it is convenient to define $w[\mathsf{te}] : \mathsf{TExp}$ (Fig. 6.8) to be able to describe the constraints in $\mathcal{TC}^i$ for some i in terms of the constraints in $\mathcal{TC}$.

Lemma 6.5.8 *Let $\mathcal{TC}$ be a set of constraints and $\mathsf{tc} \in \mathcal{TC}$ and Ts be a tree schema and $\mathsf{te} \in \mathsf{TExp}$ and $w \in \mathcal{L}^*$. Then:*

1. $\mathsf{te}_w[\mathsf{Ts}] = \bar{w}[\mathsf{te}][\mathsf{Ts}]_\epsilon$
2. $\mathcal{TC}^i_{\mathsf{Ts}} = \{\bar{w}[\mathsf{te}][\mathsf{Ts}] \leq \bar{w}[\mathsf{te}'][\mathsf{Ts}] \mid \mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}, w \in \mathcal{L}^*, |w| \leq i\}$
3. *For all i holds* $\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^i_{\mathsf{Ts}}) = \Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$.

Proof.

1. By induction on w.
2. By induction on i.

Case $i \to i+1$

$$\begin{array}{rcl}
\mathcal{TC}^{i+1}_{\mathsf{Ts}} & = & \mathcal{TC}^{i}_{\mathsf{Ts}} \cup \mathsf{treeConstrs}(\mathcal{TC}^{i}_{\mathsf{Ts}}) \\
& =_{\text{I.H.}} & \{\bar{w}[\mathsf{te}][\mathsf{Ts}] \leq \bar{w}[\mathsf{te}'][\mathsf{Ts}] \mid \mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}, |w| \leq i\} \cup \\
& & \{l\,\bar{w}[\mathsf{te}][\mathsf{Ts}] \leq l\,\bar{w}[\mathsf{te}'][\mathsf{Ts}] \mid \mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}, |w| \leq i\} \\
& = & \{\bar{w}[\mathsf{te}][\mathsf{Ts}] \leq \bar{w}[\mathsf{te}'][\mathsf{Ts}] \mid |w| \leq i+1\}
\end{array}$$

3. We show

$$\begin{array}{lcl}
\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^i_{\mathsf{Ts}}) & = & \Gamma^i_{\mathsf{Ts}}(\mathcal{TC}) \\
\{\bar{w}[\mathsf{te}][\mathsf{Ts}]_\epsilon \leq \bar{w}[\mathsf{te}'][\mathsf{Ts}]_\epsilon\} & =_{2.} & \{\mathsf{te}_w[\mathsf{Ts}] \leq \mathsf{te}'_w[\mathsf{Ts}]\}
\end{array}$$

for each $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$ and $w \in \mathcal{L}^*$ with $|w| \leq i$, and that follows by 1.

□

Corollary 6.5.9 (Soundness of iteration) *Let $\mathcal{TC}$ be a set of constraints and Ts be a tree schema. Then: $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^\infty_{\mathsf{Ts}})$.*

Proof. Follows from Lemma 6.5.8, item 3., since $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \bigcup_{i\geq 0} \Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$.

□

Next, we prove termination of the iteration. The proof consists of two parts. First we notice that, since $\mathcal{TC}^i_{\mathsf{Ts}} \subseteq \mathcal{TC}^{i+1}_{\mathsf{Ts}}$ for all i, if there exists an index n_0 with $\mathsf{treeConstrs}(\mathcal{TC}^{n_0}_{\mathsf{Ts}}) \subseteq \mathcal{TC}^{n_0}_{\mathsf{Ts}}$, then $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$ and for all $i \geq n_0$ $\mathcal{TC}^i_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$. The second part of the proof consists in showing that such an index indeed exists for this sequence. It follows from the soundness of the iteration and from the fact that the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ is finite.

Lemma 6.5.10 (Termination of iteration) *Let $\mathcal{TC}$ be a set of constraints and Ts be a tree schema. Then:*

1. *If there is n_0 with $\mathsf{treeConstrs}(\mathcal{TC}^{n_0}_{\mathsf{Ts}}) \subseteq \mathcal{TC}^{n_0}_{\mathsf{Ts}}$ then $\forall i \geq n_0\,.\,\mathcal{TC}^i_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$.*
2. *There is n_0 with $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$ and $\mathcal{TC}^\infty_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$.*

Proof.

1. By induction on i.
2. Let us assume $\mathcal{TC}^i_{\mathsf{Ts}} \subsetneq \mathcal{TC}^{i+1}_{\mathsf{Ts}}$ for all i. Then, $\mathcal{TC}^\infty_{\mathsf{Ts}}$ is infinite and $\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^\infty_{\mathsf{Ts}}) = \Gamma_{\mathsf{Ts}}(\mathcal{TC})$ is infinite as well, but this is a contradiction to Lemma 6.5.3. Therefore, there exists n_0 with $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$. Then, $\mathcal{TC}^\infty_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$ follows by 1.

□

Corollary 6.5.11 (Computation of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$) *Let $\mathcal{TC}$ be a set of constraints and Ts be a tree schema and $n_0 \in \mathbb{N}$ with $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$. Then $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^{n_0}_{\mathsf{Ts}})$.*

Proof. Follows by Corollary 6.5.9 and Lemma 6.5.10. □

Example 6.5.12 (Computation of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$) *Let Ts and $\mathcal{L}$ be defined like in Example 6.5.2 and $\mathcal{TC}$ be defined like in Example 6.5.4. Then, we can build $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as follows:*

$$\begin{array}{rcl} \mathcal{TC}^0_{\mathsf{Ts}} & = & \{x_2 \sqsubseteq x_2, x_1 \sqsubseteq x_1\} \\ \mathcal{TC}^1_{\mathsf{Ts}} & = & \{x_1 \sqsubseteq x_1, x_2 \sqsubseteq x_2\} \end{array}$$

Since $\mathcal{TC}^0_{\mathsf{Ts}} = \mathcal{TC}^1_{\mathsf{Ts}}$, it follows $\mathcal{TC}^\infty_{\mathsf{Ts}} = \mathcal{TC}^1_{\mathsf{Ts}}$. Moreover

$$\begin{array}{rcl} \Gamma_{\mathsf{Ts}}(\mathcal{TC}) & = & \Gamma^0_{\mathsf{Ts}}(\{x_1 \sqsubseteq x_1, x_2 \sqsubseteq x_2\}) \\ & = & \{\lambda_1 \leq \lambda_1, \lambda_2 \leq \lambda_2\} \end{array}$$

6.5.3 Linear constraint system (LCS)

In the previous section we proved that our algorithm for solving satisfiability for a system of constraints is sound for *any* given tree schema. We also noticed that the algorithm cannot be complete, because it imposes a regularity condition on the solutions. In this section we study the following questions: Is there a subset of $\mathsf{TConstr}$, for which the algorithm is complete? Then, how do we find the right tree schemas?

A set of tree constraints $\mathcal{TC}$ induces a graph $G = (V, E)$ whose vertices V are the tree variables occurring in $\mathcal{TC}$. The set of edges E is defined as follows: for each $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$, for each $x \in \mathsf{Vars}(\mathsf{te})$ and $y \in \mathsf{Vars}(\mathsf{te}')$, we add (x, y) to E. Then, we say that a set of tree constraints $\mathcal{TC}$ *contains a loop*, when its corresponding graph G contains a closed path. Moreover, we say that a subset $\mathcal{TC}' \subseteq \mathcal{TC}$ *is a loop*, if the graph G contains a closed path P and for all $\mathsf{tc} \in \mathcal{TC}'$ there exists a variable $x_i \in P$ with $x_i \in \mathsf{Vars}(\mathsf{tc})$ and for all $x_i \in P$ there exists $\mathsf{tc} \in \mathcal{TC}'$ with $x_i \in \mathsf{Vars}(\mathsf{tc})$.

We wish to describe a subset LCS of $\mathsf{TConstr}$ such that for each system of constraints $\mathcal{C} \in$ LCS we can effectively construct a tree schema $\mathsf{Ts}_{\mathcal{C}}$ with the following property: if $\mathcal{C}$ is satisfiable, there exists a valuation π matching $\mathsf{Ts}_{\mathcal{C}}$ and satisfying $\mathcal{C}$. We will describe that set as a collection of loops of a certain restricted form together with constraints defining relations between the variables that appear in the loops. In particular, the loops should not contain compound expressions. Moreover, every variable x that appears in a loop may appear in arithmetic constraints only in sub-expressions $\Diamond(x)$. The following grammar describes the restricted sets of tree constraints $\mathsf{LTConstr}$, $\mathsf{RTConstr}$ and the restricted set of arithmetic constraints $\mathsf{LAConstr}$.

Tree schema for a LCS $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$.

$$
\begin{array}{lcl}
\mathsf{Ts}_{\mathcal{C}}.X & = & \mathsf{Vars}(\mathcal{TC}) \\
\mathsf{Ts}_{\mathcal{C}}.\mathsf{T}_{\mathbb{D}}^{\mathcal{L}} & = & \{\widehat{0}, \widehat{\infty}\} \\
\forall\, x_i \in \mathsf{Ts}_{\mathcal{C}}.X : & & \\
\quad \mathsf{Ts}_{\mathcal{C}}.\Diamond(x_i) & = & \lambda_i \ \text{ where } \lambda_i \notin \mathsf{Vars}(\mathcal{AC}) \\
\quad \forall l_j \in \mathcal{L}\,.\,\mathsf{Ts}_{\mathcal{C}}.\mathsf{next}(l_j, x_i) & = & \begin{cases} x_k & \text{if } \ (l_j(x_i) \sqsubseteq x_k) \in \mathcal{TC} \ \text{ or } \ (x_k \sqsubseteq l_j(x_i)) \in \mathcal{TC} \\ \widehat{\infty} & \text{otherwise, if } \ x_i \ \text{ occurs in a left linear loop.} \\ \widehat{0} & \text{otherwise, if } \ x_i \ \text{ occurs in a right linear loop.} \end{cases} \\
\pi_a^{(\pi_t, \mathsf{Ts})} & = & \{\delta \mapsto \Diamond(\pi_t(x)) \mid x \in \mathsf{Ts}.X, \mathsf{Ts}.\Diamond(x) = \delta\}
\end{array}
$$

Figure 6.9: Tree schema for a linear constraint system.

$$
\begin{array}{lcll}
\mathsf{ltc} & ::= & l(x) \sqsubseteq x & \in \mathsf{LTConstr} \\
\mathsf{rtc} & ::= & x \sqsubseteq l(x) & \in \mathsf{RTConstr} \\
\mathsf{pae} & ::= & n \mid \lambda \mid \Diamond(x) \mid \mathsf{pae} + \mathsf{pae} & \in \mathsf{PAExp} \\
\mathsf{lac} & ::= & \mathsf{pae} \leq \mathsf{pae} & \in \mathsf{LAConstr}
\end{array}
$$

Definition 6.5.13 (Linear loop) *Let $\mathcal{TC}' \subseteq \mathcal{TC}$ be a loop.*

1. *We say that $\mathcal{TC}'$ is a* left linear loop *if $\mathcal{TC}' \subseteq \mathsf{LTConstr}$ and for all $x \in \mathcal{TC}'$ holds x occurs only positively in $\mathcal{TC} \setminus \mathcal{TC}'$.*

2. *Further, we say that $\mathcal{TC}'$ is a* right linear loop *if $\mathcal{TC}' \subseteq \mathsf{RTConstr}$ and for all $x \in \mathcal{TC}'$ holds x occurs only negatively in $\mathcal{TC} \setminus \mathcal{TC}'$.*

3. *We say that $\mathcal{TC}'$ is a* linear loop *if it is either a left linear or a right linear loop and for all $x \in \mathsf{Vars}(\mathcal{TC}')$ holds if $x \in \mathsf{Vars}(\mathsf{ac})$ for some arithmetic constraint ac then $\mathsf{ac} \in \mathsf{LAConstr}$.*

Definition 6.5.14 (Linear constraint system (LCS)) *We say that $\mathcal{TC}$ is* linear *if $\mathcal{TC} = \mathcal{TC}' \cup (\bigcup_{i=1,\ldots,n} \mathcal{TC}_i)$ where each $\mathcal{TC}_i$ is a linear loop with $\mathsf{Vars}(\mathcal{TC}_i) \cap \mathsf{Vars}(\mathcal{TC}_j) = \emptyset$ for $i \neq j$ and $\mathsf{Vars}(\mathcal{TC}') \subseteq \mathsf{Vars}(\bigcup_{i=1,\ldots,n} \mathcal{TC}_i)$.*

We remark that the subsystem $\mathcal{TC}'$ from the definition does not contain loops. This follows by the definition since the variables in $\mathcal{TC}$ must appear either only positively or only negatively in $\mathcal{TC}'$.

Example 6.5.15 *Let $\mathcal{L} = \{l\}$ and let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and*

$$
\mathcal{TC} = \{ \begin{array}{lcl} l(x_1) & \sqsubseteq & x_2, \\ l(x_2) & \sqsubseteq & x_1 \end{array} \} \qquad \mathcal{AC} = \{ \begin{array}{lcl} \Diamond(x_1) & \leq & \Diamond(x_2) + 1, \\ 1 & \leq & \Diamond(x_2) \end{array} \}
$$

Then, $\mathcal{TC}$ is a left linear loop and $\mathcal{C} \in$ LCS.

In Fig. 6.9 we define a type of tree schema for any given LCS $\mathcal{C}$. We show in the following Lemma that if $\mathcal{C}$ is satisfiable there is a valuation that

both satisfies $\mathcal{C}$ and matches the tree schema $\mathsf{Ts}_{\mathcal{C}}$. For the construction of such valuation we use a valuation $\pi_a^{(\pi_t,\mathsf{Ts})} : \Lambda \to \mathbb{D}$ (Fig. 6.9) that we build on the basis of another valuation π_t and a tree schema Ts.

Lemma 6.5.16 *Let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ be a satisfiable LCS. Then there is a valuation π' with $\pi' \models \mathcal{C}$ and π' matches $\mathsf{Ts}_{\mathcal{C}}$.*

Proof. We have given a valuation $\pi = (\pi_t, \pi_a)$ with $\pi \models \mathcal{C}$. Let $\mathcal{TC} = \mathcal{TC}' \cup (\bigcup_{j=1,\ldots,m} \mathcal{TC}_j(\vec{x_j}))$. For ease of notation, let us assume that $|\vec{x_j}| = 1$. Thus, $\vec{x_j} = x_j$ and let $\pi(x_j) = t_j$. We define $\hat{t}_j$ by:

Case $\mathcal{TC}_j = l(x_j) \sqsubseteq x_j)$. We set $l_k(\hat{t}_j) = \widehat{\infty}$ for $l_k \neq l \in \mathcal{L}$.

Case $\mathcal{TC}_j = x_j \sqsubseteq l(x_j)$. We set $l_k(\hat{t}_j) = \widehat{0}$ for $l_k \neq l \in \mathcal{L}$.

Moreover we set $\Diamond(\hat{t}_j) = \Diamond(t_j)$ and $l(\hat{t}_j) = \hat{t}_j$. Now we set $\hat{\pi}_t = \pi_t[x_j \mapsto \hat{t}_j]$ and $\hat{\pi}_a = \pi_a \cup \pi_a^{(\hat{\pi}_t,\mathsf{Ts}_{\mathcal{C}})}$ and $\hat{\pi} = (\hat{\pi}_t, \hat{\pi}_a)$. We show $\hat{\pi} \models \mathcal{C}$ and $\hat{\pi}$ matches $\mathsf{Ts}_{\mathcal{C}}$: $\hat{\pi}$ matches $\mathsf{Ts}_{\mathcal{C}}$ by construction, $\hat{\pi} \models \mathcal{TC}_j$ follows by construction and $\hat{\pi} \models \mathcal{AC}$ follows by $\pi \models \mathcal{AC}$ and $\Diamond(\hat{t}_j) = \Diamond(t_j)$. Moreover, $\hat{\pi} \models \mathcal{TC}'$ follows by Lemma 6.3.6 because if $\mathcal{TC}_j$ is a left linear loop then $\mathcal{TC}'(x_j^+)$ and $t_j \sqsubseteq \hat{t}_j$ and if $\mathcal{TC}_j$ is a right linear loop then $\mathcal{TC}'(x_j^-)$ and $\hat{t}_j \sqsubseteq t_j$.

We remark that, in the general case, where $|\vec{x_j}| \geq 1$, and $\vec{x_j} = x_{j1}, \ldots, x_{jn}$, and we are given that $\pi(x_{ji}) = t_{ji}$, we build $\hat{t}_{ji}$ analogously, and we prove by coinduction that if $\mathcal{TC}_j$ is a left linear loop then $t_{ji} \sqsubseteq \hat{t}_{ji}$ and, if $\mathcal{TC}_j$ is a right linear loop then $\hat{t}_{ji} \sqsubseteq t_{ji}$. □

Theorem 6.5.17 (Completeness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$) *Let $\mathcal{C}$ be a satisfiable LCS. Then there is a tree schema Ts and a valuation π_a with $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$.*

Proof. By Lemma 6.5.16 we obtain a valuation $\pi \models \mathcal{C}$ with $\pi = (\pi_t, \pi_a)$ matches $\mathsf{Ts}_{\mathcal{C}}$. Moreover, by Lemma 6.5.7, we obtain $\pi_a \models \Delta_{\mathsf{Ts}_{\mathcal{C}}}(\mathcal{C})$. □

The restriction to linear constraint systems could seem very strong. However, we presented an algorithm for eliminating variables from constraints while maintaining their satisfiability in Section 6.4. In most cases we are able to eliminate the variables that are not part of a loop with that procedure. Further, we can often bring the loops in the required form by eliminating intermediate variables. For example, the loop $\{l(x) \sqsubseteq y, y \sqsubseteq x\}$ can be transformed into $l(x) \sqsubseteq x$ if we eliminate y. On the other hand, there are systems such as $\{x + x \sqsubseteq l(x)\}, \{1 \leq \Diamond(x)\}$ that can not be transformed into an equivalent linear one. In fact, there is no regular solution for that system.

Left or right linear loops to linear loops. After we have transformed the constraints into a collection of left and right linear loops using variable elimination, then we can transform those loops into linear loops. In particular, if the variables in the loop appear in arithmetic expressions of the

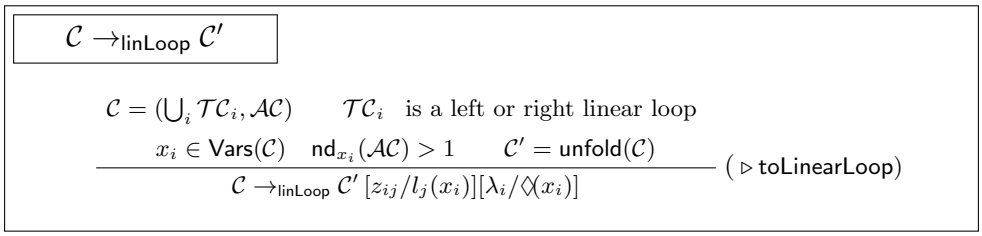


Figure 6.10: Bringing left or right linear loops into linear loops.

form $\Diamond(\mathsf{tae}(x))$, we show how to transform the loops into equivalent ones, whose variables appear in arithmetic expressions of the required canonical form $\Diamond(x)$.

The transformation is based on unfolding the definition of inequality and substituting atomic expressions with fresh new variables, just like in the rule $(\triangleright\, \mathsf{Elim}^{+/-})$ of the elimination procedure. Concretely, we define the reduction $\mathcal{C} \to_{\mathsf{linLoop}} \mathcal{C}'$ by the rule $(\triangleright\, \mathsf{toLinearLoop})$ shown in Fig. 6.10. Starting from a set of left or right linear loops, whose variables appear in the arithmetic constraints with a nesting depth greater than 1, the resulting system of constraints is built by unfolding the constraints, followed by the substitution of the expressions $l_j(x_i)$ and $\Diamond(x_i)$ with fresh variables z_{ij} and λ_i, respectively. The idea is to iterate this process until, for each variable x, the nested depth of x in the arithmetic constraints is 1, resulting in a linear system of constraints. We write $\mathcal{C} \to^*_{\mathsf{linLoop}} \mathcal{C}''$ for meaning the described iteration. Consequently, $\mathcal{C}''$ is a linear constraint system. In the following we show the correctness of the algorithm and that the iteration terminates.

Lemma 6.5.18 *Let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ be a system of constraints and let $\mathcal{TC}$ be a set of left or right linear loops and let $\mathcal{C} \to_{\mathsf{linLoop}} \mathcal{C}' = (\mathcal{TC}', \mathcal{AC}')$. Then:*

1. $\mathcal{C} \iff \mathcal{C}'$.
2. $\mathsf{nd}_{z_{ij}}(\mathcal{AC}') < \mathsf{nd}_{x_i}(\mathcal{AC})$ *or* $\mathsf{nd}_{z_{ij}}(\mathcal{AC}') = 1$.

Proof.

1. Follows by Lemma 6.3.3.
2. Let $\mathsf{ac}' \in \mathcal{AC}'$ with $z_{ij} \in \mathsf{Vars}(\mathsf{ac})$. Then $\mathsf{ac} = \mathsf{ac}'\,[l_j(x_i)/z_{ij}, \Diamond(x_i)/\lambda_i] \in \mathcal{C}'$. Then is either

   Case $\mathsf{ac} \in \mathcal{AC}$. Then $\mathsf{nd}_{z_{ij}}(\mathsf{ac}') = \mathsf{nd}_{x_i}(\mathsf{ac}) - 1$.

   Case $\mathsf{ac} \notin \mathcal{AC}$. Then $\mathsf{nd}_{z_{ij}}(\mathsf{ac}') = 1$ because $\mathsf{nd}_{x_i}(\mathsf{ac}) = 2$ because of the shape of the loops in $\mathcal{TC}$, i.e., if $l(x_i) \sqsubseteq x_i \in \mathcal{C}$ then $\Diamond(l(x_i)) \leq \Diamond(x_i) \in \mathcal{AC}'$ and $\mathsf{nd}_{x_i}(\Diamond(l(x_i)) \leq \Diamond(x_i)) = 2$. □

Corollary 6.5.19 *Let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ be a system of constraints and let $\mathcal{TC}$ be a set of left or right linear loops and let $\mathcal{C} \rightarrow^*_{\mathsf{linLoop}} \mathcal{C}' = (\mathcal{TC}', \mathcal{AC}')$. Then:*

1. *$\mathcal{C} \iff \mathcal{C}'$.*
2. *$\mathcal{C}'$ is a linear loop and the iteration terminates.*

Proof.

1. Follows by Lemma 6.5.18, item 1.
2. We obtain by Lemma 6.5.18, item 2: $\mathsf{nd}_{z_{ij}}(\mathcal{AC}') < \mathsf{nd}_{x_i}(\mathcal{AC})$ or $\mathsf{nd}_{z_{ij}}(\mathcal{AC}') = 1$. Thus, the iteration terminates when $\mathsf{nd}_{z_{ij}}(\mathcal{AC}') = 1$, because of the side condition $\mathsf{nd}_{z_{ij}}(\mathcal{AC}) > 1$ of the rule ($\triangleright$toLinearLoop). Consequently, if $z_{ij} \in \mathsf{ac}$ for some $\mathsf{ac} \in \mathcal{AC}'$, then $\mathsf{ac} \in \mathsf{LAConstr}$.

□

Example 6.5.20 *Let $\mathcal{L} = \{l\}$ and let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and*

$$\mathcal{TC} = \{\ l(x) \ \sqsubseteq\ x,\ \} \qquad \mathcal{AC} = \{\ \Diamond\!\langle(l(x)) \ \leq\ 1,\ \}$$

Then, $\mathcal{C}' = \mathsf{unfold}(\mathcal{C})$:

$$\mathcal{TC}' = \{\ l(l(x)) \ \sqsubseteq\ l(x),\ \} \qquad \mathcal{AC}' = \{\ \begin{array}{lll} \Diamond\!\langle(l(x)) & \leq & 1, \\ \Diamond\!\langle(l(x)) & \leq & \Diamond\!\langle(x) \end{array}\ \}$$

Then, $\mathcal{C}'' = \mathcal{C}'\,[z/l(x)][\lambda/\Diamond\!\langle(x)]$:

$$\mathcal{TC}'' = \{\ l(z) \ \sqsubseteq\ z,\ \} \qquad \mathcal{AC}'' = \{\ \begin{array}{lll} \Diamond\!\langle(z) & \leq & 1, \\ \Diamond\!\langle(z) & \leq & \lambda \end{array}\ \}$$

and $\mathcal{C}'' \in LCS$.

6.5.4 Heuristic algorithm for solving a system of constraints

In the previous section we presented an algorithm for solving a system of constraints, when a tree schema is given. Moreover, we described a subset LCS for which we can effectively build a tree schema that we can use for solving the constraints. Moreover, we saw how to bring systems of constraints into LCS in some cases; that is, in some cases when the loops in the system do not contain compound expressions. Here, we wish to summarise the algorithms and procedures presented so far, to build a heuristic algorithm for solving a system of constraints. The algorithm consists of the steps described in Fig 6.11.

In the following we show the correctness of the heuristic algorithm, which follows from the correctness of each step. Notice that the algorithm is sound

Heuristic algorithm for solving a system of constraints $\mathcal{C}$.

1. Eliminate variables from $\mathcal{C}$ until the only remaining variables are those that appear in a loop, and obtain $\mathcal{C}_1$.
2. Try to bring the loops of $\mathcal{C}_1$ with variable elimination to be left or right linear, and obtain $\mathcal{C}_2$.
3. If all the loops are either left or right linear then apply the algorithm for creating linear loops to $\mathcal{C}_2$ ($\mathcal{C}_2 \rightarrow^*_{\mathsf{linLoop}} \mathcal{C}_3$). Then, construct the tree schema Ts for $\mathcal{C}_3$ as described in Section 6.5.3.
4. Otherwise, construct a tree schema Ts for $\mathcal{C}_1$ using a heuristic procedure.
5. Compute $\Delta_{\mathsf{Ts}}(\mathcal{C}_3)$ (or $\Delta_{\mathsf{Ts}}(\mathcal{C}_1)$) by the iteration procedure described in Section 6.5.3.
6. Try to solve $\Delta_{\mathsf{Ts}}(\mathcal{C}_3)$ (or $\Delta_{\mathsf{Ts}}(\mathcal{C}_1)$) with an LP-Solver, to obtain a valuation π_a.
7. Build a valuation $\mathsf{Ts}[\pi_a]$ based on π_a and Ts.

Figure 6.11: Heuristic algorithm for solving a system of constraints $\mathcal{C}$.

for any tree schema. Our algorithm is not complete, yet we shall see in Chapter 7 that we can compute linear bounds on the heap-space requirements of many interesting programs with a tool that uses this algorithm.

We remark that, because we eliminate variables, the valuation given by the algorithm needs to be extended to satisfy the constraints. Nevertheless, as discussed in Section 6.4, we provide a procedure for extending the valuation in the proof of Theorem 6.4.1.

Theorem 6.5.21 (Soundness of the algorithm) *Let $\mathcal{C}$ be a system of constraints and let π be the valuation given by the algorithm described in Fig. 6.11. Then, there exists $\pi' \supseteq \pi$ with $\pi' \models \mathcal{C}$.*

Proof. By Corollary 6.5.11, the iteration indeed computes $\Delta_{\mathsf{Ts}}(\mathcal{C}_3)$. Then, by Theorem 6.5.6, we know $\pi \models \mathcal{C}_3$. Then, $\pi \models \mathcal{C}_2$, by Corollary 6.5.19, and there exists $\pi' \supseteq \pi$ with $\pi' \models \mathcal{C}_1$ and $\pi'' \supseteq \pi'$ with $\pi'' \models \mathcal{C}$, both by Theorem 6.4.1. □

Next, we show that the algorithm terminates.

Theorem 6.5.22 (Termination of the algorithm)
The algorithm described in Fig. 6.11 terminates.

Proof. Follows by the termination of the elimination procedure (Theorem 6.4.2), termination of the iteration for calculating $\Delta_{\mathsf{Ts}}(\mathcal{C}_3)$ (Lemma 6.5.10), and termination of the creation of linear loops (Lemma 6.5.19, item 2). □

6.6 Applications to resource analysis

In this section we show a reduction from the constraints that are generated by the algorithm from Section 5.3, and that build polymorphic RAJA method types, to the systems of constraints described in this chapter. The reduction allows the use of the heuristic algorithm presented in the previous section, for solving the subtyping and arithmetic constraints, which describe the heap-space requirements of FJEU programs.

6.6.1 Views to infinite trees

First, we present a reduction from views to infinite trees. As we discussed in Chapter 3, views are infinite trees themselves. However, the infinite trees described in this chapter are simpler. On the one hand, the nodes of the infinite trees contain one nonnegative real number, whereas the nodes of views contain one such number for each class. On the other hand, subtyping for views is contravariant in the set views; that is, in some of its subtrees. In contrast, inequality over infinite trees is covariant in all subtrees. Thus, the idea of the reduction is to separate the "positive parts" and "negative parts" of a view, to build infinite trees.

To reduce a view $r \in \mathscr{V}^{\mathscr{C}}$, we define, for each class $C_i \in \mathscr{C}$, the infinite trees $r_i^+, r_i^- \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, where

$$\begin{array}{lcl} \mathcal{L} & = & \mathcal{L}^+ \cup \mathcal{L}^- \text{ and} \\ \mathcal{L}^+ & = & \{l^+_{kj} \mid C_k \in \mathscr{C}, a_j \in \mathsf{A}(C_k)\} \\ \mathcal{L}^- & = & \{l^-_{kj} \mid C_k \in \mathscr{C}, a_j \in \mathsf{A}(C_k)\} \end{array} \tag{6.6.1}$$

such that we can reduce any subtyping statement between views to inequalities between trees. More exactly, we want to prove:

$$r \sqsubseteq s \iff \bigwedge_{C_i \in \mathscr{C}} s_i^+ \sqsubseteq r_i^+ \ \wedge \ r_i^- \sqsubseteq s_i^- \tag{6.6.2}$$

Definition 6.6.1 *Let $r \in \mathscr{V}^{\mathscr{C}}$. We define the function* $\mathsf{expand}(r) = (\vec{r^+}, \vec{r^-})$, *where r_i^+ and r_i^- are defined coinductively as follows. Let $C_i \in \mathscr{C}$ and $C_k \in \mathscr{C}$ and $a_j \in \mathsf{A}(C_k)$.*

$$\begin{array}{lcllcl} \Diamond(r_i^+) & = & \Diamond(C_i^r) & \Diamond(r_i^-) & = & 0 \\ l^+_{kj}(r_i^+) & = & \mathsf{A}^{\mathrm{get}}(C_k^r, a_j)_i^+ & l^+_{kj}(r_i^-) & = & \mathsf{A}^{\mathrm{get}}(C_k^r, a_j)_i^- \\ l^-_{kj}(r_i^+) & = & \mathsf{A}^{\mathrm{set}}(C_k^r, a_j)_i^- & l^-_{kj}(r_i^-) & = & \mathsf{A}^{\mathrm{set}}(C_k^r, a_j)_i^+ \end{array}$$

Notice that expand is an injective function. Fig. 6.12 shows a representation of the reduction applied to the view rich, assuming that the only class in the program is Cons, where g represents Cons.next. get and s represents Cons.next. set.

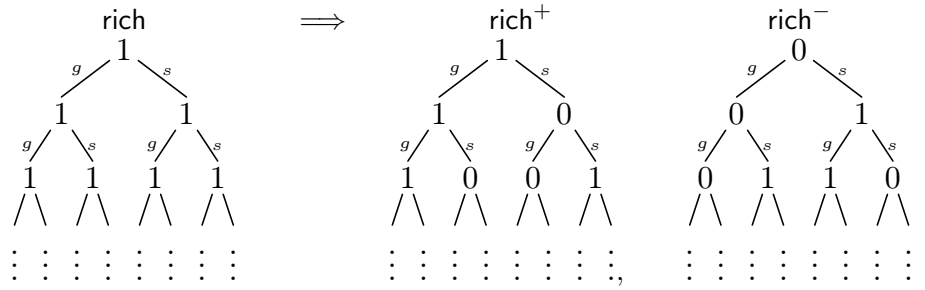


Figure 6.12: View rich and how it is reduced to the infinite trees rich^+ and rich^-.

In this context it is useful to define positive and negative infinite trees.

Definition 6.6.2 *Let $t \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$.*

1. *We say that t* is positive *if for each $l^+ \in \mathcal{L}^+$ holds $l^+(t)$ is positive and for each $l^- \in \mathcal{L}^-$ holds $l^-(t)$ is negative.*

2. *We say that t* is negative *if $\Diamond(t) = 0$ and for each $l^+ \in \mathcal{L}^+$ holds $l^+(t)$ is negative and for each $l^- \in \mathcal{L}^-$ holds $l^-(t)$ is positive.*

It is clear that if $\mathsf{expand}(r) = (\vec{r^+}, \vec{r^-})$, then r_i^+ is positive and r_i^- is negative. In the following two lemmas, we prove the required property (6.6.2).

Lemma 6.6.3 *Let $r, s \in \mathscr{V}^{\mathscr{C}}$ and $r \sqsubseteq s$ and let $(\vec{r^+}, \vec{r^-}) = \mathsf{expand}(r)$ and $(\vec{s^+}, \vec{s^-}) = \mathsf{expand}(s)$. Then:*

1. *$s_i^+ \sqsubseteq r_i^+$ for all i.*

2. *$r_i^- \sqsubseteq s_i^-$ for all i.*

Proof. Simultaneously by coinduction.

1. • We show
$$\begin{array}{rcll} \Diamond({s_i}^+) & \leq & \Diamond({r_i}^+) & \iff \\ \Diamond(C_i^s) & \leq & \Diamond(C_i^r) & \end{array}$$
which follows by $r \sqsubseteq s$.

   • We show
$$\begin{array}{lcll} l_{kj}^+({s_i}^+) & \sqsubseteq & l_{kj}^+({r_i}^+) & \iff \\ \mathsf{A}^{\mathrm{get}}(C_k^s, a_j)_i^+ & \sqsubseteq & \mathsf{A}^{\mathrm{get}}(C_k^r, a_j)_i^+ & \end{array}$$
which follows by the coinduction hypothesis 1., because $\mathsf{A}^{\mathrm{get}}(C_k^r, a_j) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C_k^s, a_j)$ by assumption.

- We show
$$\begin{array}{lcll} l^-_{kj}(s_i{}^+) & \sqsubseteq & l^-_{kj}(r_i{}^+) & \Longleftrightarrow \\ \mathsf{A}^{\mathrm{set}}(C^s_k\, a_j)^-_i & \sqsubseteq & \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)^-_i & \end{array}$$
which follows by the coinduction hypothesis 2., because $\mathsf{A}^{\mathrm{set}}(C^s_k\, a_j) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)$ by assumption.

2. - We show
$$\begin{array}{lcll} \Diamond\!\!(r_i{}^-) & \leq & \Diamond\!\!(s_i{}^-) & \Longleftrightarrow \\ 0 & \leq & 0 & \end{array}$$
which follows trivially.
- We show
$$\begin{array}{lcll} l^+_{kj}(r_i{}^-) & \sqsubseteq & l^+_{kj}(s_i{}^-) & \Longleftrightarrow \\ \mathsf{A}^{\mathrm{get}}(C^r_k\, a_j)^-_i & \sqsubseteq & \mathsf{A}^{\mathrm{get}}(C^s_k\, a_j)^-_i & \end{array}$$
which follows by the coinduction hypothesis 2., because $\mathsf{A}^{\mathrm{get}}(C^r_k\, a_j) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^s_k\, a_j)$ by assumption.
- We show
$$\begin{array}{lcll} l^-_{kj}(r_i{}^-) & \sqsubseteq & l^-_{kj}(s_i{}^-) & \Longleftrightarrow \\ \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)^+_i & \sqsubseteq & \mathsf{A}^{\mathrm{set}}(C^s_k\, a_j)^+_i & \end{array}$$
which follows by the coinduction hypothesis 1., because $\mathsf{A}^{\mathrm{set}}(C^s_k\, a_j) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)$ by assumption.

□

Lemma 6.6.4 *Let $r, s \in \mathscr{V}^{\mathscr{C}}$ and $(\vec{r^+}, \vec{r^-}) = \mathsf{expand}(r)$ and $(\vec{s^+}, \vec{s^-}) = \mathsf{expand}(s)$. Then: $s_i^+ \sqsubseteq r_i^+$ and $r_i^- \sqsubseteq s_i^-$ for all i imply $r \sqsubseteq s$.*

Proof. By coinduction.

- We show $\Diamond\!\!(C^r_i) \geq \Diamond\!\!(C^s_i) \Rightarrow \Diamond\!\!(r_i{}^+) \geq \Diamond\!\!(s_i{}^+)$ which follows by assumption.
- We show $\mathsf{A}^{\mathrm{get}}(C^r_k\, a_j) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^s_k\, a_j)$, which follows by the coinduction hypothesis, because by assumption we have:
$$\begin{array}{llllll} l^+_{kj}(s_i{}^+) & \sqsubseteq & l^+_{kj}(r_i{}^+), & l^+_{kj}(r_i{}^-) & \sqsubseteq & l^+_{kj}(s_i{}^-) \Longleftrightarrow \\ \mathsf{A}^{\mathrm{get}}(C^s_k\, a_j)^+_i & \sqsubseteq & \mathsf{A}^{\mathrm{get}}(C^r_k\, a_j)^+_i, & \mathsf{A}^{\mathrm{get}}(C^r_k\, a_j)^-_i & \sqsubseteq & \mathsf{A}^{\mathrm{get}}(C^s_k\, a_j)^-_i \end{array}$$
- We show $\mathsf{A}^{\mathrm{set}}(C^s_k\, a_j) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)$, which follows by the coinduction hypothesis, because by assumption we have:
$$\begin{array}{llllll} l^-_{kj}(s_i{}^+) & \sqsubseteq & l^-_{kj}(r_i{}^+), & l^-_{kj}(r_i{}^-) & \sqsubseteq & l^-_{kj}(s_i{}^-) \Longleftrightarrow \\ \mathsf{A}^{\mathrm{set}}(C^s_k\, a_j)^-_i & \sqsubseteq & \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)^-_i, & \mathsf{A}^{\mathrm{set}}(C^r_k\, a_j)^+_i & \sqsubseteq & \mathsf{A}^{\mathrm{set}}(C^s_k\, a_j)^+_i \end{array}$$
□

The views defined in Section 3.2 ($r_1 \vee r_2, r_1 \wedge r_2, r_1 \oplus r_2, , r_1 \boxplus r_2$) can be reduced to infinite trees with the previous reduction.

Lemma 6.6.5 *Let $r_1, r_2 \in \mathscr{V}^{\mathscr{C}}$. Then:*

1. $(r_1 \vee r_2)^+ = r_1^+ \wedge r_2^+$.
2. $(r_1 \wedge r_2)^+ = r_1^+ \vee r_2^+$.
3. $(r_1 \vee r_2)^- = r_1^- \vee r_2^-$.
4. $(r_1 \wedge r_2)^- = r_1^- \wedge r_2^-$.

Proof. Simultaneously by coinduction. We show the first item for an illustration.

- We show

$$\begin{array}{lcll} \Diamond\big((r_1 \vee r_2)^+\big) & = & \Diamond(r_1{}^+ \wedge r_2{}^+) & \iff \\ \Diamond(C^{r_1 \vee r_2}) & = & \min(\Diamond(r_1{}^+), \Diamond(r_2{}^+)) & \iff \\ \min(\Diamond(C^{r_1}), \Diamond(C^{r_2})) & = & \min(\Diamond(C^{r_1}), \Diamond(C^{r_2})) & \end{array}$$

- We show

$$\begin{array}{lcll} l_{kj}^+((r_1 \vee r_2)^+) & = & l_{kj}^+(r_1{}^+ \wedge r_2{}^+) & \iff \\ \mathsf{A}^{\mathrm{get}}\big(C_k^{r_1 \vee r_2}, a_j\big)^+ & = & l_{kj}^+(r_1{}^+) \wedge l_{kj}^+(r_2{}^+) & \iff \\ (\mathsf{A}^{\mathrm{get}}\big(C_k^{r_1}, a_j\big) \vee \mathsf{A}^{\mathrm{get}}\big(C_k^{r_2}, a_j\big))^+ & = & \mathsf{A}^{\mathrm{get}}\big(C_k^{r_1}, a_j\big)^+ \wedge \mathsf{A}^{\mathrm{get}}\big(C_k^{r_2}, a_j\big)^+ & \end{array}$$

  which follows by the coinduction hypothesis 1.

- We show

$$\begin{array}{lcll} l_{kj}^-((r_1 \vee r_2)^+) & = & l_{kj}^-(r_1{}^+ \wedge r_2{}^+) & \iff \\ \mathsf{A}^{\mathrm{set}}\big(C_k^{r_1 \vee r_2}, a_j\big)^- & = & l_{kj}^-(r_1{}^+) \wedge l_{kj}^-(r_2{}^+) & \iff \\ (\mathsf{A}^{\mathrm{set}}\big(C_k^{r_1}, a_j\big) \wedge \mathsf{A}^{\mathrm{set}}\big(C_k^{r_2}, a_j\big))^- & = & \mathsf{A}^{\mathrm{set}}\big(C_k^{r_1}, a_j\big)^- \wedge \mathsf{A}^{\mathrm{set}}\big(C_k^{r_2}, a_j\big)^- & \end{array}$$

  which follows by the coinduction hypothesis 4.

□

Lemma 6.6.6 *Let $r_1, r_2 \in \mathscr{V}^{\mathscr{C}}$. Then:*

1. $(r_1 \oplus r_2)^+ = r_1^+ + r_2^+$.
2. $(r_1 \boxplus r_2)^+ = r_1^+ \wedge r_2^+$.
3. $(r_1 \oplus r_2)^- = r_1^- \wedge r_2^-$.
4. $(r_1 \boxplus r_2)^- = r_1^- + r_2^-$.

Proof. Simultaneously by coinduction. □

6.6.2 Infinite trees to views

Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ be an FJEU program and let $\mathcal{L}$ be defined as in (6.6.1). Then, we wish to build a view from two vectors of infinite trees. Given two vectors $\vec{t}$ and $\vec{t'}$, with $|\vec{t}| = |\vec{t'}| = |\mathscr{C}|$, such that each $t_i, t'_i \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, we define a function $\mathsf{reduce}(\vec{t}, \vec{t'})$, such that the following holds:

$$\bigwedge_{C_i \in \mathscr{C}} t'_i \sqsubseteq p'_i \ \wedge \ p_i \sqsubseteq t_i \Rightarrow \mathsf{reduce}(\vec{t}, \vec{t'}) \sqsubseteq \mathsf{reduce}(\vec{p}, \vec{p'}) \tag{6.6.3}$$

Definition 6.6.7 *Let $t_i, t'_i \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, for each $C_i \in \mathscr{C}$. We define the function $\mathsf{reduce}(\vec{t}, \vec{t'}) = r$, where $r \in \mathscr{V}^{\mathscr{C}}$, coinductively as follows.*

$$\begin{array}{lcl} \langle\!\langle C^r_i \rangle\!\rangle & = & \langle\!\langle t_i \rangle\!\rangle \dot{-} \langle\!\langle t'_i \rangle\!\rangle \\ \mathsf{A}^{\mathrm{get}}(C^r_k, a_j) & = & \mathsf{reduce}(\overrightarrow{l^+_{kj}(t_i)}, \overrightarrow{l^+_{kj}(t'_i)}) \\ \mathsf{A}^{\mathrm{set}}(C^r_k, a_j) & = & \mathsf{reduce}(\overrightarrow{l^-_{kj}(t'_i)}, \overrightarrow{l^-_{kj}(t_i)}) \end{array}$$

First, we notice that reduce is the left inverse of expand.

Lemma 6.6.8 *Let $r \in \mathscr{V}^{\mathscr{C}}$. Then $\mathsf{reduce}(\mathsf{expand}(r)) = r$.*

Proof. Let $\mathsf{expand}(r) = (\vec{r^+}, \vec{r^-})$ and $\mathsf{reduce}(\vec{r^+}, \vec{r^-}) = s$. We show $r = s$ by coinduction.

1. We show
$$\begin{array}{lcll} \langle\!\langle C^r_i \rangle\!\rangle & = & \langle\!\langle C^s_i \rangle\!\rangle & \Longleftrightarrow \\ & = & \langle\!\langle {r_i}^+ \rangle\!\rangle \dot{-} \langle\!\langle {r_i}^- \rangle\!\rangle & \Longleftrightarrow \\ & = & \langle\!\langle C^r_i \rangle\!\rangle \dot{-} 0 & \end{array}$$
which follows trivially.

2. We show
$$\begin{array}{lcll} \mathsf{A}^{\mathrm{get}}(C^r_k, a_j) & = & \mathsf{A}^{\mathrm{get}}(C^s_k, a_j) & \Longleftrightarrow \\ & = & \mathsf{reduce}(\overrightarrow{l^+_{kj}({r_i}^+)}, \overrightarrow{l^+_{kj}({r_i}^-)}) & \Longleftrightarrow \\ & = & \mathsf{reduce}(\overrightarrow{\mathsf{A}^{\mathrm{get}}(C^r_k, a_j)^+_i}, \overrightarrow{\mathsf{A}^{\mathrm{get}}(C^r_k, a_j)^-_i}) & \end{array}$$
which follows by the coinduction hypothesis.

3. We show
$$\begin{array}{lcll} \mathsf{A}^{\mathrm{set}}(C^r_k, a_j) & = & \mathsf{A}^{\mathrm{set}}(C^s_k, a_j) & \Longleftrightarrow \\ & = & \mathsf{reduce}(\overrightarrow{l^-_{kj}({r_i}^-)}, \overrightarrow{l^-_{kj}({r_i}^+)}) & \Longleftrightarrow \\ & = & \mathsf{reduce}(\overrightarrow{\mathsf{A}^{\mathrm{set}}(C^r_k, a_j)^+_i}, \overrightarrow{\mathsf{A}^{\mathrm{set}}(C^r_k, a_j)^-_i}) & \end{array}$$
which follows by the coinduction hypothesis.

□

Next, we prove the required property (6.6.3).

Lemma 6.6.9 *Let $\vec{t}$, $\vec{t'}$, $\vec{p}$, $\vec{p'}$ be vectors of infinite trees. Then, if $t'_i \sqsubseteq p'_i$ and $p_i \sqsubseteq t_i$ for each i, then $r = \mathsf{reduce}(\vec{t}, \vec{t'}) \sqsubseteq \mathsf{reduce}(\vec{p}, \vec{p'}) = s$.*

Proof.

- We show
$$\begin{array}{lcl} \Diamond(C^r_i) & \geq & \Diamond(C^s_i) \\ \Diamond(t_i) \dot{-} \Diamond(t'_i) & \geq & \Diamond(p_i) \dot{-} \Diamond(p'_i) \end{array}$$
and this follows from $\Diamond(t'_i) \leq \Diamond(p'_i)$ and $\Diamond(t_i) \geq \Diamond(p_i)$, which follows by assumption.

- We show
$$\begin{array}{lcl} \mathsf{A}^{\mathrm{get}}(C^r_k, a_j) & \sqsubseteq & \mathsf{A}^{\mathrm{get}}(C^s_k, a_j) \iff \\ \mathsf{reduce}(\overrightarrow{l^+_{kj}(t_i)}, \overrightarrow{l^+_{kj}(t'_i)}) & \sqsubseteq & \mathsf{reduce}(\overrightarrow{l^+_{kj}(p_i)}, \overrightarrow{l^+_{kj}(p'_i)}) \end{array}$$
which follows by the coinduction hypothesis, because we have by assumption $l^+_{kj}(p_i) \sqsubseteq l^+_{kj}(t_i)$ and $l^+_{kj}(t'_i) \sqsubseteq l^+_{kj}(p'_i)$.

- We show
$$\begin{array}{lcl} \mathsf{A}^{\mathrm{set}}(C^s_k, a_j) & \sqsubseteq & \mathsf{A}^{\mathrm{set}}(C^r_k, a_j) \iff \\ \mathsf{reduce}(\overrightarrow{l^+_{kj}(p'_i)}, \overrightarrow{l^+_{kj}(p_i)}) & \sqsubseteq & \mathsf{reduce}(\overrightarrow{l^+_{kj}(t'_i)}, \overrightarrow{l^+_{kj}(t_i)}) \end{array}$$
which follows by the coinduction hypothesis, because we have by assumption $l^+_{kj}(t'_i) \sqsubseteq l^+_{kj}(p'_i)$ and $l^+_{kj}(p_i) \sqsubseteq l^+_{kj}(t_i)$.

□

The previous property holds when reduce is applied to *any* two vectors of trees. The following property, however, holds only when reduce is applied to a vector of positive trees and a vector of negative trees.

Lemma 6.6.10 *Let $\vec{t}, \vec{t'}, \vec{p}, \vec{p'}$ be vectors of infinite trees, where each t_i, p_i is positive and each t'_i, p'_i is negative. Then:*

1. $\mathsf{reduce}(\vec{t}, \vec{t'}) \oplus \mathsf{reduce}(\vec{p}, \vec{p'}) = \mathsf{reduce}(\overrightarrow{t_i + p_i}, \overrightarrow{t'_i \wedge p'_i})$.

2. $\mathsf{reduce}(\vec{t}, \vec{t'}) \boxplus \mathsf{reduce}(\vec{p}, \vec{p'}) = \mathsf{reduce}(\overrightarrow{t_i \wedge p_i}, \overrightarrow{t'_i + p'_i})$.

Proof. Simultaneously by coinduction. □

6.6.3 Subtyping and arithmetic constraints to systems of constraints

Now we are ready to present the reduction from subtyping and arithmetic constraints to systems of constraints. The reduction is based on the reduction from views to infinite trees and on the reduction from infinite trees to views, which we described in the previous sections.

Given a conjunction of subtyping and arithmetic constraints $\mathcal{C}$, for each view variable $v \in \mathsf{Vars}(\mathcal{C})$ and class $C_i \in \mathscr{C}$, we introduce view variables v_i^+ and v_i^-. Moreover, we reduce each subtyping constraint $C^{\mathsf{vexp}} <: D^{\mathsf{vexp}'}$ to an FJEU subtyping statement $C <: D$ and the constraints $\mathsf{vexp}'^+_i \sqsubseteq \mathsf{vexp}^+_i$ and $\mathsf{vexp}^-_i \sqsubseteq \mathsf{vexp}'^-_i$, where vexp^+_i and vexp^-_i are defined inductively[1] in Fig. 6.13. Further, we require that each value for the variable v_i^+ must be a positive tree and that each value for the variable v_i^- must be a negative tree. Here we do not specify how we ensure this; we shall see later how we make sure that this requirement holds when solving systems of constraints by using the algorithm from Fig. 6.11.

We also reduce each linear arithmetic constraint $\mathsf{ae}_1 \leq \mathsf{ae}_2$ to $\mathsf{ae}_1^+ \leq \mathsf{ae}_2^+$, where ae^+ is also defined inductively in Fig. 6.13. Then, based on those reductions, we build $\mathcal{C}^{\mathsf{tree}}$, a system of constraints, and $\mathcal{C}^{\mathsf{class}}$, a conjunction of FJEU subtyping statements (Fig. 6.13).

Notice that we assume that all the subtyping constraints are of the form $C^{\mathsf{vexp}} <: D^{\mathsf{vexp}'}$, despite of the fact that constraints over views; that is, constraints of the form $\mathsf{vexp} \sqsubseteq \mathsf{vexp}'$ are also allowed. We ignore this fact here for ease of notation; we can always extend constraints over views to equivalent subtyping constraints by using the same class as a base type on both sides of the constraint. Equalities over views ($\mathsf{vexp}_1 = \mathsf{vexp}_2$) are also allowed, but they stand for the two subtyping constraints $\mathsf{vexp}_1 \sqsubseteq \mathsf{vexp}_2$ and $\mathsf{vexp}_2 \sqsubseteq \mathsf{vexp}_1$.

We also assume that the arithmetic constraints are of the form $\mathsf{ae}_1 \leq \mathsf{ae}_2$, although constraints of the form $\mathsf{ae}_1 \geq \mathsf{ae}_2$ are allowed as well. But it is clear how to bring those constraints into this canonical form: $\mathsf{ae}_1 \leq \mathsf{ae}_2 \iff \mathsf{ae}_2 \geq \mathsf{ae}_1$.

[1] We define $(v \oplus v')^-_i$ to be $v_i^- \wedge v'^-_i$, although we have not defined an expression $\mathsf{te}_1 \wedge \mathsf{te}_2$. However, since expressions $v_1 \oplus v_2$ occur only on the right hand side of constraints (Fig. 5.2), then a constraint $u \sqsubseteq v \oplus v'$ reduces to the constraint $u_i^- \sqsubseteq v_i^- \wedge v_i'^-$, which is equivalent to $u_i^- \sqsubseteq v_i^-$ and $u_i^- \sqsubseteq v_i'^-$, by the l.u.b. property.

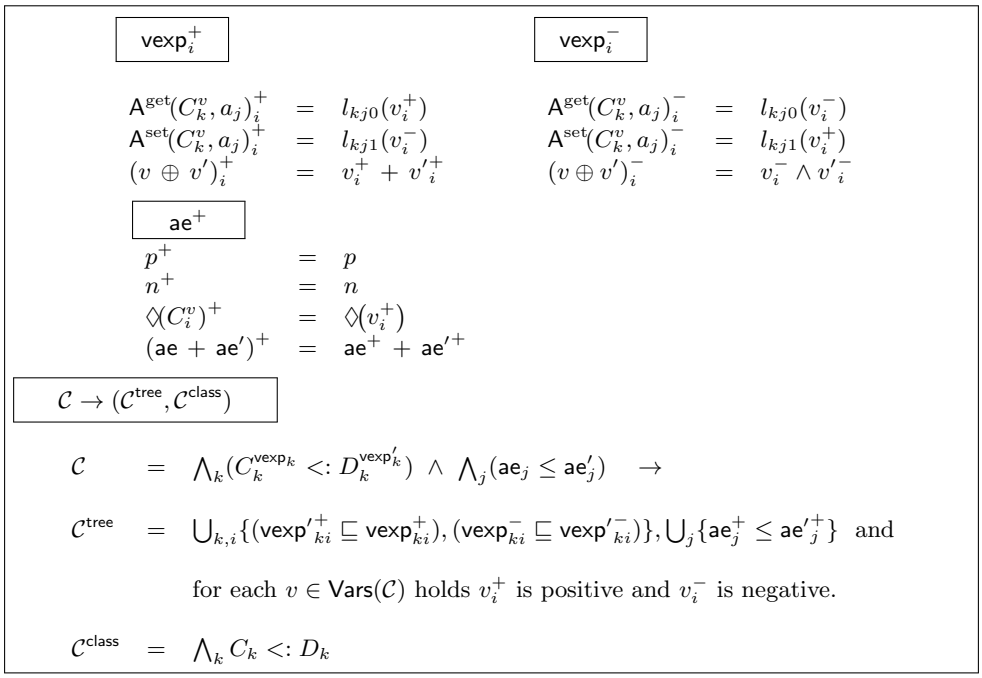


Figure 6.13: Reducing a conjunction of subtyping and arithmetic constraints to a system of constraints.

Building a satisfying valuation for a system of constraints

When we are given a conjunction of subtyping and arithmetic constraints $\mathcal{C}$ and a valuation that satisfies the constraints, we can build another valuation in such a way that, if we reduce $\mathcal{C}$ to a system of constraints $\mathcal{C}^{\mathsf{tree}}$, the new valuation satisfies $\mathcal{C}^{\mathsf{tree}}$.

Definition 6.6.11 *Let $\pi = (\pi_v, \pi_a)$ be a valuation from view variables to views and arithmetic variables to numbers. Then, set*

$$\mathrm{dom}(\pi_i^+) = \{v_{ji}^+ \in X \mid \textit{for each } v_j \in \mathrm{dom}(\pi) \textit{ and each } C_i \in \mathcal{C}\}$$

and set $\pi_i^+(v_{ji}^+) = r_{ji}^+$, if $\pi(v_j) = r$. Analogously set

$$\mathrm{dom}(\pi_i^-) = \{v_{ji}^- \in X \mid \textit{for each } v_j \in \mathrm{dom}(\pi) \textit{ and each } C_i \in \mathcal{C}\}$$

and set $\pi_i^-(v_{ji}^-) = r_{ji}^-$, if $\pi(v_j) = r$. Finally, set $\pi^{+/-} = (\bigcup_i \pi_i^+ \cup \bigcup_i \pi_i^-, \pi_a)$.

Lemma 6.6.12 *Let* vexp *be a view expression and exp be an arithmetic expression and* π *be a valuation. Then*

1. $\pi^{+/-}(\mathsf{vexp}_i^+) = \pi(\mathsf{vexp})_i^+$.
2. $\pi^{+/-}(\mathsf{vexp}_i^-) = \pi(\mathsf{vexp})_i^-$.
3. $\pi^{+/-}(\mathsf{ae}^+) = \pi(\mathsf{ae})$.

Proof. 1. and 2. Simultaneously by induction on vexp.

1. *Case* $\mathsf{vexp} = v$. We show $\pi^{+/-}(v_i^+) = \pi(v)_i^+$, which follows by definition of $\pi^{+/-}$.

   Case $\mathsf{vexp} = \mathsf{A}^{\mathrm{get}}(C_k^v, a_j)$. We have

$$\begin{aligned}
\pi^{+/-}(\mathsf{A}^{\mathrm{get}}(C_k^v, a_j)_i^+) &=_{\text{Fig. 6.13}} && \pi^{+/-}(l_{kj}^+(v_i^+)) \\
&= && l_{kj}^+(\pi^{+/-}(v_i^+)) \\
&=_{\text{I.H. 1.}} && l_{kj}^+(\pi(v)_i^+) \\
&=_{\text{Def. 6.6.1}} && \mathsf{A}^{\mathrm{get}}\Big(C_k^{\pi(v)}, a_j\Big)_i^+ \\
&= && \pi(\mathsf{A}^{\mathrm{get}}(C_k^v, a_j))_i^+
\end{aligned}$$

   Case $\mathsf{vexp} = \mathsf{A}^{\mathrm{set}}(C_k^v, a_j)$. Similar.

   Case $\mathsf{vexp} = v_1 \oplus v_2$. We have

$$\begin{aligned}
\pi^{+/-}((v_1 \oplus v_2)_i^+) &=_{\text{Fig. 6.13}} && \pi^{+/-}({v_1}_i^+ + {v_2}_i^+) \\
&= && \pi^{+/-}({v_1}_i^+) + \pi^{+/-}({v_2}_i^+) \\
&=_{\text{I.H. 1.}} && \pi(v_1)_i^+ + \pi(v_2)_i^+ \\
&=_{\text{Lem. 6.6.6}} && (\pi(v_1) \oplus \pi(v_2))_i^+ \\
&= && \pi(v_1 \oplus v_2)_i^+
\end{aligned}$$

2. Very similar to the previous case.
3. By induction on ae.

□

Lemma 6.6.13 *Let π be a valuation and let $C^{\mathsf{vexp}} <: D^{\mathsf{vexp}'}$ be a subtyping constraint and $\mathsf{ae} \leq \mathsf{ae}'$ be an arithmetic constraint and v be a view variable.*

1. *If $\pi \models C^{\mathsf{vexp}} <: D^{\mathsf{vexp}'}$ then $\pi^{+/-} \models (\mathsf{vexp}'^{+} \sqsubseteq \mathsf{vexp}^{+})$ and $\pi^{+/-} \models (\mathsf{vexp}^{-} \sqsubseteq \mathsf{vexp}'^{-})$ and $C <: D$.*
2. *If $\pi \models \mathsf{ae} \leq \mathsf{ae}'$ then $\pi^{+/-} \models \mathsf{ae}^{+} \leq \mathsf{ae}'^{+}$.*
3. *$\pi^{+/-}(v_i^{+})$ positive and $\pi^{+/-}(v_i^{-})$ is negative.*

Proof.

1. Let $\pi(\mathsf{vexp}) \sqsubseteq \pi(\mathsf{vexp}')$. We show

$$\begin{array}{lcl} \pi^{+/-}(\mathsf{vexp}'^{+}) & \sqsubseteq & \pi^{+/-}(\mathsf{vexp}^{+}) \text{ and} \\ \pi^{+/-}(\mathsf{vexp}^{-}) & \sqsubseteq & \pi^{+/-}(\mathsf{vexp}'^{-}) \iff \text{Lem. 6.6.12} \end{array}$$

$$\begin{array}{lcl} \pi(\mathsf{vexp}')^{+} & \sqsubseteq & \pi(\mathsf{vexp})^{+} \text{ and} \\ \pi(\mathsf{vexp})^{-} & \sqsubseteq & \pi(\mathsf{vexp}')^{-} \end{array}$$

   which follows by Lemma 6.6.3.

2. Follows by Lemma 6.6.12, item 3.
3. Follows by definition of $\pi^{+/-}$.

□

Building a satisfying valuation for a conjunction of constraints

Now we wish to give a satisfying valuation for a conjunction of subtyping and arithmetic constraints, based on a valuation that satisfies a system of constraints.

Definition 6.6.14 *Let $\pi = (\pi_t, \pi_a)$ be a valuation where π_t is a map from tree variables to trees and π_a is a map from arithmetic variables to numbers. Moreover let $\mathcal{V}$ be a set of view variables such that*

$$\mathrm{dom}(\pi_t) = \{v_i^{+}, v_i^{-} \mid v \in \mathcal{V}\}$$

Then, we set $\mathrm{dom}(\pi_v) = \mathcal{V}$ *and set*

$$\pi_v(v) = \mathsf{reduce}(\overrightarrow{\pi_t(v_i^{+})}, \overrightarrow{\pi_t(v_i^{-})})$$

Finally, set $\pi^{\mathsf{v}} = (\pi_v, \pi_a)$.

Lemma 6.6.15 *Let* vexp *be a view expression and* ae *be an arithmetic expression and let* π *be a valuation such that for each* $v \in \mathsf{Vars}(\mathsf{vexp}) \cup \mathsf{Vars}(\mathsf{ae})$ *holds* $\pi(v_i^+)$ *is a positive tree and* $\pi(v_i^-)$ *is a negative tree. Then:*

1. $\pi^{\mathsf{v}}(\mathsf{vexp}) = \mathsf{reduce}(\overrightarrow{\pi(\mathsf{vexp}_i^+)}, \overrightarrow{\pi(\mathsf{vexp}_i^-)})$.
2. $\pi^{\mathsf{v}}(\mathsf{ae}) = \pi(\mathsf{ae}_i^+)$.

Proof.

1. By induction on vexp.

   Case $\mathsf{vexp} = v$. We show $\pi^{\mathsf{v}}(v) = \mathsf{reduce}(\overrightarrow{\pi_t(v_i^+)}, \overrightarrow{\pi_t(v_i^-)})$, which follows by definition of π^{v}.

   Case $\mathsf{vexp} = \mathsf{A}^{\mathrm{get}}(C_k^v, a_j)$. We have

$$\begin{array}{lll}
\pi^{\mathsf{v}}(\mathsf{A}^{\mathrm{get}}(C_k^v, a_j)) & = & \mathsf{A}^{\mathrm{get}}\Big(C_k^{\pi^{\mathsf{v}}(v)}, a_j\Big) \\
& =_{\text{I.H.}} & \mathsf{A}^{\mathrm{get}}\Big(C_k^{\mathsf{reduce}(\pi(v_i^+),\pi(v_i^-))}, a_j\Big) \\
& =_{\text{Def 6.6.14}} & \mathsf{reduce}(\overrightarrow{l_{kj}^+(\pi(v_i^+))}, \overrightarrow{l_{kj}^+(\pi(v_i^-))}) \\
& =_{\text{Fig 6.13}} & \mathsf{reduce}(\overrightarrow{\pi(\mathsf{A}^{\mathrm{get}}(C_k^v, a_j)_i^+)}, \overrightarrow{\pi(\mathsf{A}^{\mathrm{get}}(C_k^v, a_j)_i^-)})
\end{array}$$

   Case $\mathsf{vexp} = \mathsf{A}^{\mathrm{set}}(C_k^v, a_j)$. Similar.

   Case $\mathsf{vexp} = v_1 \oplus v_2$. We have

$$\begin{array}{lll}
\pi^{\mathsf{v}}(v_1 \oplus v_2) & = & \pi^{\mathsf{v}}(v_1) \oplus \pi^{\mathsf{v}}(v_2) \\
& =_{\text{I.H.}} & \mathsf{reduce}(\overrightarrow{\pi(v_{1i}^+)}, \overrightarrow{\pi(v_{1i}^-)}) \oplus \mathsf{reduce}(\overrightarrow{\pi(v_{2i}^+)}, \overrightarrow{\pi(v_{2i}^-)}) \\
& =_{\text{Lem 6.6.10}} & \mathsf{reduce}(\overrightarrow{\pi(v_{1i}^+) + \pi(v_{2i}^+)}, \overrightarrow{\pi(v_{1i}^-) \wedge \pi(v_{2i}^-)}) \\
& =_{\text{Fig 6.13}} & \mathsf{reduce}(\overrightarrow{\pi((v_1 \oplus v_2)_i^+)}, \overrightarrow{\pi((v_1 \oplus v_2)_i^-)})
\end{array}$$

2. By induction on ae.

   Case $\Diamond(C^v)$. We have

$$\begin{array}{lll}
\pi^{\mathsf{v}}(\Diamond(C_i^v)) & = & \Diamond\Big(C_i^{\pi^{\mathsf{v}}(v)}\Big) \\
& =_{\text{Def. 6.6.14}} & \Diamond\left(C_i^{\mathsf{reduce}(\pi(v_i^+),\pi(v_i^-))}\right) \\
& =_{\text{Def. 6.6.7}} & \Diamond\big(\pi(v_i^+)\big) \dot{-} \Diamond\big(\pi(v_i^-)\big) \\
& =_{\text{assumption}} & \Diamond\big(\pi(v_i^+)\big) \dot{-} 0 \\
& = & \Diamond\big(\pi(v_i^+)\big) \\
& = & \pi(\Diamond\big(v_i^+\big))
\end{array}$$

   □

Lemma 6.6.16 *Let $\mathcal{C}$ be a conjunction of subtyping and arithmetic constraints and let π be a valuation such that, for each $v \in \mathsf{Vars}(\mathcal{C})$ holds $\pi(v_i^+)$ is positive and $\pi(v_i^-)$ is negative.*

1. *Let $\mathsf{vexp} \sqsubseteq \mathsf{vexp}'$ be a constraint in $\mathcal{C}$. Then, if $\pi \models \{\mathsf{vexp}'^+_i \sqsubseteq \mathsf{vexp}^+_i, \mathsf{vexp}^-_i \sqsubseteq \mathsf{vexp}'^-_i\}$, then $\pi^{\mathsf{v}} \models \mathsf{vexp} \sqsubseteq \mathsf{vexp}'$.*

2. *Let $\mathsf{ae} \leq \mathsf{ae}'$ be a constraint in $\mathcal{C}$. Then, if $\pi \models \mathsf{ae}^+ \leq \mathsf{ae}'^+$ then $\pi^{\mathsf{v}} \models \mathsf{ae} \leq \mathsf{ae}'$.*

Proof.

1. Follows by Lemma 6.6.15 and Lemma 6.6.9.

2. Follows by Lemma 6.6.15.

□

Theorem 6.6.17 (Soundness and completeness of the reduction) *Let $\mathcal{C}$ be a conjunction of subtyping and arithmetic constraints. Then there exists a valuation π with $\pi \models \mathcal{C}$ iff there exists a valuation π' with $\pi' \models \mathcal{C}^{\mathsf{tree}}$ such that, for each $v \in \mathsf{Vars}(\mathcal{C})$ holds $\pi'(v_i^+)$ is positive and $\pi'(v_i^-)$ is negative and $\mathcal{C}^{\mathsf{class}}$ holds.*

Proof.

Case "⇒" Follows by Lemma 6.6.13.

Case "⇐" Follows by Lemma 6.6.16.

□

6.6.4 Algorithm for solving subtyping and arithmetic constraints

In this section we shall present a heuristic algorithm for solving a conjunction of subtyping and arithmetic constraints, by first reducing the constraints to systems of constraints, and then applying the heuristic algorithm for solving systems of constraints, described in Section 6.5.4. The algorithm consists of the steps shown in Fig. 6.14.

First, we reduce the conjunction of subtyping and arithmetic constraints $\mathcal{C}$ to the system of constraints $\mathcal{C}^{\mathsf{tree}}$ and the set of subtyping judgements $\mathcal{C}^{\mathsf{class}}$. Then, we create a set of negative variables X^- and add each v_i^- to it.

For proving the soundness of the reduction (Theorem 6.6.17) we require that the values for the positive tree variables v_i^+ are positive trees and that the values for the negative tree variables v_i^- are negative trees. Recall that an infinite tree is positive if some of its subtrees are positive (the ones that we obtain by applying the labels in $\mathcal{L}^+$), and all other subtrees are negative. Moreover, a tree is negative if its root node is labelled with the number 0, and moreover, some of its subtrees are negative (again, the ones that we obtain

Heuristic algorithm for solving a conjunction of constraints $\mathcal{C}$.

1. Create $\mathcal{C}^{\mathsf{tree}}$ and $\mathcal{C}^{\mathsf{class}}$.
2. Build a set X^- of negative tree variables; add each v_i^- to X^-, where $v \in \mathsf{Vars}(\mathcal{C})$.
3. Eliminate variables from $\mathcal{C}^{\mathsf{tree}}$ (steps 1. and 2. from Fig. 6.11) and try to create linear loops (step 3. from Fig. 6.11)). For each new variable z that is created by the rule ($\triangleright\,\mathsf{Elim}^{+/-}$) (or rule ($\triangleright\,\mathsf{toLinearLoop}$)), if we substitute a negative path with z, then add z to X^-. Moreover, for each new arithmetic variable λ that is created in the rule ($\triangleright\,\mathsf{Elim}^{+/-}$) (or rule ($\triangleright\,\mathsf{toLinearLoop}$)), if we substitute $\Diamond\!\!\!\!-(x)$ with z, and $x \in X^-$, then add the constraint $\lambda = 0$ to $\mathcal{C}^{\mathsf{tree}}$.
4. Obtain a tree schema Ts by applying the step 4. from Fig. 6.11.
5. Create $\Delta_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$ by iteration, where in each iteration step, whenever we substitute an atomic expression tae with a variable x, if tae is a negative path, we add x to X^-.
6. For each variable $x \in X^-$, add the constraint $\Diamond\!\!\!\!-(x) = 0$ to $\Delta_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$, and obtain $\Delta^{+/-}_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$.
7. Continue with items 6. and 7. from Fig. 6.11, using the extended set $\Delta^{+/-}_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$, and obtain a valuation π.

Figure 6.14: Heuristic algorithm for solving a conjunction of subtyping and arithmetic constraints $\mathcal{C}$.

by applying the labels in $\mathcal{L}^+$), and all others are positive. Thus, we can force infinite trees to be positive or negative by requiring the appropriate nodes to be labelled with 0. For instance, we must require $\Diamond\!\!\!\!-\big(v_i^-\big) = 0$ and $\Diamond\!\!\!\!-\big(l^-(v_i^+)\big) = 0$, and so on. That is why we modify the elimination and iteration procedures and the procedure for creating linear loops slightly (Fig. 6.14): Whenever we unfold the constraints over trees, or create new tree variables, we need to keep track of which new variables or paths are negative.

Then, we enrich $\Delta_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$ with the constraints $\Diamond\!\!\!\!-(x) = 0$, for each negative variable $x \in X^-$. Finally, we continue with the steps six and seven from the algorithm from Fig. 6.11; that is, we attempt to solve the set of constraints $\Delta_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$ enriched with the negativity constraints, using an LP-Solver. If we obtain a valuation π, we then create regular trees for the tree variables, by combining π with the tree schema.

Theorem 6.6.18 (Soundness of the algorithm from Fig. 6.14)
Let $\mathcal{C}$ be a conjunction of subtyping and arithmetic constraints and let π be the valuation given by the algorithm described in Fig. 6.14 and let $\mathcal{C}^{\mathsf{class}}$ hold. Then, there exists $\pi' \supseteq \pi$ with $\pi' \models \mathcal{C}$.

Proof sketch. By Theorem 6.5.21 we have there exists $\pi' \supseteq \pi$ such that $\pi \models \mathcal{C}^{\mathsf{tree}}$. Moreover, the fact that for each $v \in \mathsf{Vars}(\mathcal{C})$ holds $\pi'(v_i^+)$ is positive and $\pi'(v_i^-)$ is negative follows by modifications to the elimination and iteration procedures and the procedure for creating linear loops and by the extension of $\Delta_{\mathsf{Ts}}(\mathcal{C}^{\mathsf{tree}})$ with the negativity constraints. Then, by Theorem 6.6.17, $\pi'^{\mathsf{v}} \models \mathcal{C}$. □

Chapter 7

Prototype Implementation

This chapter describes a prototype implementation of the algorithms presented in this thesis. Section 7.1 describes a memory aware interpreter for FJEU programs. Section 7.1.1 shows how to compile and to use the tool, and Section 7.1.2 describes the analyser module with all its sub-modules. Finally, Section 7.2 describes various programs that we could successfully analyse with the tool.

7.1 Memory aware interpreter for FJEU programs

I have implemented a tool in OCaml for type checking and evaluating FJEU programs. The main module of the tool is an analyser of the heap-space requirements of the code, which is based on the algorithms presented in this thesis. The tool uses the result of the analysis for building an optimised heap for evaluating the programs; that is, it creates a heap with a size equal to the size predicted by the analysis.

The tool assumes that each FJEU program contains a main method which has one parameter of type List. It also assumes that the program contains the classes List, Cons and Nil which define singly linked lists. Further, the tool assumes that it is given an input file for the program execution. The interpreter then creates a singly linked list (one node for each row of the input file) and saves it in the heap before it starts executing the program.

Thus, if we obtain an upper bound to the heap-space consumption of the main method, as a function of the length of main's argument, then it is also an upper-bound to the heap-space requirements of the program, as a function of the size of the input file.

The analyser component of the tool can analyse methods whose heap-space consumption is a linear function on the size of its arguments. When it analyses the main method of a program, it delivers two nonnegative real numbers a and b, which shall mean that the program can be evaluated with

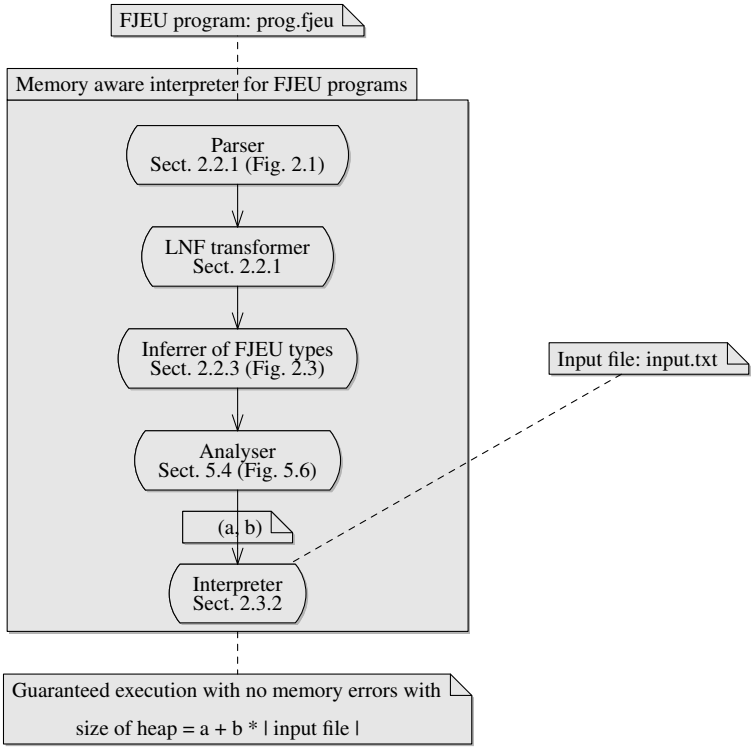


Figure 7.1: Schematic structure of the implementation of a memory aware interpreter for FJEU programs.

no memory errors with a heap of size at least

$$a + b \cdot |\text{input file}|$$

The tool consists of 8473 lines of code and is organised in different modules which we shall describe in the following. Fig. 7.1 shows the modules and how they interact.

Parser The parser module consists of a parser for FJEU programs, built with the help of the parser-generator Ocamlyacc[1]; and a scanner, built with the scanner-generator Ocamllex[2]. The parser takes an FJEU file and delivers a list of class declarations. It accepts FJEU programs defined by the grammar described in Fig. 2.1. However, it also accepts programs that are not in let normal form, because we also provide a module for transforming expressions into let normal form. Moreover, it accept not only classes as types for variables and fields, but also basic

[1] http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial
[2] http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/

data types, such as int, string and bool. Further, the parser accepts arithmetic expressions, such as $x + y$ or $x * y$, and boolean expressions, such as $x == y$ or $x < y$.

LNF transformer This module takes a list of class declarations and delivers a modified version of this list, where all the expressions in the methods bodies are in let normal form. The transformation consists of creating let expressions for each sub-expression in nested expressions. For instance, the expression $x.a.b$ is transformed into let $y = x.a$ in $y.b$. Notice that, although this transformation increases the size of the program, it does not change its heap-space requirements.

Inferrer of FJEU types The inferrer module takes a list of class declarations, where all expressions are in let normal form, and infers the types for the variables in let expressions, by using the algorithm described in Fig. 2.3 of Section 2.2.3.

Analyser The analyser module takes a list of class declarations, where all the expressions are in let normal form, and all the variables in let expressions are annotated with a type, and returns two nonnegative real numbers a and b. These numbers are the parameters of the linear function $f(n) = a + b \cdot n$ that describes the heap-space consumption of the program, where n is the length of the input file. The module implements the algorithms for constraint generation and solving described in the previous chapters. It shall be described in detail in the following section.

Interpreter The interpreter module takes a list of class declarations, the parameters a and b delivered by the analyser, and an input file. It then builds a heap of size $a + b \cdot |\text{input file}|$; the soundness of the analysis guarantees that a heap of that size will be sufficient for evaluating the program with no risk of running out of memory. This module implements the operational semantics rules for FJEU programs described in Fig. 2.5 of Section 2.3.2. As described in that section, the interpreter also keeps counters (non-negative natural numbers m and m') for keeping track of the number of unused heap units before and after evaluating an expression, respectively. This way it can display the real amount of heap units used by the program, so that it can be compared to the amount predicted by the analysis.

7.1.1 Usage

The OCaml files that compose the analyser and interpreter tool can be compiled using a makefile that we also provide. If you execute the command

```
> make
```

then the files will be compiled with the OCaml native-code compiler `ocamlopt`[3], which creates the executable file `raja`.

We provide various command line options that allow to customise the tool. For type checking, analysing and evaluating an FJEU program you can execute the following command:

```
> ./raja <fjeu file> -d <working directory> -i <input file>
```

The tool will search for the given FJEU file in the given working directory. If you only wish to analyse the program but not to evaluate it, then you can achieve this with the command:

```
> ./raja <fjeu file> -d <working directory> -a
```

On the other hand, if you only wish to type check and to evaluate the program, but not to analyse it, then you can type:

```
> ./raja <fjeu file> -d <working directory> -i <input file> -r
```

We provide more command line options: If the option `-h` is provided, and the option `-a` is not provided; that is, the program will be evaluated, then the last heap configuration will be printed. Finally, if the option `-help` is provided, a list of options will be displayed.

You can download the source code from the RAJA web site[4].

7.1.2 Analyser module

In this section we shall explain the analyser module in more detail. This module is the most complex one, comprising several sub-modules. Fig 7.2 shows the sub-modules and how they relate.

Constraint generator The first step of the analysis is the generation of subtyping and arithmetic constraints. This module implements the constraint generation rules described in the Figures 5.2 and 5.3 from Section 5.3. As mentioned before, in the implementation we admit basic data types as well as class types. Thus, in the rules for generating constraints, we need to check whether the types involved are class types or not, and only when they are class types, we take them into account when collecting constraints.

Optimisations. For increasing the efficiency of the analysis, it is useful to eliminate tree variables from the constraints as soon as possible; that is, as soon as the generator creates all the constraints that contain one particular variable, it can eliminate that variable from the

[3] http://caml.inria.fr/, OCaml version 3.11.2.
[4] http://raja.tcs.ifi.lmu.de/download

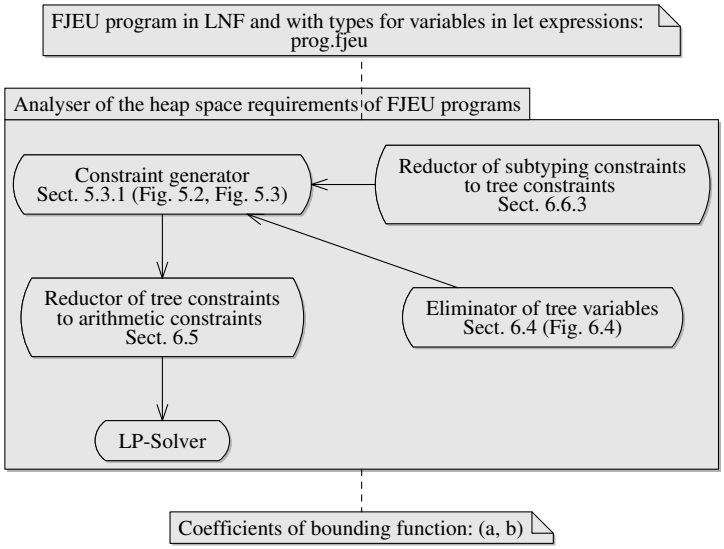


Figure 7.2: Schematic structure of the implementation of the analyser.

constraints. For instance, in the rule (∇Let) it can eliminate the variables u, $\vec{w}$ and $\vec{u}$. This reduces the amount of variables and the amount of constraints. Still, there is one problem: the rules generate subtyping constraints, but the elimination procedure works for tree constraints. The solution is to adapt the constraint generation rules, such that they reduce the subtyping constraints to tree constraints, and deliver those. So, the polymorphic RAJA method types in our implementation consist of view and arithmetic variables and systems of constraints (tree constraints and arithmetic constraints). We relate the view variables to the constraints over trees by implementing tree variables as tuples $(v, \mathsf{cl}, \mathsf{sign})$, where v is a view variable, cl is an FJEU class, and sign is either $+$ or $-$.

Reductor of subtyping constraints to tree constraints This module implements the algorithm described in Fig. 6.13 of Section 6.6.3. As mentioned before, the constraint generator module uses this module for reducing subtyping constraints to tree constraints in each rule.

Eliminator of tree variables This module implements the algorithm from Fig. 6.4 in Section 6.4. The algorithm is not complete, so some variables cannot be eliminated. However, most variables can be eliminated, which increases the efficiency of the analysis significantly. For this reason, the main factor in the complexity of the analysis is the complexity of the elimination procedure.

Optimisations. For choosing which rule should be applied, the elim-

inator needs to check whether the variable to be eliminated appears positively or negatively in each of the tree and arithmetic constraints. However, if an arithmetic constraint contains no tree expressions, the eliminator does not need to check it. Thus, for avoiding unnecessary checks, we keep the arithmetic constraints in two sets: one set for the constraints that contain tree expressions, and another set for the constraints that do not contain tree expressions.

There is certainly room for more optimisations in this procedure.

Reductor of tree constraints to arithmetic constraints After the constraints of all methods in the program have been generated, the next step of the analysis is to solve the system of constraints from the polymorphic RAJA method type of main. For doing that, we follow the heuristic algorithm described in Fig. 6.11 from Section 6.5.4; that is, we build a tree schema and obtain a set of arithmetic constraints by the iteration procedure.

LP-Solver The last step consists of solving the arithmetic constraints, by using an LP-Solver. The module LP-Solver consists of a simple interface to the LP-Solver `lp_solve`[5]. We build a model file `model.lp`, which contains the arithmetic constraints in a very readable format and call `lp_solve` with it. The LP-Solver returns either a valuation π in the file `output.txt`, or returns that the problem is infeasible.

Because our algorithm is not complete, the latter case can mean different things: either that we were not able to find the right tree schema, or that the heap-space consumption of the program cannot be described as a linear function, or that the program is not typeable in our type system $\mathrm{RAJA}^{\mathrm{m}}$, or that there is a memory leak; that is, there is a programming error and the program needs infinite amount of heap-space.

On the other hand, if the LP-Solver returns a valuation, then we can obtain the two parameters a and b as follows. Assume that the polymorphic RAJA method type of main is defined by:

$$\forall \mathsf{v}_{\mathsf{self}}, \mathsf{v}_{\mathsf{l}}, \mathsf{v}_{\mathsf{res}}\, \exists \vec{w}\, \vec{t}\,.\, C^{\mathsf{v}_{\mathsf{self}}}; \mathsf{List}^{\mathsf{v}_{\mathsf{l}}} \xrightarrow{q_1/q_2} D^{\mathsf{v}_{\mathsf{res}}} \;\&\; \mathcal{C}$$

where C and D are classes from the FJEU program. Then, the arithmetic constraints that the LP-Solver tried to solve contain the variables $\mathsf{vl}^{+}_{\mathsf{Cons}}$, $\mathsf{vl}^{+}_{\mathsf{Nil}}$, $\mathsf{v}_{\mathsf{l}}{}^{+}_{\mathsf{List}}$ and q_1 and q_2. The variable $\mathsf{v}_{\mathsf{l}}{}^{+}_{\mathsf{cl}}$ with $\mathsf{cl} \in \{\mathsf{Cons}, \mathsf{List}, \mathsf{Nil}\}$ corresponds to the arithmetic expression $\Diamond((\mathsf{v}_{\mathsf{l}}, \mathsf{cl}, +))$. Thus, by Theorem 6.6.17, a valuation π^{v} where $\pi^{\mathsf{v}}(v_l) = r_l$, and r_l is

[5] http://lpsolve.sourceforge.net/5.5/

given by:

$$\begin{array}{lcl} \Diamond(\mathsf{Cons}^{r_l}) & = & \pi(\mathsf{vl}^{+}_{\mathsf{Cons}}) \\ \Diamond(\mathsf{List}^{r_l}) & = & \pi(\mathsf{vl}^{+}_{\mathsf{List}}) \\ \Diamond(\mathsf{Nil}^{r_l}) & = & \pi(\mathsf{vl}^{+}_{\mathsf{Nil}}) \\ \\ \mathsf{A}^{\mathrm{get}}(\,\mathsf{Cons}^{r_l},\,\mathsf{next}) & = & r_l \end{array}$$

and $\pi^{\mathsf{v}}(q_i) = \pi(q_i)$ satisfies the subtyping and arithmetic constraints of main. Thus, we can set the parameters n and m as follows:

$$a := \pi(q_1) + \pi(\mathsf{vl}^{+}_{\mathsf{Nil}}) \qquad b := \pi(\mathsf{vl}^{+}_{\mathsf{Cons}})$$

and the soundness of the analysis follows by Corollary 3.3.12. Moreover, to ensure that the upper bound is as tight as possible, we set the objective function for the LP-Solver as follows:

$$\min : \mathsf{v_l}^{+}_{\mathsf{Cons}}$$

That way, if the heap-space consumption of the program is constant, the valuation will return $\pi(\mathsf{v_l}^{+}_{\mathsf{Cons}}) = 0$, and the parameters n and m will build a constant function.

Notice that we ignore the view variable $\mathsf{v_{self}}$ in the analysis because it corresponds to the variable this, which cannot be used in the main method.

7.2 Experimental results

Fig. 7.3 shows some programs that we could analyse with our tool. For each example, we could solve the constraints and resultantly provide a (linear) upper bound for its heap-space requirements.

The experiments were performed on a 2.20GHz Intel(R) Core(TM)2 Duo CPU laptop with 2GB RAM. The runtime of the analysis varied from 0.2s to about 10 minutes on a program of 908 LoC.

The elimination of tree variables is the step that takes most of the time; we believe that there is room for improvement. We included the columns **Nr. of variables**, that shows the overall number of tree variables created when analysing the given program, and the column **Nr. of variables after elimination**, which shows the number of tree variables in the constraints of the main method after eliminating all the variables that can be eliminated. These data should illustrate that the elimination procedure reduces the amount of tree variables – and consequently the amount of constraints – significantly. Whereas the number of tree variables that were generated is proportional to the size of the programs, the number of variables that remain after the elimination reflects the amount of loops in the constraints and the amount of variables in the loops.

Program	LoC	Heap space	Nr. of variables	Nr. of variables after elimination	Run time
Copy	37	1 + n	362	8	0.2s
CircList	56	1 + n	1176	14	1.6s
ConstAppend	60	2 + 2n	666	16	0.7s
InsSort	66	2 + n	860	16	1.9s
DList	70	3 + n	880	14	1.2s
Append	80	2 + n	1512	20	3s
MergeSort	127	1	3038	22	10.3s
BankAccount	200	2 + 8n	3710	14	6.3s
Bank	908	11 + 6n	24038	67	9.8 min

Figure 7.3: Experimental results. The column **LoC** represents the lines of code of the program, the column **Heap-space** shows the result of the analysis: the prediction of the required size of the heap, which is in all cases equal to the actual heap-space consumption of the program. **Nr. of variables** represents the number of tree variables that the program creates when generating constraints. **Nr. of variables after elimination** represents the number of tree variables that appear in the constraints of main after eliminating all possible variables. **Run time** represents the run time of the analysis. n represents the size of the input.

The programs consist of list manipulations including copying a list (*Copy*), sorting a list (*InsSort*, *MergeSort*), converting a list into a doubly-linked list (*DList*), or to a circular list (*CircList*), or to a singly-linked list with a link to the last element (*ConstAppend*) or to a list of objects (*BankAccount*, *Bank*). All bounds are exact in our experiments although our soundness result only ensure an upper bound. There is a demo website where all the examples can be analysed and downloaded[6]. In the following we shall describe the programs in more detail.

Copy The program *Copy* has been our running example in this thesis and was described in Fig. 3.1. It takes a list and returns a copy of the list. It creates a new Cons object for each node of the list and a Nil object. Hence it needs $n + 1$ heap-cells for its execution.

Append We described the program *Append* in Fig. 3.2. In that program, the main function takes two lists and returns the list obtained by appending the second list to the first. The program we analysed is slightly different, since our interpreter assumes that main has only one argument of type List. It takes a list, copies it, and then appends the copy

[6] http://raja.tcs.ifi.lmu.de

to the original list. The program requires $1 + n$ heap-cells for copying the list and 1 heap-cell in the append method.

DList This program takes a singly-linked list and creates a doubly-linked list that contains exactly the elements of the input list. It then creates a singly-linked list again while it disposes the doubly-linked list. It creates a DCons object for each node of the input list and two DNil objects for creating both ends of the doubly-linked list. Later, it creates a Cons node for each node of the doubly-linked list after it disposes the corresponding DCons node. Finally, it creates a Nil node. Hence it needs $n + 3$ heap-cells for its execution.

CircList This program takes a singly-linked list and creates a circular list (a singly-linked list, whose last node is linked to the first one). Then, it linearises the circular list by adding a link from the last node to a Nil object. It creates a Cons object for each node of the input list and a Nil object. Thus, it needs $n + 1$ heap-cells for its execution. This program shows that our tool performs well in the presence of circular data.

ConstAppend The program *ConstAppend* takes a list and creates two singly-linked lists with a link to the last node of the list. Then it appends the second list to the first, which takes constant time due to the extra link to the last node of the list. The heap-space consumption of this program is $2 + 2n$ ($1 + n$ for each new list created).

InsSort The program *InsSort* implements the well-known sorting algorithm insertion sort. It takes a list of integers and delivers the sorted list. For sorting the list, it creates new nodes and inserts them in the right place in the new list. Thus, it takes $1 + n$ heap-cells for sorting the list and 1 heap-cell for creating a Main object. So, in total the program needs $2 + n$ heap-cells for its execution.

MergeSort The program *MergeSort* implements another well-known sorting algorithm: merge sort. The idea of the algorithm is to split the list in two halves and then to merge the two halves, and to do this recursively. In our implementation, when we split the list in two halves, we deallocate the original list. So the program only takes constant heap-space (1 heap-cell). The analysis shows that our tool performs well in the presence of deallocations.

BankAccount This program and the program *Bank* show the use of our tool in programs that contain various interacting objects. The program *BankAccount* takes a list of strings, and creates a list of bank accounts, by processing the elements of the input list. Further, it copies the list of bank accounts and delivers the copy. For creating a list of

bank accounts the program creates an ACons object, a Person object, a BankAccount object and a SavingsAccount object, taking in total $4n + 1$ heap-cells. Moreover, for copying the list, it takes another $4n + 1$ cells. In total, the program requires $8n + 2$ heap-cells.

Bank The program *Bank* is much larger than the other programs (908 LoC). With this program we wish to show that our tool performs well when given a rather large program, and that it does not take a prohibitively large amount of time for the analysis. Of course, there is room for further performance optimisations.

The program takes a list of strings and creates a list of bank accounts based on the data from the input list. Moreover, it creates a Bank object, which contains a list of bank accounts, a list of savings accounts, a balance, a transfer fee, an interest rate and a currency converter. Then it performs various transactions such as transfers and withdrawals from a fixed deposit. Finally, it sorts the list of bank accounts using merge sort. Creating the list of bank accounts takes $6n$ heap-cells, creating the currency converter takes 8 heap-cells and sorting the list takes 1 cell. Further, the program creates a System object; the class System is an internal library that contains useful functions. In total, the program needs $6n + 11$ heap-cells.

Chapter 8

Related Work

In this chapter we give an overview of the research related to this thesis. First, we review works that have similar goals to the work in this thesis, i.e. automatic resource analysis. These often follow very different approaches to the type-based approach that we follow in this thesis.

Second, we consider works that are related to this thesis because they use similar techniques, e.g. precise type systems for capturing program properties, such as refinement types; or type systems that take aliasing into account, such as linear types or types-and-capabilities systems.

This chapter is organised as follows. Section 8.1 reviews research related to resource analysis. Section 8.2 reviews research on refinement types. Finally, Section 8.3 shows some works on linear types and on types-and-capabilities systems.

8.1 Resource analysis

The automatic computation of bounds for the resource usage of programs is a very relevant topic. It has applications specially in embedded and real-time systems, where resource use is constrained. Moreover, ensuring that those systems deliver results on time is very important, since failure can lead to catastrophic consequences.

On the other hand, the problem is undecidable. This means that we will never find a tool that can analyse precisely and efficiently all programs. These are the main reasons why the topic has been researched so intensively in the past years and why so many different approaches to it have been proposed. The approaches are not directly comparable, since some are particularly useful for some sort of programs, while others work better for other kinds of programs.

The problem of analysing the resource usage of imperative and object-oriented programs is even more difficult because analysing data structures that are stored in a mutable heap, which are possibly shared and can even be cyclic, is a hard problem.

Tool	Paradigm	Automatic?	Resource	Aliasing?	GC?
SPEED [GMC09]	oo	no	time	yes	-
LOOPUS [ZGSV11]	imperative	yes	time	no	-
aiT [FHL+01]	imperative	no	WCET	no	-
[PHS10]	oo	no	heap-space	no	no
COSTA [AAG+07]	oo	yes	generic	yes	yes
[CNQR05]	oo	no	heap, stack	no	no
[CNPQ08]	oo	yes	heap, stack	no	no
[HQLC09]	imperative	no	heap, stack	yes	no
[Atk11]	imperative	no	generic	yes	no
[FM11]	oo	no	generic	yes	no
[MP07]	oo	no	time	no	-
[BFGY08]	oo	no	peak-memory	no	yes
RAJA	oo	yes	heap-space	yes	no

Figure 8.1: Overview of tools for resource usage analysis for imperative programs. The column **Paradigm** shows whether the tool is for object-oriented programs or just for imperative programs. The column **Automatic?** shows if the tool is fully-automatic. The column **Resource** shows the resource that the tool analyses. Column **Aliasing?** shows whether the tool takes aliasing into account. Finally, column **GC?** shows whether the tool takes garbage collection into account. This is only applicable if the resource is heap-space or generic.

In this section we give an overview of the different approaches to resource analysis that have been proposed in the literature, with a special emphasis on the tools that analyse imperative and object-oriented programs. We also show how the different approaches tackle the problem of analysing programs with shared mutable data structures. Often they rely on user-defined assertions, in other cases the tools do not handle programs with shared data structures. Fig. 8.1 shows an overview of the approaches for imperative programs.

This section is organised as follows. Section 8.1.1 describes tools based on abstract interpretation. Section 8.1.2 presents tools that follow the classical approach of generating and solving recurrence equations. Section 8.1.3 describes tools based on type systems, as RAJA does. We distinguish between tools that use sized types and tools that use amortised analysis. Section 8.1.4 shows tools based on separation logic, which aim at analysing programs with shared mutable data structures. Finally, Section 8.1.5 presents other approaches.

8.1.1 Abstract interpretation

Abstract interpretation [CC77, CH78] is a powerful technique for static program analysis. The technique was first described by Cousot [CC77]:

> A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract

objects, so that the results of abstract execution give some information on the actual computations.

Various works on resource analysis based on abstract interpretation have been proposed in the literature. Wilhelm and Wachter [WW08] presented a good introduction to abstract interpretation with applications to time analysis.

Gomez and Liu [GL02] described a time analysis for higher-order functional programs. They built time-bound functions and evaluated them based on partially known input structures. Hence, this analysis does not provide a general cost function for all possible inputs, as RAJA does.

Ross [Ros06] developed an automatic time complexity analysis to help C++ library users in selecting types for use in their programs. Each library call is replaced by a cost-bound function, which is declared on the C++ Standard's specification of the libraries. Then, an abstract interpretation is used to approximate the effects of running the program on any possible input. The problem tackled by the author is slightly different than the problem that we study in this thesis, because he does not try to infer resource bounds on concrete implementations, but uses known complexity functions given in specifications.

Gulwani et al. [GMC09] developed SPEED, a system for computing bounds on the number of statements a procedure executes in C++ programs. The system uses counter variables and user-defined quantitative functions that are associated with abstract data-structures. Then, a linear invariant generation tool with support for uninterpreted functions computes linear bounds on the counter variables. Finally, the bounds on the individual counter variables are composed to generate non-linear and disjunctive bounds. In contrast to SPEED, RAJA can compute only linear bounds on procedures that iterate over a data structure. SPEED is particularly suitable for computing bounds on loops that iterate over arithmetic variables. On the other hand, RAJA is fully automatic and does not require any user-input.

More recently, Zuleger et al. [ZGSV11] presented a bound analysis on C programs based on abstract interpretation. More concretely, they used the size-change abstraction (SCA), a predicate abstraction domain that consists of boolean combinations of inequality constraints between integer variables in disjunctive normal form. They did not handle the problem of aliasing, but handled memory references using optimistic assumptions.

AbsInt's **aiT** [FHL$^+$01] is a successful commercial static analyser for the worst-case execution time (WCET) of tasks in real-time systems. It analyses binary executables and takes the cache and pipeline behaviour into account. It performs a value analysis, a cache analysis and a pipeline analysis using abstract interpretation; and a path analysis using integer linear programming.

Puffitsch et al. [PHS10] applied the techniques used for the analysis of WCET in real-time systems to bound the worst-case heap allocations of tasks (WCHAs) of Java programs. Their analysis takes into account the object layout of the underlying JVM and provides different cost functions for the heap allocation based on the different object layouts. The results of their experiments showed that their analysis could find tight bounds for programs but provided rather pessimistic bounds on programs not designed for real-time systems.

The main difference between the research on WCET or WCHAs and ours is the fact that these systems take into account low-level features of hardware, whereas our analysis abstracts away from concrete memory models.

8.1.2 Recurrence solving

This section describes systems for computing resource bounds that use the technique of generating and solving recurrence equations based on the programs. However, these systems generate the equations by symbolic evaluating the programs, often based on abstract interpretation, and so there is some overlap between related work here and that in the previous section.

The techniques based on recurrence solving have the advantage that the bounds they produce are not restricted to a complexity class. On the other hand, solving recurrences is a difficult task.

The first work on automatic time analysis was that of Wegbreit [Weg75]. His prototype system Metric was capable of analysing Lisp programs by symbolically evaluating the functions to determine a set of recurrence equations, and by obtaining closed-form expressions for the execution behaviour of the functions, after solving the equations. He also described the concept of assigning *cost expressions* to program expressions, that describe their cost under a given measure.

Benzinger [Ben01] described the ACAp system, whose main purpose was the automatic complexity analysis of functional programs synthesised with the Nuprl theorem prover [CAA+86]. He followed Wegbreit's approach: a symbolic evaluator generated complexity expressions by abstract interpretation of the program; a recurrence generator translated recursive calls into recurrence equations; a recurrence solver tried to solve the equations. He used the Mathematica symbolic algebra system [Wol03] for recurrence solving.

Unnikrishnan and Stoller [US09] applied Wegbreit's framework to the analysis of the live heap-space usage of programs in a functional language with garbage collection. More concretely, their analysis computes heap usage of programs in the presence of perfect garbage collection. The recurrences generated by the analysis contain the max operator, and they provided methods for solving them. Their analysis is limited to a first-order

functional language with lists as the only data-type. Moreover, they do not allow mutual recursion in the programs. In contrast, RAJA can analyse programs that contain any data type and mutual recursion, but does not handle garbage collection.

The COSTA system by Albert et al. [AAG+07, AAG+12] analyses a subset of Java bytecode using Wegbreit's technique. The system transforms the bytecode in an intermediate rule-based representation and applies a path-length analysis [HPSH06] based on abstract interpretation to infer size relations among program variables. Then, it obtains automatically cost relations from the program based on a given cost model. Finally, it tries to obtain a closed form solution for the cost relations, by using the recurrence solver PUBS [AAGP11]. Since the system is parametric in the cost model, it can analyse various resources such as time or heap-space. The group also used similar techniques to develop a termination analyser [AAC+08]. Moreover, they presented a heap-space analysis which is parametric in the garbage collector [AGGZ10].

Because the bounds generated by the COSTA system are solutions to recurrence equations, they are not restricted to a specific complexity class. In contrast, RAJA can generate only linear bounds. Like RAJA, COSTA does not require user-annotations. On the other hand, the path-length analysis performed by COSTA to infer size-relations is sound with the condition that there is no aliasing and cyclic data, whereas the analysis performed by RAJA is sound for all programs. The possibility of aliasing between variables has been integrated in the type system, which implies that the system can also detect cyclic data, without the need of a shape analysis [SRW02] or an acyclicity analysis [RS06].

We are interested in investigating the relationship between our tree constraints and recurrence equations. Perhaps this will enable us to improve our algorithm for constraint solving, allowing us to compute non-linear bounds.

8.1.3 Type systems

Sized Types

Sized types are a special kind of dependent types to express bounds on the sizes of data structures. They were first described by Hughes et al. [HPS96] for proving properties of reactive systems.

Chin et al. [CNQR05] described a sized type system for characterising the amount of memory required to execute program components. The type of each data structure included parameters that characterised its size properties, using Pressburger arithmetic constraints. The type system is for a Java-like language similar to FJEU, that contains an expression dispose for object deallocation, similar to FJEU's free. They developed a technique for inserting the dispose expression automatically with the help of alias annotations, which we could possibly adapt to the RAJA system. They required

that the objects which are being disposed are non-null. We do not require this explicitly because, according to the operational semantics of FJEU programs, the evaluation of a disposed expression stops if the object to be disposed is null, and our analysis is sound only under the condition that the expression to be analysed evaluates successfully. The authors provided a type-checking algorithm and an implementation, but did not treat the problem of type inference.

More recently, Chin et al. [CNPQ08] inferred linear bounds on the heap and stack usage of low-level assembly-like programs. The analysis relies on a Pressburger arithmetic solver and also performs fixpoint analysis for handling loops and recursion. In contrast, our type inference analysis does not need to perform fixpoint computations because it only handles monomorphic recursion. On the other hand, unlike RAJA, the system from [CNPQ08] inferred path-sensitive information, which improved the precision of the analysis. However, they did not track the values of mutable fields.

In his PhD thesis, Vasconcelos [Vas08] described a sized and effect type system for obtaining upper bounds on the dynamic space usage of functional programs. He inferred sizes of recursive data types and bounds for the stack and heap usage of functions using abstract interpretation techniques.

Amortised Analysis

Systems based on amortised analysis, including RAJA, aim at assigning data structures a non-negative number, called a potential, that they can use to "pay" for using resources. The total potential used by the input data structures in the program then gives an upper bound on the resource usage. Type based methods for amortised analysis use type systems for assigning and sharing the potential.

The first approach to automatically analyse the heap-space usage of programs using type-based amortised analysis was presented by Hofmann and Jost [HJ03]. Jost mentioned in his PhD thesis [Jos10] that this work was the result of constructing a type inference algorithm for Hofmann's LFPL type system [Hof00]. The type inference algorithm in [HJ03] consisted in generating linear arithmetic constraints that arose from the side conditions of typing derivations, which were solved by an LP-Solver. The system was able to compute linear bounds for first-order functional programs automatically.

Campbell [Cam08] studied in his thesis the analysis of stack space of functional programs using the amortised analysis technique. He expressed bounds for stack usage in terms of the depth of data structures.

Shkaravska et al. [SvEvK09] developed a size-aware type system that also used the amortised analysis technique for checking size-dependencies in first-order functional programs. Their analysis was restricted to *shapely* functions, which means that the size of the result is a polynomial in terms of the argument sizes.

Jost et al. [JLS+09] constructed a fully automatic WCET analysis based on the ideas from [HJ03], by building an amortised analysis system for the programming language Hume [HM03].

More recently, Jost et al. [JLHH10] extended the amortised analysis technique to higher-order polymorphic functional programs. Their analysis was generic in the resource to be analysed and they discussed worst-case execution time, stack-space usage and heap-memory consumption. Moreover, they could also handle arbitrary recursive data-types. However, the system was still limited to computing linear bounds.

In his PhD thesis, Hoffmann [Hof11] described an automatic analysis for resource usage of first-order functional programs with lists and binary trees as the only data types, based on the same techniques. In contrast to the previous systems, Hoffmann's system was able to compute polynomial bounds. He annotated types with so called *multivariate resource polynomials*, a generalisation of non-negative linear combinations of binomial coefficients. His experiments showed that the amortised approach is very effective when analysing programs with nested data-types, delivering tight bounds. However, it is not clear how to extend his technique to higher-order functional programs with arbitrary data-types.

Hofmann and Jost [HJ06] also applied the amortised analysis technique to the heap-space analysis of object-oriented programs. Because the type system aimed at describing object-oriented features like imperative update and inheritance, it was more complicated than the equivalent type system for functional programs. As a consequence, they did not provide type inference but provided only the soundness proof. Later, Hofmann and Rodriguez [HR09] presented an automatic type-checking algorithm for the system.

Finally, this thesis provides a type inference algorithm for a modified version of the type system from [HJ06]. The algorithm generates tree and arithmetic constraints and solves the tree constraints using a heuristic algorithm [HR12]. Moreover, in this thesis I also add polymorphism to the refined types which enables a modular analysis. However, I restrict the type inference to handle only programs with monomorphic recursion, to avoid calculating a fixpoint when analysing recursive functions. Another limitation of the algorithm is that it can only compute linear bounds, but I remark that this is due to our back-end constraint solver and not a limitation of the type system itself.

8.1.4 Separation logic

Separation logic [Rey02] is an extension of Hoare logic that facilitates reasoning about imperative programs with mutable heap. It is often used to perform shape analysis in a local way [DOY06]. Various works on resource analysis based on separation logic have been described.

The system HIP/SLEEK [CDNQ10, CDG11] by Chin et al. is aimed

at the automatic verification of properties of heap manipulating programs, based on separation logic. He et al. [HQLC09] proposed a procedure for analysing the heap and stack usage of imperative programs by integrating it in the system HIP/SLEEK. They instrumented the program with explicit operations over variables heap and stk, which represented the memory usage behaviour of the original program. Then, they passed the modified program together with its expected memory specification to HIP/SLEEK. This way they could verify programs that used shared mutable data structures. It is remarkable the similarity between their functions for instrumenting heap space *dec_hp* and *inc_hp* and our rules for constraint generation for the expressions new $((\triangle New))$ and free $((\triangle Free))$, with the main difference that on object creation we not only count the size of the object but also its potential. In contrast to RAJA which is fully automatic, this system relies on user-defined memory usage specifications. Moreover, as shown in the experiments section (Sect. 7.2), our system is also capable of analysing programs that use shared mutable data structures such as doubly-linked lists.

Atkey [Atk11] combined amortised analysis and separation logic to analyse imperative programs, which are similar to Java bytecode but without object-oriented features. He used assertions that describe the current shape of the heap and the resource that the program may consume. The idea is to specify resources that depend on the shape of data structures in the heap. He also employed a proof search procedure to verify that the preconditions of the methods imply their post-conditions, and to infer the resource annotations, using linear programming. Later, Fenacci and MacKenzie [FM11] extended the analysis to Java bytecode. Like RAJA, Atkey's system can compute only linear bounds. On the other hand, the user needs to provide complex annotations, whereas RAJA is fully automatic.

8.1.5 Other techniques

Marion and Péchoux [MP07] described an analysis of object-oriented programs based on sup-interpretations, which provide an upper bound on the outputs sizes of the function denoted by a symbol. However, they did not provide inference of the sup-interpretation functions.

Braberman et al. [BFGY08] computed polynomial bounds on the peak-memory consumption of programs using a region-based memory management. Their analysis generated a set of polynomial maximisation problems which they solved using the Bernstein basis technique. They obtained good results by testing their approach with well-known benchmarks. However, their memory model does not allow aliasing and object deallocations, and their analysis does not deal with memory-consuming recursions. They also mentioned that they could integrate their technique with type-based methods for a better handling of complex data-structures.

8.2 Refinement types

Type systems are very useful for specifying and checking program properties at compile-time. Traditional type systems can check many program properties, but there are many other properties, that could potentially be checked at compile-time, which they cannot check. This has given rise to extensive research in the field of *refinement types*. Those works aim at refining existing type systems to define more precise properties, for which type checking and type inference are decidable, or at least require few program annotations.

The term *refinement type* was first described by Freeman and Pfenning [FP91]. They described a refinement of ML's type system. The system allowed the specification of types such as singleton lists, that refined the notion of lists. They also provided type inference of refinement types, by performing abstract interpretation over a finite-lattice of refinements of ML types.

Xi and Pfenning [XP99] presented DML(C), an extension of ML with a restricted form of dependent types, which was parametrised over the domain of constraints C, from which the type index objects were taken. Type inference for the extended system was not possible, but they reduced type-checking of annotated programs to the problem of constraint satisfaction in the constraint domain C. They also provided an implementation for the domain of linear inequalities on integers, and presented applications such as the elimination of array bound checks.

Later, Xi [Xi00] described the dependently typed imperative programming language Xanadu. To handle mutable index expressions, he proposed allowing to change the type of a variable during evaluation.

The systems of refinement types provided so far were rather restricted to ensure that specifications could be checked statically. In the following, various authors tried to increase the expressivity of specifications by mixing type refinement systems with other techniques.

Mandelbaum et al. [MWH03] developed a two level system for reasoning about effectful programs. The first level was a standard ML-style type system, and the second level used a logic of type refinements to check more precise properties. Their aim was "to provide a general-purpose logical framework for reasoning about effectful computations."

Flanagan [Fla06] developed hybrid type checking, which was a synthesis of static type checking and dynamically-checked contracts, as an attempt to overcome the limitations of purely-static and purely-dynamic approaches. He illustrated this idea on an expressive (an statically undecidable) dependently-typed system, which he called λ^H. When the type-checker could neither accept nor refute a subtyping judgement, then it accepted the corresponding programs but added dynamic checks to ensure that no violations occurred at runtime.

Later, Knowles and Flanagan [KF07] developed a type reconstruction

algorithm for λ^H. Although λ^H is an extension of the lambda calculus with dependent function types and refined based types, we found the general idea of this algorithm surprisingly useful for our type inference algorithm for RAJA. We followed the three phases of type reconstruction they proposed: 1. Processing the input program to obtain a set of subtyping constraints, by constructing constraint generation rules which are sound and complete with respect with the typing rules. 2. Reducing the subtyping constraints into a set of implication constraints for their system, and a set of linear arithmetic constraints for our system. 3. Solving the constraints which gives a valuation π (or type replacement), that implies typeability of the program.

Rondon et al. [RKJ08] presented *Liquid Types*, a refinement type system for which they developed type inference based on a combination of Hindley-Milner type inference and predicate abstraction. They implemented the inference on DSOLVE, which analysed OCaml programs and obtained very good results in verifying programs with very few annotations.

Nystrom et al. [NSPG08] described a system of constraint-based types, a form of dependent types, for an object-oriented language. They associated constraints with class definitions and with method and constructor definitions. Moreover, the system was parametric in the constraint system, such that it supported different constraint systems using compiler pluggings. They developed a type checking algorithm that passed the constraints to the appropriate constraint solver which checked the constraints for satisfiability. To check subtyping, the constraint solver also needed to check constraint entailment. The main similarity between this system and RAJA is that both are constraint-based type systems for an object-oriented program. However, RAJA does not allow constraints in classes but only in methods, and the constraints are over types, not over values. Moreover, we present a type inference algorithm for RAJA types, whereas they did not treat the problem of type inference.

Jhala et al. [JMR10] reduced the problem of refinement type inference to computing invariants of first-order imperative programs without recursive data-types. They aimed at using existing abstract interpretation tools for imperative programs to infer refinement types.

8.3 Linear types and capabilities

Wadler [Wad90] described a linear type system, where values belonging to a linear type must be used exactly once. To make the type system more flexible, he also introduced the `let!` constructor to allow read-only access. Our type system is similar to linear type systems, because duplication of references happens in a controlled way, via the sharing rule.

Crary et al. [CWM99] described the Capability Calculus, a compiler intermediate language for supporting region-based memory management.

The calculus handles aliasing by tracking non-aliasing. They tracked regions with one of two multiplicities: $\{r^+\}$ is the capability to access region r without restrictions, and $\{r^1\}$ adds the restriction that r is unique. Our refined types have some similarity with unduplicatable capabilities, except for types with potential 0, which are duplicatable.

Degen et al. presented Java(X) [DTW07], an extension of Java with a refinement type system, parametrised by a poset X, from which the annotations are drawn. They also introduced the concept of activity annotations which is a capability for updating a field in an object. Moreover, they described a splitting relation that splits the capability for a resource between different paths to it. This relation is similar to our sharing relation. In contrast to RAJA, they did not handle inheritance and did not provide type inference. Another difference to RAJA is that Java(X) supports typestate change, while RAJA does not.

More recently, Pilkiewicz and Pottier [PP11] extended a type-and-capability system with the notion of monotonic state, meaning that "only the owner of an object is allowed to change its type, and, furthermore, only in a monotonic manner". They also applied the system for implementing types that represent time complexity properties that can be assigned to thunks, using amortised analysis. Interestingly, they implemented credits as capabilities ($n\$$ is a linear capability that represents n credits). They also posed the subtyping axiom $(n+p)\$ \equiv n\$ + p\$$, which underlines the same general idea as our sharing relation.

Chapter 9

Conclusions

In this thesis we have presented a type-based analysis of the heap-space requirements of object-oriented programs. The soundness of each step of the analysis has been rigorously proved. Moreover, the analysis was modular, enabled by the use of polymorphic types and a procedure for locally eliminating variables from constraints. Thus, in principle, the analysis is capable of scaling to large programs. We can see this in practice via our experiments, which have shown that we can analyse programs of 900 LoC in around 10 minutes. Nevertheless, there is plenty of room for improvement.

Polymorphic types also enable an incremental analysis. The types can be saved after the analysis, and in most cases they do not need to be re-analysed when more classes and methods are added to the programs. One could imagine a scenario where libraries are delivered together with RAJA types, and the application can use the RAJA types of the libraries to analyse its resource consumption, without the need to re-analyse the libraries, and without needing the source code of the libraries.

We also showed that the type inference delivers a finite RAJA program; that is, a RAJA program, where each method has a finite set of monomorphic RAJA types, and where all the views are regular. For those programs, we also developed a simple and efficient type checking algorithm. The RAJA types can be regarded as a certificate for resource consumption, that can be checked efficiently with the type checking algorithm.

9.1 Further directions

We can think of various directions for future research. There are two main branches of research that may increase the number of programs that RAJA can analyse and improve the precision of the computed bounds. First, we can seek to improve the type system; second, we can seek to improve the algorithm for solving constraints over the infinite trees.

Improving the type system Although we could not yet grasp the full expressive power of the RAJA type system because of the difficulties with constraint solving, we are convinced that RAJA is a very expressive type system, capable of computing non-linear bounds. However, the type system can be improved. Perhaps adding typestates will increase the expressivity and precision of the system, because changing the types of objects after imperative update can make their types more precise. Another way of improving the precision of the bounds is to make the analysis path sensitive in the style of Chin et al. [CNPQ08].

The RAJA type system concentrates on the cases when resource consumption depends on the size and shape of data-structures. However, it does not handle the cases where resource consumption depends on the values of arithmetic variables, that are incremented in a loop. There are very good tools that can analyse such programs very precisely, such as SPEED [GMC09]. Perhaps combining such ideas with our analysis would lead to a more useful tool.

In the current RAJA system, we use potential from the variable this only, because this is the only variable guaranteed to be non null at compile time. However, we could think of an extension of the system that allowed to use the potential of any variable combined with a static non-nullity analysis. This may lead to a more expressive type system.

In principle, it is clear how to extend the RAJA system to the analysis of other resources other than heap-space. However, this implies changing the typing rules and proving their soundness again. It would be desirable to have a type system where the concrete resource is not fixed in the typing rules, in the style of Jost et al. [JLH+09]. This would make the analysis truly parametric in the resource.

Solving constraints over infinite trees In this thesis we encountered the problem of arithmetic constraints over infinite trees, and we developed a heuristic algorithm for its solution, that often succeeds when the constraints admit regular solutions. However, the research in this area is still preliminary and there is plenty of room for future research. For instance, it would be useful to settle the question of decidability of this problem in general, and also to identify larger tracktable subproblems relevant for resource analysis.

We have already started to analyse ways of reducing constraints over trees to recurrences, to be able to use recurrence solvers to solve them. For instance, the constraints $l(x) = x + x, \Diamond(x) = 1$ can be reduced to the recurrence equation $f(n+1) = f(n) + f(n)$ where $f(n_0) = 1$, which has the solution $f(n) = 2^n$. Then, the solution to the tree constraint would be a tree t_0 belonging to the family of trees $t_{i|i\in\mathbb{N}}$ where each t_i is defined by: $\Diamond(t_i) = 2^i$ and $l(t_i) = t_{i+1}$.

Finally, studying the optimisation problem would be also useful for computing optimised bounds of resource usage.

More Java features This thesis focused on making possible the automatic heap-space analysis of FJEU programs and did not analyse the problem of extending FJEU with more Java features. However, to enable the resource analysis of Java programs with our approach, it is necessary to add more Java features to our target language. Perhaps the simplest way of doing this is by providing a resource-preserving translation of Java programs to FJEU programs. For instance, one could translate the loops to recursive functions, and exceptions to conditional expressions.

Also, the `free`() expressions should be introduced automatically, by following an approach that models the behaviour of the garbage collector. There are various works on the static prediction of garbage collection [US09, AGGZ10] that could be adapted to our system.

These transformations and extensions are well studied. Thus, we can conclude that our analysis has the potential to become an efficient, sound and fully automatic tool for the resource analysis of Java programs.

Bibliography

[AAC+08] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination analysis of java bytecode. In Gilles Barthe and Frank S. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2008.

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[AGGZ10] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In Jan Vitek and Doug Lea, editors, *ISMM*, pages 121–130. ACM, 2010.

[Atk11] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.

[BBG+00] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, 2000. Available from http://www.rtsj.org.

[Ben01] Ralph Benzinger. Automated complexity analysis of nuprl extracted programs journal of functional programming. *J. Funct. Program.*, 11(1):3–31, 2001.

[BFGY08] Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In Jones and Blackburn [JB08], pages 141–150.

[BG00] Achim Blumensath and Erich Grädel. Automatic structures. In *LICS*, pages 51–62, 2000.

[BGH10] Lennart Beringer, Robert Grabowski, and Martin Hofmann. Verifying pointer and string analyses with region type systems. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2010.

[BIL03] Marius Bozga, Radu Iosif, and Yassine Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 55–65, New York, NY, USA, 2003. ACM Press.

[CAA+86] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing mathematics with the nuprl proof development system, 1986.

[Cam08] Brian Campbell. *Type-based amortized stack memory prediction.* PhD thesis, University of Edinburgh, 2008.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CDG11] Wei-Ngan Chin, Cristina David, and Cristian Gherghina. A hip and sleek verification system. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA Companion*, pages 9–10. ACM, 2011.

[CDNQ10] Wei-Ngan Chin, Cristina David, Huu H. Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, August 2010.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *POPL*, pages 84–96. ACM Press, 1978.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* MIT Press, September 2001.

[CNPQ08] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In Jones and Blackburn [JB08], pages 151–160.

[CNQR05] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for oo programs. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[CWM99] Karl Crary, David Walker, and J. Gregory Morrisett. Typed memory management in a calculus of capabilities. In Andrew W. Appel and Alex Aiken, editors, *POPL*, pages 262–275. ACM, 1999.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order: Second Edition.* Cambridge University Press, 2002.

[DTW07] Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with java(X). In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 550–574. Springer, 2007.

[DV07] Stefan Dantchev and Frank D. Valencia. On infinite csp's, 2007.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 171–183, New York, January 1998. Association for Computing Machinery.

[Fla06] Cormac Flanagan. Hybrid type checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 245–256. ACM, 2006.

[FM11] Damon Fenacci and Kenneth MacKenzie. Static resource analysis for java bytecode using amortisation and separation logic. *Electr. Notes Theor. Comput. Sci.*, 279(1):19–32, 2011.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[GL02] Gustavo Gomez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language. In Kenichi Asai and Wei-Ngan Chin, editors, *PEPM*, pages 75–86. ACM, 2002.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 127–139. ACM, 2009.

[Gra11] *Type-Based Enforcement of Secure Programming Guidelines Code Injection Prevention at SAP*, 2011.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 357–370. ACM, 2011.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 185–197. ACM, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[HJR] Martin Hofmann, Steffen Jost, and Dulma Rodriguez. Type-based amortised heap space analysis. (complete soundness proof). In http://raja.tcs.ifi.lmu.de/download/files/rajaSoundProof.pdf.

[HM03] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In *In Proc. Conf. Generative Programming and Component Engineering, Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2003.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179. Springer, 2000.

[Hof11] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universiät München, 2011.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL*, pages 410–423. ACM Press, 1996.

[HPSH06] Patricia M. Hill, Etienne Payet, Fausto Spoto, and Hill. Path-length analysis of object-oriented programs. In *In Proc. EAAI*, 2006.

[HQLC09] Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. Memory usage verification using hip/sleek. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2009.

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[HR12] Martin Hofmann and Dulma Rodriguez. Linear constraints over infinite trees. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2012.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–26, New York, NY, USA, 2001. ACM Press.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[JB08] Richard Jones and Stephen M. Blackburn, editors. *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*. ACM, 2008.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369, Heidelberg, 2009. Springer.

[JLHH10] Steffen Jost, Hans-Wolfgang Loid, Kevin Hammond, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–236, New York, NY, USA, January 2010. ACM.

[JLS+09] S. Jost, H-W. Loidl, N. Scaife, K. Hammond, G. Michaelson, and M. Hofmann. Worst-Case Execution Time Analysis through Types. In *21st Euromicro Conf. on Real-Time Systems (ECRTS'09)*, pages 13–16. ACM, July 2009. Work-in-Progress Session.

[JMR10] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. Refinement type inference via abstract interpretation. *CoRR*, abs/1004.2884, 2010.

[Jon81] H.B.M. Jonkers. Abstract storage structures. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[Jos10] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Faculty of Mathematics, Computer Science and Statistics, LMU Munich, Germany, September 2010.

[Ker09] Rody Kersten. A strict-size logic for featherweight java extended with update. Master's thesis, Radboud University Nijmegen, 2009.

[KF07] Kenneth L. Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 505–519. Springer, 2007.

[KN04] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[lP05] Thomas Mashølhave and Lars H. Petersen. Assignment featherweight java: Bringing mutable state to featherweight java. Master's thesis, University of Aarhus, 2005.

[Mot36] TS Motzkin. *Beiträege zur Theorie der linearen Ungleichungen*. PhD thesis, University of Basel, 1936.

[MP07] Jean-Yves Marion and Romain Péchoux. Resource control of object-oriented programs. *CoRR*, abs/0706.2293, 2007.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 213–225. ACM, 2003.

[NSPG08] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. *ACM SIGPLAN Notices*, 43(10):457–474, September 2008.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[PHS10] Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 464–478. Springer, 2010.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PP11] Alexandre Pilkiewicz and Francois Pottier. The essence of monotonic state. In Stephanie Weirich and Derek Dreyer, editors, *TLDI*, pages 73–86. ACM, 2011.

[Rai92] Gary Raines. Real time ada in the international space station freedom. In *Proceedings of the 11th Ada-Europe International Conference on Ada: Moving Towards 2000*, 1992.

[RBR+05] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 296–309, New York, NY, USA, 2005. ACM Press.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *ACM SIGPLAN Notices*, 43(6):159–169, June 2008.

[Ros06] Kyle D. Ross. Towards an automatic complexity analysis for generic programs. In Ralf Hinze, editor, *ICFP-WGP*, pages 87–95. ACM, 2006.

[RS06] Stefano Rossignoli and Fausto Spoto. Detecting non-cyclicity by abstract compilation into boolean functions. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2006.

[SR10] Alexandra Silva and Jan J. M. M. Rutten. A coinductive calculus of binary trees. *Inf. Comput*, 208(5):578–593, 2010.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SvEvK09] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2), 2009.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[Tur99] Jim Turley. Embedded processors by the numbers. *Embedded Systems Programming*, 1999.

[US09] Leena Unnikrishnan and Scott D. Stoller. Parametric heap usage analysis for functional programs. In Hillel Kolodner and Guy L. Steele Jr., editors, *ISMM*, pages 139–148. ACM, 2009.

[Vas08] P. Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008.

[Wad90] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.

[Wol03] Stephen Wolfram. *The Mathematica book (5. ed.)*. Wolfram-Media, 2003.

[WW08] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008.

[Xi00] Xi. Imperative programming with dependent types. In *LICS: IEEE Symposium on Logic in Computer Science*, 2000.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 20–22, 1999.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2011.