Increasing Software Quality of JavaScript Programs

An Approach Based on Type Systems

Phillip Heidegger

Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg im Breisgau

Dekan:	Prof. Dr. Bernd Becker
Erstgutachter:	Prof. Dr. Peter Thiemann, Universität Freiburg
Zweitgutachter:	Prof. Dr. Anders Møller, Aarhus University
Tag der Disputation:	28.06.2012

Abstract

Software development in JavaScript (JS), one of the most important programming languages today, is a particular challenge because it is hard to obtain accurate descriptions of the statics and dynamics of a program. Static and dynamic program analysis can provide such descriptions, but some features of JS make it hard to analyze programs, especially the with-statement and the dynamic execution of source code with eval.

This dissertation presents two approaches to increasing the efficiency of software development in JS. The first part presents a static type system that uses flow-sensitivity to precisely analyze the initialization phase of objects and that treats objects later on in their life cycle flow-insensitively to keep the type system simple. The second part presents the tool JSConTest that is inspired by the Design by Contract approach. JSConTest is based on monitoring and (guided) random testing.

The static type system of the first part is capable of typing JS programs with cyclic object initialization without falling back to use null pointers. It categorizes objects based on constructors or allocation sites. Because of its flow-sensitivity in the initialization phase of objects, the type system can reflect changes to the shape of the objects precisely. The type system is proven sound by a progress and preservation lemma. The proofs heavily depend on an invariant stating that in the initialization phase, the type system can establish a one-to-one relation between the object to initialize and its suitable abstraction. A type inference algorithm based on constraint generation and constraint solving reduces the annotation burden significantly.

The second part of this work presents the dynamic tool JSConTest. Following the Design by Contract methodology, functions can be annotated by contracts. JSConTest supports two different kinds of contracts, type contracts (TC) and access permission contracts (APC). TC are similar to type signatures known from statically typed languages (e.g. Java, ML, Haskell), but they are as expressive as dependent types. APC are a new approach to specifying what side effects a function is allowed to perform. A formalization of APC together with a discussion of its design principles lay the foundation of an implementation of APC. An inference algorithm for APC has been developed, too. The two parts of JSConTest, TC and APC, are evaluated with cases studies that show how they increase the quality of JS programs.

Zusammenfassung

Softwareentwicklung in JavaScript (JS), eine der wichtigsten Programmiersprachen unserer Zeit, ist eine besondere Herausforderung. Ein Grund hierfür ist, dass es zahlreiche Merkmale von JS gibt, die das statische Analysieren von JS verhindern, oder zumindest erschweren. Vor allem das with Statement und das dynamische Ausführen von Quelltext mit eval bereiten Probleme.

Diese Dissertation stellt zwei Ansätze vor, die Produktivität der Softwareentwicklung in JS zu steigern. Der erste Ansatz besteht aus einem statischem Typsystem, das wegen seiner Flusssensitivität Objekte in ihrer Initialisierungsphase präzise analysieren kann. Durch die flussinsensitive Behandlung der Objekte nach ihrer Initialisierung ist es möglich den Einfluss der Flusssensitivität zu beschränken und damit das Typsystem einfach zu halten. Als zweiter Ansatz wird ein Tool mit dem Namen JSConTest vorgestellt, welches durch das Konzept „Design by Contract" inspiriert ist. Verträge („contracts") in JSConTest werden durch Monitoring und randomisiertes Testen geprüft.

Das statische Typsystem des ersten Teils ist in der Lage, JS Programme, die zyklische Objektstrukturen initialisieren, zu typen, ohne dabei Null-Referenzen einsetzen zu müssen. Das Typsystem kategorisiert Objekte, basierend auf dem Konstruktor mit dem sie erzeugt wurden, oder anhand der Zeile in der die Objekte erzeugt wurden. Wegen der Flusssensitivität während der Initialisierungsphase des Objekts kann das Typsystem Änderungen in der Struktur des Objekts nachhalten. Die Korrektheit („soundness") des Typsystems wird mithilfe eines „Progress" und „Preservation" Lemmas bewiesen. Die Beweise der Lemmata basieren auf einer Invariante, welche aussagt, dass das Typsystem eine eins zu eins Relation zwischen Typen und Objekten während deren Initialisierungsphase sicherstellen kann. Ein Typinferenzalgorithmus, welcher auf dem Generieren und Vereinfachen von Constraints besteht, reduziert den Annotationsaufwand beträchtlich.

Der zweite Teil der Dissertation stellt das Tool JSConTest vor, welches zwei unterschiedliche Arten von Verträgen, Typverträge (TV) und „Access Permission" Verträge (APV), unterstützt. TV sind Typsignaturen aus Sprachen wie Java, ML oder Haskell ähnlich. Jedoch ist die Ausdrucksstärke von TV nicht eingeschränkt, denn sie werden durch Monitoring überprüft. APV stellen eine neue Methode, Seiteneffekte von Funktionen zu beschreiben, dar. Eine Formalisierung der APV zusammen mit eine ausführlichen Erörterung der Designprinzipien bieten eine solide Grundlage für eine Implementierung. Sowohl TV wie auch APV werden anhand von Fallstudien auf ihre Effektivität beim Softwareentwurf überprüft.

Acknowledgements

Peter Thiemann offered me the chance to spend a lot of time with interesting practical and theoretical questions in the area of programming languages. I got a lot of freedom to try out new ideas. Peter guides me in a way that I could focus myself on important aspects of research without restricting my freedom. I'm very grateful that he helped me developing my presentation skills, both in vocal and written form. Thank you very much, Peter for your guidance and support!

I would like to thank Anders Møller for being my second advisor. Thank you for reviewing my thesis and traveling to Germany for my disputation.

Matthias Neubauer, Stefan Wehr, Markus Degen, Annette Bieniusa, Konrad Anton, Manuel Geffken, Matthias Keil and Robert Jakob, my colleagues during my time at the University of Freiburg, discussed topics from my thesis with me. Dirk Kienle, Stefan Frank, Annette Bieniusa and Peter Thiemann contributed code to JSConTest. Stefan W., Stefan Jansen, Manuel, Matthias K. and Robert were willing to proofread my thesis. Thank you all for the great time together and your help.

For Conny

Contents

List of Figures

List of Tables

List of Programs

1 Introduction

Harvard Business School professor John Quelch states in his article “Quantifying the Economic Impact of the Internet” [103] that each Internet job supports approximately 1.5 additional jobs elsewhere in the economy. He estimates the economic value of the Internet to be $1.1 trillion. Clearly, the Internet is important in our economy and society.

Recent numbers from W3Techs [116] tell that 90% of the top 1 million websites use JavaScript as client-side language in their web site. Today’s webpages contain not just simple, small JavaScript fragments to make the webpage a little bit more convenient. Today, whole applications with complicated requirements are written in JavaScript. According to the TIOBE Programming Community index [22], which measures the popularity of programming languages, JavaScript is among the most important programming languages in recent years. Thus, finding methods to make software development more productive in JavaScript is an important task.

One key factor that determines the efficiency of software development is the choice of the programming language – or from the viewpoint of a researcher in the field of programming languages – the choice of a set of appropriate language constructs that make software development efficient. Measuring the impact of choosing a programming language is a difficult task [13], but it is apparent that the influence of the programming language is rather high.

For JavaScript, some particular problems come to mind. For example all JavaScript fragments included in a webpage share the same global environment, a problem addressed by many recent publications [27, 67, 79, 80]. The difficulties have emerged because JavaScript grew out of its originally intended use, which was to write small scripts to make web pages interactive and comfortable. Other problems are caused by the automatic value conversion [114], the execution of code using eval or a poorly designed language construct that breaks static scoping called with-statement. To make the situation worse, JavaScript lacks many features that are usually considered as very important for large-scale software development. It does not provide a package management system. It lacks facilities for data encapsulation. It supports only a dynamic type system. Therefore, a lot of common programming erros, for example calling a method that is not supported by an object, cannot be inhibited.

Programming languages today offer software developers two different kinds of mechanisms to increase productivity of the software development process, static approaches and dynamic approaches.

Static mechanisms [47–49] ensure that a program fulfills a property by analyzing the program source code. Unlike dynamic approaches, a static analysis does not execute the program to achieve its results. Static analyses differ a lot in the goal they try to obtain. Some systems only focus on generating good suggestions or warnings for the programmer. Consequentially, the programmer gets false positives and false negatives from these kind of systems. These systems do not give a guarantee that the warning is actually based on a programming error, and they also do not guaranty that a programming error does not exist if no warning is presented. An example for such a system is JSLint [24], a program which reports warnings about bad style, bad practice or typical pitfalls in JavaScript source code. JSLint is based on a simliar system called Lint developed at Bell Labs by Stephen C. Johnson for the programming language C during the 1970s.

Nowadays, static analyses typically provide a soundness result; that is, if the analysis does not present a violation, a specific kind of error cannot happen during all possible program executions. Consequentially, a static analysis gives a guaranty about the absence of specific kinds of errors. As an example, in the type system of Java it is not possible to multiply strings with integers. Such systems are often integrated with the compiler realized as a type checker.

Verification [12, 58, 76, 97] is another kind of static analysis. Unlike type systems, verification is a semiautomatic process. That means, a person has to help the proof assistent to complete its task. Hence, verification is clearly much more powerful with respect to the properties it can establish. As an example, a typical property that is too complicated to prove for most static type systems is to prove the absence of an index out of bounds errors for arrays in a given program.

Dynamic approaches [21, 31, 78, 84, 108] take a different path. Instead of proving that a program fulfills its specification, dynamic approaches execute the program. Therefore, they cannot guarantee a termination of their analysis, because the program may not terminate. Typically, dynamic methods can only provide evidence about the presence of an error. Only in rare cases[1] these methods are able to prove the absence of erros. Their strength is that each time they find an error in the program, they come up with an example that spots the error. Correcting a program having such a counter example at hand is much simpler than doing the correction based on a the result of a static analysis. In the latter case the programmer needs a lot of knowledge about how the static analysis works to correct the bug.

The first part of this dissertation presents a static type system for JavaScript. It guarantees the absence of null pointer exceptions for JavaScript while supporting the initialization of cyclic object structures. In JavaScript this task is much more difficult than in classical languages as C, C++ or Java because the set of fields (or properties) for an object is not known statically. In Java or C++ for example, the set of fields of an object is determined by the class. Inheritance, as present in these languages, also does not break this guarantee, because it only allows to extend the set of available fields. In these languages it is safe to conclude from the static type of a variable that the corresponding object always has the appropriate

[1] If the input domain is not dividable into a finite class of equivalent inputs.

field during program execution.

Since there is no static class information available, significant challenges arise when one is confronted with the task to design a static type system for JavaScript. One critical question is how to abstract from actual objects. The type system of the first part follows the approach of Jones and Muchnick [68] to use allocation points to abstract over objects. But instead of using one abstraction per allocation point, the type system uses two abstractions. One abstraction is used to represent all old objects, and one is used to represent the most-recent object. This idea is based on the work of Balakrishnan and Reps [9], who call the idea recency abstraction. Because a one-to-one relation between the type and value level is established for most-recent objects, the type system can safely allow strong updates; that is, the type system safely allows a type change for properties of most-recent objects.

For a dynamic approach the first question typically is, what is the property that should hold for a program. The next step is to decide, depending on the property, what kind of inputs are used to execute the program. Today, especially in dynamically typed languages like JavaScript, it is common to create a large test suite with a lot of test cases to ensure correctness of programs.

In addition to a set of test cases to assure a property, one can formulate the property in a more general form and let a monitoring approach observe the property during program execution. For example, a function sum that computes the sum of a list of natural numbers[2] fulfills the following property: $\mathsf{sum}(\mathsf{l}) < \mathsf{sum}(\mathsf{x} :: \mathsf{l})$ if $\mathsf{x} :: \mathsf{l}$ is the list l extended with the element x.

The second part of this dissertation presents JSConTest, a tool for partial specification of JavaScript operations. To use JSConTest, the programmer annotates his JavaScript functions with type signatures. JSConTest uses a dynamic monitoring approach to detect if a type signature of an annotated function is valid. Additionally, JSConTest uses the type signatures to randomly generate input values for operations. This dual role of type signatures makes the approach efficient because the software developer has a simple method at hand to create a random test suite and at the same time to observe the types of his operations. Because observing the correctness of type signatures not only makes sense during random testing, JSConTest offers a monitoring facility capable of observing the types of values during execution. It can be used for regression testing, unit testing, system testing and during actual execution in the browser of the webpage visitor.

But type signatures specify only the functional behavior of an operation. Because JavaScript is an imperative language, it is also important to specify the side effects an operation performs. The observation that all read and write accesses in JavaScript can be expressed using a base object, a property path (a list of properties) and a classifier to specify if the access is a read or write operation let us define an access permission contract (APC) as set of classified access path. A developer can enrich the type signatures of operations by an APC to specify what read accesses and what write accesses are allowed to be performed by the function.

[2] $\mathbb{N}_+$, see Section 1.3

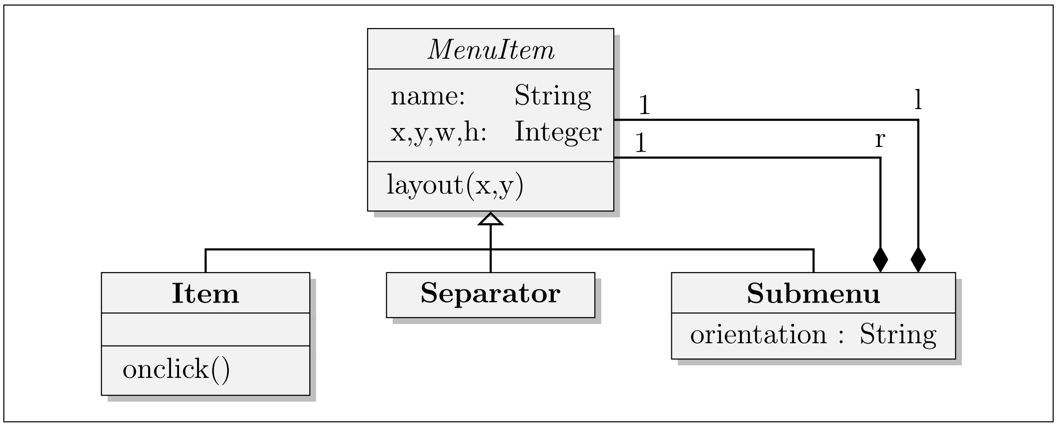


Figure 1.1: Class diagram of a menu class hierarchy.

Program 1.1 JavaScript – Menu.

```
/*c (int, int) → {w:int, h:int}         ← type signature
   with [this./l|r/*./x|y|h|w/]         ← access permission contract
*/
function layout(x, y) { /** source code of layout function here */ }
```

As an example, consider the class hierarchy from Figure 1.1. A `MenuItem` is either an `Item`, a `Separator` or a `Submenu`. The layout function, which has the purpose to compute the positions of the different elements on the screen, carries the type signature and APC as shown in Program 1.1. The type signature expresses that the function takes two integer numbers and returns an object that contains the two properties h and w. The syntax for APCs is similar to JavaScript Regular Expressions (regular expression literals are enclosed in slashes in JavaScript). In the example, this specifies the base object, /l|r/* grants the right to traverse the heap by following the properties l and r arbitrary often, and /x|y|h|w/ states that the function is allowed to modify the properties $\{x, y, h, w\}$. Semantically, the APC of the layout function states that the function is allowed to modify the positions of all tree elements, but not to modify the structure of the tree.

Instead of going on to explain how type signatures and access permission contracts work in detail, we proceed by stating the contributions of this dissertation.

1.1 Contributions

The first part of the dissertation presents a static type system, which is based on joint work with Peter Thiemann [56, 57], which makes the following contributions:

- It presents $\mathcal{JSC}$, a fully formalized core language for JavaScript with its small step operational semantics.

- It formalizes $\mathcal{JSR}$, a core calculus similar to $\mathcal{JSC}$ that uses the idea of recency abstraction in its static type system. It proves type soundness, determinacy of evaluation and the decidability of type checking and type inference for $\mathcal{JSR}$.
- It discusses the extension of $\mathcal{JSR}$ with features that are important for the full JavaScript language and presents formalizations for some extensions.
- Since type inference for $\mathcal{JSR}$ is expensive, the dissertation presents a practical solution for inferring the places of demote expressions and abstract locations in form of simple preprocessing steps.

  It sketches a constraint generation and constraint solving algorithm, which is implemented as a prototype and available on the web [51]. The implementation contains approximately 9000 lines of OCaml code.

The second part of the dissertation presents a dynamic contract system for JavaScript. It is based on joint work with Peter Thiemann [53–55] and Annette Bieniusa [53]. It makes the following contributions:

- It introduces a contract system based on simple type contracts to dynamically ensure type safety for JavaScript.
- It presents access permission contracts (APC), which are used to specify the side effects of functions. It explores the design space for APC and formalizes a calculus of APC.
- It presents case studies to show the efficiency of contracts and APC using the method of mutation testing.
- It illustrates an inference algorithm for access contracts based on dynamic monitoring.
- A prototype implementation of contracts, access contracts and the source code of case studies is available on the web [52]. The implementation consists of about 10,000 lines of OCaml code and approximately 15,000 lines of JavaScript code.

1.2 Outline

After the introduction Chapter 2 introduces the aspects of JS relevant to this dissertation.

The first main part of this work presents a static type system for JavaScript. It starts with an introduction to the idea of recency abstraction (Chapter 3), which was invented in the context of abstract interpretation [9]. It continues to present the formal system of the core calculus for $\mathcal{JSC}$ and $\mathcal{JSR}$ in Chapter 4. This chapter also contains the soundness proof of $\mathcal{JSR}$. The type inference for

$\mathcal{JSR}$ is presented in Chapter 5 together with the poof of decidability of type inference. Chapter 6 presents extensions to $\mathcal{JSR}$ and discusses how to include the most important language features of JavaScript that are not represented in $\mathcal{JSR}$. Chapter 7 compares the work with the literature.

The second part of the work starts by giving a short introduction into the contract and access contract system from a user perspective in Chapter 8. Next, Chapter 9 discusses the design principles on which the access contract system are based. In Chapter 10 the access inference algorithm is presented and the soundness of the algorithm is proved. Chapter 11 presents the implementation of JSConTest. The case studies done to provide evidence that the contract and access contract system is useful are presented in Chapter 12. Chapter 13 presents related work.

The dissertation finishes with a conclusion in Chapter 14.

1.3 Mathematical Notation

Logical Foundation The symbol $\wedge$ stands for *logical and*, the symbol $\vee$ for *logical or*, the symbol $\rightarrow$ for the *logical implication* and $\neg$ is *logical negation*. The universal quantifier $\forall$ and the existential quantifier $\exists$ denote *all* and *exists* as usual in first order logic. As usual in *ZFC*, for a set A, $a \in A$ denotes that *a is an element of* A, and $a = b$ denotes that a is *equal* to b.

For a relational symbol R and a first order formular F let $\forall x \in R : F$ be a shortcut for $\forall x(Rx \rightarrow F)$. We often omit existential quantifiers, because the formulas are easier to read without having them written down. For an example consider the next paragraph.

Sets For a set A let 2^A denote the power set of A; that is the set of all subsets of A. For two sets, A and B, $A \cup B$ denotes the union ($A \cup B := \{a \mid a \in A \vee a \in B\}$) of the sets A and B, $A \cap B := \{a \mid a \in A \wedge a \in B\}$ the intersection of A and B and $A - B$ the difference, $\{a \mid a \in A \wedge a \notin B\}$. The subset relation for two sets A and B, $A \subseteq B$, holds, if all elements from A are elements of B, so to say: $A \subseteq B$ iff $\forall a(a \in A \rightarrow a \in B)$.

For sets A_i let $A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i\}$ denote the Cartesian product of the sets A_i. The elements of $A_1 \times \cdots \times A_n$ are called n-tuples. For a set A, which contains n-tuples, $A_{\downarrow i}$ projects out the ith component for all elements of A: $A_{\downarrow i} := \{x_i \mid \exists x_1 \ldots \exists x_{i-1} \exists x_{i+1} \exists x_n (x_1, \ldots, x_n) \in A\}$. Often, we omit the existential quantifiers and write $A_{\downarrow i} := \{x_i \mid (x_1, \ldots, x_n) \in A\}$.

For a set A and a formula f, $\exists^{\leq 1} x \in A : f$ means there exists at most one element in A, named x, such that f is fulfilled. This is an usual extension to first order logic. It can be defined easily as a shortcut:

$$\exists^{\leq 1} x \in A : f ::= \neg \exists x \in A : f \vee \exists x \in A : (f \wedge \forall x' \in A : f[x \mapsto x'] \implies x' = x)$$

Popular Sets Let $\mathbb{N}$ be the set of non-negative integers ($\mathbb{N} := \{0, 1, 2, \ldots\}$ and let $\mathbb{N}_+$ be the set of positive integers/natural numbers ($\mathbb{N}_+ = \{1, 2, 3, \ldots\}$).

We denote with Prop the set of property names, which is an unspecified, infinite countable set of strings. We use the metavariables a, b and p, q to range over properties. The set $\mathsf{Variable}$ is an infinite countable set of variable names and we use typically x and f as metavariables to range over variables. The set Value is an infinite countable set of values that always contains the special value `udf`. Typically, we use v as a metavariable to range over values. Since the precise definition of Value differs in the two parts of the dissertation, it is defined precisely in the corresponding chapters.

Functions $\mathrm{dom}(f)$ denotes for a partial mapping $f : A \to B$ the domain, $f \downarrow X$ restricts the mapping f to the domain $\mathrm{dom}(f) \cap X$ and $f \uparrow X$ restricts the mapping to the domain $\mathrm{dom}(f) - X$. The notation $f : A \overset{fin}{\longrightarrow} B$ is the set of finite partial functions from A to B. To write down functions, we use the notation $\emptyset$ to denote the empty function, and $f[a \mapsto b]$ to define the function that returns b for the input a and the same value as f for all other inputs.

Property Maps In JavaScript a special kind of function makes it less complicated to formally define the calculus. We call this kind of function property maps. A property map m is a total function $m : \mathsf{Prop} \to \mathsf{Value}$. It takes a property as argument and returns a value. A property map also fulfills that $m(a) \neq \mathtt{udf}$ for finitely many $a \in \mathsf{Prop}$. In order to obtain a convenient presentation, we define $\{\}$ to be property map that returns always `udf`. The property map $m\{a \mapsto v\}$ is the map, which returns v for the property a, and the same value as m for all the other properties. Hence, we define map lookup $m\$a$ for a property map m and a property a as follows:

$$\begin{array}{rcll} \{\}\$a & = & \mathtt{udf} & \\ m\{a \mapsto v\}\$a & = & v & \\ m\{b \mapsto v\}\$a & = & m\$a & \text{if } a \neq b \end{array}$$

For a property map m, $\mathrm{dom}(m)$ is not defined as it is usual for functions (which will always return Prop because m is a total function over the set Prop):

$$\begin{array}{rcl} \mathrm{dom}(\{\}) & = & \emptyset \\ \mathrm{dom}(m\{a \mapsto v\}) & = & \mathrm{dom}(m) \cup \{a\} \end{array}$$

Implicitly, we have already defined a map update operation $m\{a \mapsto v\}$, because for a map m and $m' = m\{a \mapsto v\}$ it holds that $\forall b \in \mathsf{Prop} : m'\$b = m\$b$ if $a \neq b$, and $m'\$a = v$.

To state that a function is a property map, we write $f : \mathsf{Prop} \overset{prop}{\longrightarrow} \mathsf{Value}$ from now on.

Heaps Heaps are finite functions from references to objects. Objects are property maps. Therefore, we use the notation for map lookup and map update for heaps and the property map lookup notation for objects. For example, $H(i)\$a$ is a map lookup, followed by a property lookup. It returns the content of the property a for the object stored under reference i. The notation $H\{(i)(a) \mapsto v\}$ is a shortcut for: $H[i \mapsto H(i)\{a \mapsto v\}]$ and updates the property a of the object stored under reference i in the heap.

We often write down heaps using JavaScripts' object literals:

$$[i \mapsto \{a : v_a, b : v_b\}, j \mapsto \{\}] \quad .$$

The above heap contains two objects, the first at position i, the second at position j. The object at position i has two properties, a and b. The corresponding values are v_a and v_b. The object at position j is the empty object. We consider the above heap as syntactic sugar for:

$$[i \mapsto \{\}\{a \mapsto v_1\}\{b \mapsto v_2\}, j \mapsto \{\}] \quad .$$

2 JavaScript

JavaScript is a programming language developed originally by Netscape to support dynamic web sites in its favorite web browser Netscape Navigator [85, 86]. At the beginning the language was not equipped with a language definition, and like many other scripting languages it has grown evolutionary, for example PHP [99] has evolved similarly. The language specification of JavaScript is published since 1998 under the name ECMAScript [59]. It was also standardized by ISO/IEC [62]. In the year 2005, the specification was extended under the name ECMAScript for XML (E4X) [61]. Later the fifth edition of ECMA-262 introduced a lot of new features to the language [60].

This dissertation works with the third edition of the language specification, because it is the one that is implemented in most browsers. The E4X specification never was fully supported by a large majority of the browsers. Even if the developer version of a browser (e.g. Firefox 4.0 Beta) does support the new fifth language specification, JavaScript programmers cannot rely on the features of the fifth edition. JavaScrip referes in this dissertation, if not mentioned otherwise, to the third version of the ECMAScript specification from 1998.

The language specifications [59–61] do not define a single programming language, but a set of languages due to the fact that JavaScript is designed to be integrated into an environment. In almost all situations this environment is a web browser. But nowadays JavaScript also found its ways into other environments, for example the node.js project [90] provides a possibility to run JavaScript code on the server side. It is also possible to use JavaScript interpreters that do provide their own environment, for example the Java Project Rhino is a JavaScript interpreter written in Java [113].

2.1 Objects

JavaScript is an object oriented programming language. But it is not class based, as most object oriented programming languages are. In JavaScript objects are hash maps from strings to values; that means each object in JavaScript carries a mapping from property names to values with it that is used to perform property reads and property writes. Additionally to that map, a mechanism based on prototype links is used. Either a constructor call or an object literal is used to create objects. Calling a constructor is done with the keyword `new`, while an object literal is just a list of property/expression pairs, separated by a colon and surrounded by curly braces. The object literal provides a compact notation for object creation. But it lacks the possibility to set the prototype of the object. Consider Program 2.1 for an

Program 2.1 JavaScript – Constructor and Object Literal.

```
1 var proto = { p: "value", q: "value2" };
2 var f = function f(p) {
3   this.p = p;
4   return this;
5 }
6 f.prototype = proto;
7 var o = new f("test");
8 alert(o.p);  // shows "test"
9 alert(o.q); // shows "value2"
```

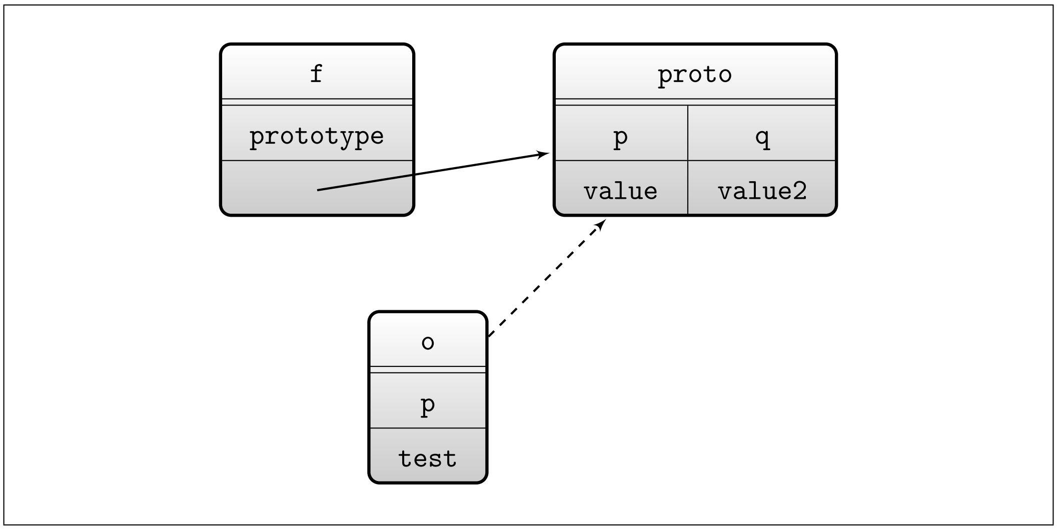


Figure 2.1: Object graph for Program 2.1. The arrow from `f` to `proto` is a common reference to the `proto` object. The dashed arrow represents the implicit prototype link from `o` to `proto`.

example. The object literal in line 1 creates an object with two properties: p and q. The property p carries the string value "value" and the property q the string value "value2". In Figure 2.1 the object proto is represented by a box with a table that contains the corresponding entries. If objects are created using a constructor, as it is done in line 7 of the example by the expression new f("test"), the prototype mechanism is used. The prototype reference for the object created by the new expression is defined by the constructor. In JavaScript, functions are objects, too. Hence, a function may carry properties. If a function does have a property of name prototype, the property value is the source for setting the prototype link for a new object that is created if the function is used as a constructor. In the graphical representation the function f has a property prototype that carries a reference to the object proto. The image visualizes that by an arrow from f to proto. In the example the expression new f("test") creates a new object o, which has a prototype link to the object proto. The image visualizes the prototype link as a dashed arrow to avoid confusion with common references. The property prototype is a common property, except that it is used if the function is called by a new expression. On the other hand the implicit prototype link is special. There does not exist any possibility to directly access the implicit prototype link in JavaScript[1]. The implicit prototype link is used in JavaScript to model inheritance.

For objects without implicit prototype links, property read is simple. If there exists an entry in the hash table the corresponding value is returned. If there does not exist an entry for the property the property read returns the default value undefined. So in the example the expression proto.p return "value", while proto.x returns undefined.

For objects with an implicit prototype link property read works a little bit different. First, the property map of the object itself is considered. If an entry for the property exists in the property map the corresponding value is returned. As an example the property read in line 8 returns "test". But if the property map does not have an entry for the property, the implicit prototype link is used. Therefore, the property read in last line returns "value2". Only, if no object in the prototype link chain contains a corresponding property the default value undefined is returned by a property read.

Property writes do not consider the prototype chain; that means, for an expression o.p either the entry p is created in the hash table of the object o, or the entry is overridden. Further information about the prototype mechanism can be found in the language specification. A more practical approach, on how to use the mechanism, is presented in the book “JavaScript: The Good Parts” from Douglas Crockford [23].

2.2 Functions

In JavaScript functions are first class citizen’s of the language. This means, functions can be passed around as any other value, especially by passing a function as a parameter to other functions. Hence, in this sense, JavaScript is a functional programming language. It fully supports higher order functions, which makes the event driven programming style possible.

The event driven programming style nowadays is not only used inside of browsers, for example the node.js project [90] (originally invented by Ryan Dahl, now organized as a community project) does use the event driven approach in a server setting. This heavily depends on the possibility to work with functions as first class values.

[1] A lot of implementations do provide this access by the special property __proto__. But this special syntax is not supported by ECMAScript.

Program 2.2 JavaScript – Conversion from Float to String.

```
var x = 7;
var y = 6;
var z = x * y;
// Shows a dialog with "The answer to life, ...: 42".
alert("The answer to life, ...: " + z);
```

2.3 Automatic Value Conversion

In a lot of programming languages a static or dynamic type system prohibits the combination of values from different types. For example addition of a string value and a float value is typically not allowed. In contrast, JavaScript tries to not impose any restrictions on the operations the language provides. Instead of throwing a runtime error or presenting a static type error, JavaScript converts values from one type to another, depending on the use of the value. This is convenient for the programmer, if the automatic conversion works as expected. Consider Program 2.2 for a corresponding example.

But a very negative aspect of the conversions is that there are cases in which a conversion results in unexpected behavior. Let us have a look at an example for a situation, where the automatic conversions may be the reason for an error. In Program 2.3 the first two lines define three variables, x, y and v. The variable x is initialized with an object containing the property a carrying the float value 1, while y is initialized with the float value 2, as is v with 0. The next lines are straightforward and work as expected, but in the last two lines a surprising behavior is observed. Even if the assignment y.a = v does not throw any error, the two values displayed to the user in the last line are different. Therefore, setting a property of a float value is not possible, because float values are no objects, and hence do not carry properties. But because the programmer does a property assignment to y, JavaScript converts the float value bound to the variable y into an object, and performs the property assignment using this temporary object. In the next line a property read makes another conversion into an object necessary. But here another temporary object for y is created, which does not carry the property a. Hence, the first alert in the last line shows undefined instead of the expected float value 2. Thiemann reports this surprising behavior first [114] and demonstrates an issue a programmer has to face if he uses the language.

Program 2.3 JavaScript – Objects.

```
var x = { a : 1 };
var y = 2, v = 0;
x.a = x.a + 3;
alert(x.a);
alert(x.b);
v = y.a + 2;
y.a = v;
alert(y.a); alert(v);
```

Part I

$\mathcal{JSR}$ – A Static Type System Based on Recency Abstraction

3 Abstract Interpretation

This chapter introduces the idea of recency abstraction in the setting of abstract interpretation, which is the original setting in which this technique is used [9].

An abstract interpreter simulates a program execution by performing similar execution steps as a common interpreter does. The difference is that the abstract interpreter uses abstract values instead of concrete values. If the abstract values are chosen properly, the abstract interpreter can ensure that the abstract execution terminates yielding a safe approximation of all possible program executions. The connection between abstract values and concrete values is established by a function from concrete values to abstract values, which we call abstraction. The key idea behind the abstraction is to make the set of abstract values significantly simpler than the set of concrete values. Typically the set of abstract values forms a lattice with a finite chain condition [88].

Let us assume that for a simple programming language the concrete interpreter performs an operation o on the two values v_1 and v_2 that results in a value v_3. An abstract interpreter with an abstraction

$$\mathfrak{a} : \text{value} \to \text{absvalue}$$

performs the operation by taking two abstract values a_1 and a_2 with $\mathfrak{a}(v_1) = a_1$ and $\mathfrak{a}(v_2) = a_2$ and executes the abstract counterpart $o_{\mathfrak{a}}$ of the operation o. We choose the abstract counterpart in such a way that it holds:

$$\mathfrak{a}(v_3) \subseteq a_1 \, o_{\mathfrak{a}} \, a_2$$

The above condition is a simple instantiation of the more general guarantee the abstract interpreter ensures, which is that the abstract interpreter yields a safe approximation of all possible program executions. In other words, it ensures that for all possible program executions the result of the program is subsumed by the result the abstract interpreter computes for the program.

Abstract interpreters typically guarantee the termination of the abstract run of the program. Therefore, they have to ensure that it executes bodies of loops and recursive functions only finitely many times. To accomplish termination, different approaches are possible. A typical solution is to chose as abstract values a lattice with a finite chain condition. If it is ensured for all basic operations that the corresponding abstract operations are monotone, the abstract interpreter is guaranteed to terminate. A typical way to get a lattice with a finite chain condition is to chose a finite set of abstract values.

$x +_{\mathfrak{a}} y$	0	$+$	$-$	$\{0,+\}$	$\{0,-\}$	$\{+,-\}$	$\top$
0	0	$+$	$-$	$\{0,+\}$	$\{0,-\}$	$\{+,-\}$	$\top$
$+$	$+$	$+$	$\top$	$\{0,+\}$	$\top$	$\top$	$\top$
$-$	$-$	$\top$	$-$	$\top$	$\{0,-\}$	$\top$	$\top$
$\{0,+\}$	$\{0,+\}$	$\{0,+\}$	$\top$	$\{0,+\}$	$\top$	$\top$	$\top$
$\{0,-\}$	$\{0,-\}$	$\top$	$\{0,-\}$	$\top$	$\{0,-\}$	$\top$	$\top$
$\{+,-\}$	$\{+,-\}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 3.1: Abstract Addition. $\top$ is a shortcut for $\{0,+,-\}$.

In object oriented programming languages, an example for such a choice is to abstract from objects and group different objects into classes. A class is an abstract value of all instances of the class.

$$\begin{array}{rcl} \text{abstract value} & \sim & \text{concrete value} \\ \text{class} & \sim & \text{object} \end{array}$$

As another example let us consider a programming language capable of performing typical mathematical operations on integer numbers. An abstraction $\mathfrak{a} : \mathbb{Z} \to 2^{\{0,+,-\}}$, where 2^A denotes the power set of the set A, might be:

$$\mathfrak{a}(n) := \begin{cases} \{0\} & \text{if } n = 0 \\ \{-\} & \text{if } n < 0 \\ \{+\} & \text{if } n > 0 \end{cases}$$

For convenience we typically do not write the braces around singleton sets. Table 3.1 defines the abstract operation $+_{\mathfrak{a}} : 2^{\{0,+,-\}} \times 2^{\{0,+,-\}} \to 2^{\{0,+,-\}}$, which corresponds to $+$ over $\mathbb{Z}$ (with $\top = \{0,+,-\}$). The table is computed with respect to the following observations:

1. Zero is the neutral element of addition.

2. If $+$ or $-$ is part of one of the operants, then it is part of the result set.

3. If one operant contains $+$, and the other contains $-$, the result contains 0.

The abstract operation $+_{\mathfrak{a}}$ ensures that

$$\forall x, y, z \in \mathbb{Z} : x + y = z \to \mathfrak{a}(z) \subseteq \mathfrak{a}(x) +_{\mathfrak{a}} \mathfrak{a}(y)$$

holds; that is, the abstract addition is a safe approximation of the concrete addition.

Program 3.1 JavaScript – bank account example.

```
 1 function Customer(n) {
 2   return { name : n }; //@C
 3 }
 4 function Account(o) {
 5   return { owner : o }; //@A
 6 }
 7 var c1 = Customer("Lee"), a1 = Account(c1);
 8 c1.account = a1;
 9 var c2 = Customer("Smith"), a2 = Account(c2);
10 c2.account = a2;
11 var c3 = Customer("Hall"), a3 = Account(c3);
12 c3.account = a3;
```

3.1 Recency Abstraction

For a brief introduction to the idea of recency abstraction in an abstract interpretation setting consider the code from Program 3.1. The function Customer is the constructor for customer objects, the function Account is the constructor for account objects. Account objects have a property owner, which is a reference to the owner of the account. Now often it is desirable that also a customer object has a reference back to its account. This link is set by the property assignment in line 8 for the customer with name Lee. The assignment is not done inside the constructor, because this cyclic reference is not possible to establish at the time, when the customer constructor is called (the account object, the target of the reference does not exist at this time). In the next lines the program creates two other objects.

We now start to explain how an abstract interpreter works by executing the above program step by step with a concrete interpreter and, at the same time, with an abstract interpreter. Figure 3.1 contains snapshots of the heaps and abstract heaps, for example executing the program until line 7 (inclusive) creates a heap as presented at the top of Figure 3.1. The concrete execution creates an account and customer object with one link between them. In the next line, the link back from the customer to the account is established, which results in a cyclic object structure in the heap.

The abstract interpreter will execute the program similarly. Instead of creating objects, it creates abstract objects, which we will call classes. A class in the abstract interpreter describes the shape of the objects connected to the class. The benefit of working with classes is that the termination of the abstract interpreter is guaranteed. The termination is ensured by a lot of conditions, but an important one is that the set of classes for a program is finite. Independent from the number of objects in the concrete heap, a class describes the shape of all corresponding

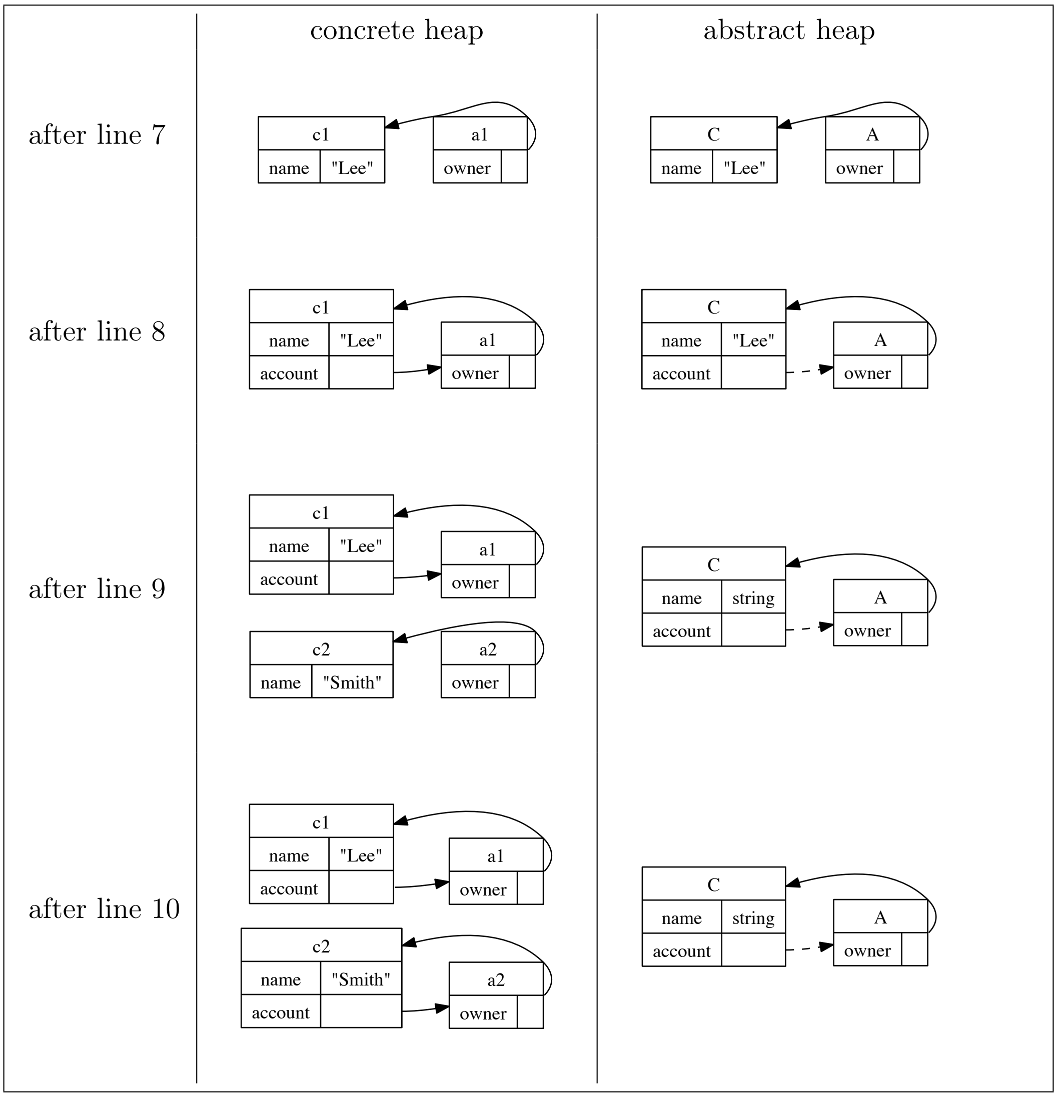


Figure 3.1: Heaps and abstract heaps of Program 3.1. Each line shows the heap and abstract heap as the program creates them. Rectangles represent objects, edges references. In the abstract heap a dashed edge represents an unsafe reference.

objects. A consequence of this is that the assignment in line 8 has a different effect than it has in the concrete execution. The abstract interpreter does not know how many customer objects exist. The only thing it knows is that before the assignment the shape of all customer objects is as shown in the first line of Figure 3.1. Therefore, adding the property `account` to one object only lets the abstract interpreter conclude that after the assignment, there exists one object with a property `account`, and maybe an arbitrary set of objects with the original shape. Hence, the abstract interpreter merges these two possible shapes in such a way that the new class reflects both possibilities. To achieve this, the abstract interpreter adds the property `account` to the class, but marks it as an unsafe reference (indicated in the picture by a dashed arrow); that means, a customer object may have the `account` property, or a customer object may not have the property. If the object contains the `account` property, the value of the property is definitely a reference to an account object.

Generally speaking, the abstract interpreter ensures in each step that the value of a property p of a class C is manipulated conservative; that is: for class C and property p it holds $C_{old}(p) \leq C_{new}(p)$. Hence, updates performed by the abstract interpeter are monotone. We call these updates *weak updates*.

These weak updates, or to stay with our example, the unsafe properties yields of course into an undesirable situation. Later on, even if in the concrete execution all customer objects have a reference to their account object, the abstract interpreter is not aware of this fact. Therefore, reading the corresponding property, for example to check up on the customer's money, leads to a false positive; that is, the abstract interpreter states that there might be an error in the program even if all possible program executions works as expected.

In such a situation the recency abstraction is helpful to not just support weak updates. Why not use an abstraction, such that this typical initialization pattern is handled in the right way? What will happen, if we spend two abstractions for each kind of object, instead of one? If we have two abstract values for customers, and two abstract values for accounts available, we can use two of them for the most recent customer and account, and the other two for all old customers and accounts. The benefit is that the abstract interpreter has a one-to-one relation available for most recent objects, as Figure 3.2 shows. The figure contains two different kind of objects, which is visualized in the figure by different shapes. For each kind of object one class for the most recent object and one class for all the old objects are available.

The consequence of this abstraction is that due to the one-to-one relation, the abstract interpreter is able to perform a *strong update* on the class for the most recent object; that is, it does not respect $C_{old}(p) \leq C_{new}(p)$ but overrides $C_{new}(p)$ with an arbitrary abstract value. In our example this results in adding a safe reference to the account object. Care must be taken, if a new object is created, because the abstract interpreter has to ensure some invariants. The first invariant is that in all situations only one object is considered to be most recent. Figure 3.3 shows how the abstract interpreter with recency evaluates the program step by

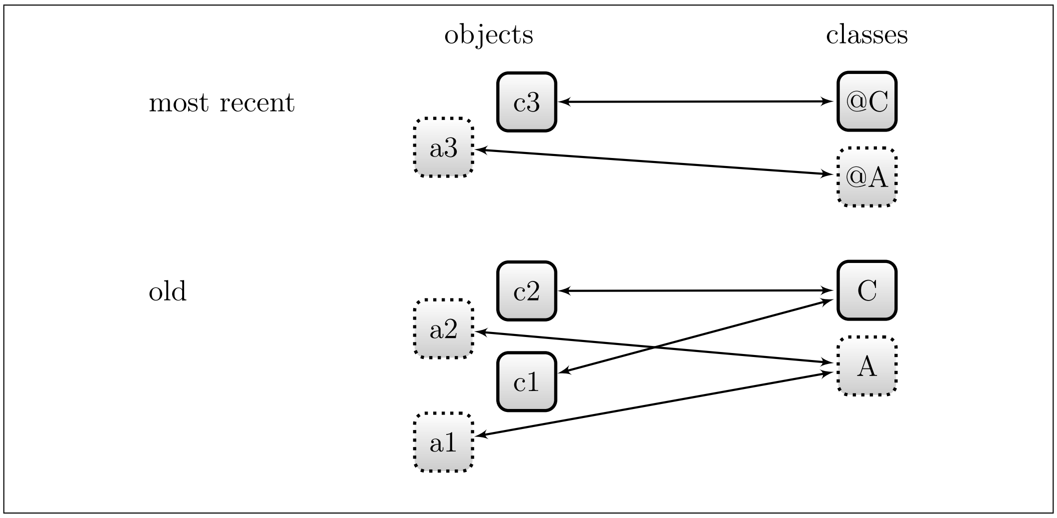


Figure 3.2: Relation between objects and classes. On the left rectangles represent objects. On the right rectangles represent classes. Different kinds of objects are marked by different shapes. Each edge connects an object with its corresponding class.

step. The most recent classes are marked with @, the "usual" classes handling the old objects have names analogous to the class names in the version without recency abstraction. The first thing to notice is that the assignment in line 8 is reflected by a strong update on the most recent class. Before the creation of the second objects the Lee object and its account become old. This is reflected in the graph by removing the @ from the abstraction. Another important step is done before the Hall object is created. In this state the abstract interpreter has already two abstractions for each class, but it needs to introduce a new one for the Hall object. Therefore, it merges the already existing two versions (consider the graph, after line 10). The result of this merging operation is that the name property now does not contain a concrete string anymore. It just contains the information that the property is a string value, because the abstract interpreter has to find a suitable abstraction for "Lee" and "Smith". We call the process of becoming old for an object demotion.

3.2 Type System

Why would we like to use a type system to use the idea of the recency abstraction? There are two answers to this question.

1. A type system can model bidirectional information flow. It is possible to propagate information in the control flow graph in both directions, making the system faster in finding a reason for a program rejection or acceptance.

2. A type system can provide a modular way to reason about functions. This

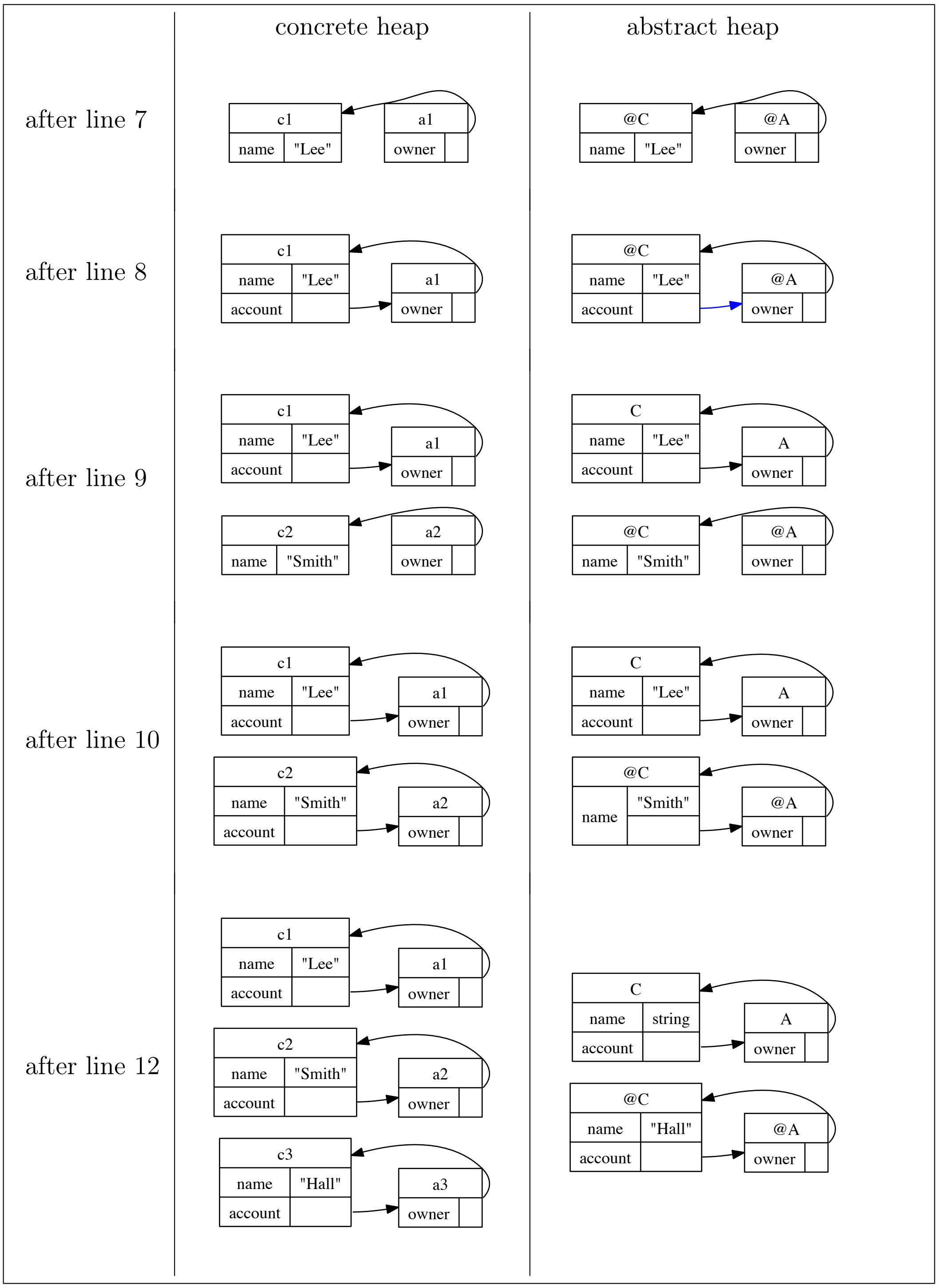


Figure 3.3: Heaps and abstract heaps with recency of Program 3.1. Each line shows the heap and abstract heap as the program creates them. Rectangles represents objects or classes, edges references. In the abstract heap, the rectangles marked with @ are the most recent classes.

is an important feature for analyzing libraries.

Using the idea of recency abstraction in a type system setting instead of an abstract interpreter requires some adjustments. First, a type system, instead of an abstract interpreter, cannot just decide on demand to merge the shapes of most recent object and old objects together in order to ensure that the one-to-one relation is garantied. To keep the two aspects, object creation and object demotion, separated, the type system has two parts:

- A flow-sensitive part handling the most recent objects
- A flow-insensitive part handling old objects

Due to the flow sensitivity and the one-to-one relation the type system supports strong updates on most recent objects (that means, an assignment overrides the content of a property). For the flow-insensitive part, the type system handles assignments as usual. It ensures that the type of a written value is a subtype of the type of the property.

4 The Formal Calculus

This chapter presents two core calculi. Both contain the most important features of JavaScript from the point of view of this work. Section 4.1 presents the first calculus – JavaScript Core ($\mathcal{JSC}$). It is a lambda calculus with imperative objects. It is similar an other core calculus suggestion in the literure [50]. In Section 4.2 we present the second calculus, JavaScript Core Recency ($\mathcal{JSR}$), which extends $\mathcal{JSC}$ with recency information. We present a formal syntax for each of the systems, a small-step operational semantics and relate the behavior of both systems. Section 4.3 shows that the only difference between $\mathcal{JSC}$ and $\mathcal{JSR}$ is that in $\mathcal{JSR}$ some additional steps are performed to model the demotion of objects. Section 4.4 presents the static semantics of $\mathcal{JSR}$. Section 4.5 proves soundness of $\mathcal{JSR}$. Section 4.6 proves decidability of type checking.

The key point to model recency abstraction in a sound manner is to model the change for a most recent object to an old object in the right way. We make demotion explicit in $\mathcal{JSR}$ by introducing an extra demotion expression. To distinguish between most recent objects and old objects we model the semantics of $\mathcal{JSR}$ with two heaps. One heap contains all the old objects, the other one contains the most recent objects. The demote operation than has to move the object from the most recent heap into the summary heap.

4.1 JavaScript Core ($\mathcal{JSC}$)

$\mathcal{JSC}$ is very basic because its main task is to show the similarities between a subset of JavaScript and $\mathcal{JSR}$. It is an intermediate step between JavaScript and $\mathcal{JSR}$ to convince the reader that adding recency to the semantics does not modify the behavior of the system.

4.1.1 Syntax

Figure 4.1 presents the abstract syntax of $\mathcal{JSC}$. The language contains values v, which are either recursive functions $\mathtt{rec}\, f(x).e$ with function name f, parameter name x and body e, or `udf` or a reference $\vec{\imath}$. Programmers are not allowed to directly include references in program source code. They are only used in the dynamic semantics as intermediate values.

Top expressions are values or let expressions $\mathtt{let}\ x = s\ \mathtt{in}\ e$. Let expressions are used to sequentially execute the program. A let expression binds the result of executing the simple expression s to the variable x and then afterwards executes

$$
\begin{array}{rlll}
\mathsf{Prop} \ni & a & & \text{countable set of property names} \\
\mathsf{Variable} \ni & x, f & ::= x_1 \mid x_2 \mid \ldots & \text{countable set of variables} \\
\mathsf{Value} \ni & v & ::= \mathtt{rec}\, f(x).e \mid \mathtt{udf} \mid \vec{i} & i \in \mathbb{N} \\
\mathsf{TopExpr} \ni & e & ::= w \mid \mathtt{let}\ x = s\ \mathtt{in}\ e & \\
\mathsf{Va(r|l)} \ni & w & ::= x \mid v & \\
\mathsf{SExpr} \ni & s & ::= \ w \mid w(w) \mid \mathtt{new} \mid w.a \mid w.a := w & \\
\mathsf{Expr} \ni & d & ::= e \mid s &
\end{array}
$$

Figure 4.1: $\mathcal{JSC}$ – syntax. Phrases marked in gray are not written by the programmer. They arise as intermediate steps in the semantics.

the top expression e. This syntax is similar to A-normal form to economize the proofs [39].

A simple expression is a variable or value (w), a function application $w(w)$, a `new` expression, an expression reading a property of an object $w.a$ or an expression writing to an object $w.a := w$.

The distinction between top expressions and simple expressions is utilized to make program execution explicit and simple. This is no restriction because the programmer can always introduce a new variable with a let expression to put the result of an arbitrary expression at a place where only variables are allowed. Often, the distinction between top expressions and simple expressions is not necessary, for example if we establish some properties for both kind of expressions. Hence, we introduce the metavariable d as an expression.

Convention 4.1.1. *The sets Prop, Variable and* $\mathbb{N}$ *are pairwise disjoint sets.* `udf` *is not an element of Prop or Variable.*

Figure 4.2 defines the set of free variables fv(d) for expressions and values. It is straightforward and similar to the corresponding definition in the lambda calculus.

Definition 4.1.2 (Closedness)**.** *A value v is closed if fv(v) $= \emptyset$. An expression d is closed if fv(d) $= \emptyset$.*

Some of our examples make use of additional features of JavaScript that are not part of the formal calculus. We use conditional expressions, functions with multiple parameters, and methods as well as several basic values like floats, booleans and strings. Chapter 6 describes how to include these extensions into the formal calculus.

4.1.1.1 Example

Program 4.1 shows the relation between JavaScript and $\mathcal{JSC}$. The program creates an object, adds a property `x` to the object, stores the function `getX` in a variable an executes the function in the last line. Program (a) on the left side is written

$$
\begin{aligned}
\mathrm{fv}(x) &= \{x\} \\
\mathrm{fv}(\mathtt{rec}\, f(y).e) &= \mathrm{fv}(e) - \{f, y\} \\
\mathrm{fv}(\mathtt{udf}) &= \mathrm{fv}(\vec{i}) = \mathrm{fv}(\mathtt{new}) = \emptyset \\
\mathrm{fv}(\mathtt{let}\ x = s\ \mathtt{in}\ e) &= \mathrm{fv}(s) \cup (\mathrm{fv}(e) - \{x\}) \\
\mathrm{fv}(w_1(w_2)) &= \mathrm{fv}(w_1) \cup \mathrm{fv}(w_2) \\
\mathrm{fv}(w.a) &= \mathrm{fv}(w) \\
\mathrm{fv}(w_1.a := w_2) &= \mathrm{fv}(w_1) \cup \mathrm{fv}(w_2)
\end{aligned}
$$

Figure 4.2: $\mathcal{JSC}$ – free variables of values and expressions. Inductive.

Program 4.1 JavaScript, $\mathcal{JSC}$ – Property Access, Method Calls. A small program that creates a new object, adds a function as a property to the object and executes the function as a method call.

```
1 var o = {};
2 o.x = 5.0;
3 getX = function getX(this) {
4   return this.x
5 };
6 getX(o);
```

(a) JavaScript.

```
1 let o = new in
2 let dummy1 = o.x := 5.0 in
3 let getX := (rec getX(this).
4   this.x)
5 in
6   getX(o)
```

(b) $\mathcal{JSC}$ with method extension.

in JavaScript syntax, while Program (b) on the right side is written in the formal syntax of $\mathcal{JSC}$.

To improve the readability of examples written in $\mathcal{JSC}$, we utilize the following shortcuts: If a variable bound in a let expression is not used, as in line 2 of the example, we use o.p := e1; e2 instead of **let** dummy = o.p := e1 **in** e2. If a function makes no use of its name we use $\lambda x.e$ as a shortcut for $\mathtt{rec}\, f(x).e$. The source of Program 4.1 using these shortcuts can be written as:

```
let o = new in
  o.x := 5.0;
  getX := λy.(this.x);
  getX(o)
```

4.1.1.2 Substitution

Figure 4.3 defines the capture avoiding substitution $d[x_i \mapsto v]$ inductively. Basically the variable x_i is replaced by the value v at all places inside an expression d. As usual, the substitution stops if a local expression (a let or a rec expres-

$$
\begin{aligned}
x_i[x_i \mapsto v] &= v \\
x_j[x_i \mapsto v] &= x_j \\
\mathtt{udf}[x_i \mapsto v] &= \mathtt{udf} \\
\vec{i}[x_i \mapsto v] &= \vec{i} \\
(w_1(w_2))[x_i \mapsto v] &= w_1[x_i \mapsto v](w_2[x_i \mapsto v]) \\
\mathtt{new}[x_i \mapsto v] &= \mathtt{new} \\
(w.a)[x_i \mapsto v] &= w[x_i \mapsto v].a \\
(w_1.a := w_2)[x_i \mapsto v] &= w_1[x_i \mapsto v].a := w_2[x_i \mapsto v] \\
(\mathtt{let}\ x_i = s\ \mathtt{in}\ e)[x_i \mapsto v] &= \mathtt{let}\ x_j = s[x_i \mapsto v]\ \mathtt{in}\ e \\
(\mathtt{let}\ x_j = s\ \mathtt{in}\ e)[x_i \mapsto v] &= \mathtt{let}\ x_j = s[x_i \mapsto v]\ \mathtt{in}\ (e[x_i \mapsto v]) && \text{if } x_j \notin \mathrm{fv}(v) \\
(\mathtt{rec}\, x_j(x_k).e)[x_i \mapsto v] &= \mathtt{rec}\, x_j(x_k).e && \text{if } i = j \text{ or } i = k \\
(\mathtt{rec}\, x_j(x_k).e)[x_i \mapsto v] &= \mathtt{rec}\, x_j(x_k).(e[x_i \mapsto v]) && \text{if } x_j \notin \mathrm{fv}(v) \\
& && \text{and } x_k \notin \mathrm{fv}(v) \\
d[x_i, x_j \mapsto v_i, v_j] &= (d[x_i \mapsto v_i])[x_j \mapsto v_j] && \text{if } x_j \notin \mathrm{fv}(v_i)
\end{aligned}
$$

Figure 4.3: $\mathcal{JSC}$ – capture avoiding substitution. Inductive. If not further specified, let $i \neq j$ and $i \neq k$. The last line defines the simultaneous substitution of two variables.

sions) overlays the scope of the variable x_i. Hence, inside the body of a function expression replacement happens, if x_i is unequal to the function name and to the parameter name. If it is equal to one of these two names, no substitution takes place inside the function body.

The substitution function is a partial function due to its side conditions $x_j \notin \mathrm{fv}(v)$ and $x_k \notin \mathrm{fv}(v)$. As an example consider $(\mathtt{rec}\, f(x).y)[y \mapsto f]$, which is not defined. The side conditions ensure that substitution is capture avoiding [100]. The partial definition is not problematic because the evaluation only substitutes closed values.

Lemma 4.1.3 (Free variables)**.** *For an expression d it holds that for all variables x*

$$fv(d[x \mapsto v]) \subseteq (fv(d) - \{x\}) \cup fv(v)$$

Proof by induction over the structure of expressions. We assume that $x = x_i$ for a fixed i.

- *Case $d = x_j$*: There are two cases, $i = j$ and $i \neq j$. Both cases yields the desired result.

- *Case $d = v'$*: All cases except the function case are trivial. Assume $v' = \mathtt{rec}\, x_k(x_j).e_b$. It holds $\mathrm{fv}(v') = \mathrm{fv}(e_b) - \{x_k, x_j\}$.
  - *Case $i = j$* or *$i = k$*: The lemma holds because $(\mathtt{rec}\, x_k(x_j).e_b)[x_i \mapsto v] = \mathtt{rec}\, x_k(x_j).e_b$.
  - *Case $i \neq j$* and *$i \neq k$*: It holds $x_j \notin \mathrm{fv}(v)$, $x_k \notin \mathrm{fv}(v)$ and $(\mathtt{rec}\, x_k(x_j).e_b)[x_i \mapsto v] = \mathtt{rec}\, x_k(x_j).(e_b[x_i \mapsto v])$. By induction, it holds
  $$\mathrm{fv}(e_b[x_i \mapsto v]) \subseteq (\mathrm{fv}(e_b) - \{x_i\}) \cup \mathrm{fv}(v) \quad .$$

  We conclude:

  $$\begin{aligned}\mathrm{fv}(d[x_i \mapsto v]) &= \mathrm{fv}((\mathtt{rec}\, x_k(x_j).e_b)[x_i \mapsto v]) \\ &= \mathrm{fv}(\mathtt{rec}\, x_k(x_j).(e_b[x_i \mapsto v])) \\ &= \mathrm{fv}(e_b[x_i \mapsto v]) - \{x_k, x_j\} \\ &\subseteq ((\mathrm{fv}(e_b) - \{x_i\}) \cup \mathrm{fv}(v)) - \{x_k, x_j\} \\ &= ((\mathrm{fv}(e_b) - \{x_k, x_j\}) - \{x\}) \cup \mathrm{fv}(v) \\ &= (\mathrm{fv}(e) - \{x_i\}) \cup \mathrm{fv}(v) \quad .\end{aligned}$$

□

We use in our dynamic semantics simultaneous substitution of two variables $d[x_i, x_j \mapsto v_i, v_j]$. We can define it as $d[x_i \mapsto v_i][x_j \mapsto v_j]$. This definition works well for our purpose because we always use it with closed values v_i and v_j. Therefore, the case in which v_i contains x_j as a free variable cannot occur.

$$
\begin{array}{rcccl}
H & \in & \mathsf{Heap} & = & \mathbb{N} \xrightarrow{\mathit{fin}} \mathsf{Object} \\
o & \in & \mathsf{Object} & = & \mathsf{Prop} \xrightarrow{\mathit{prop}} \mathsf{Value} \\
\mathcal{E} & ::= & \Box \mid \texttt{let}\ x = s\ \texttt{in}\ \mathcal{E} & &
\end{array}
$$

$$
\begin{array}{llcll}
\text{S0-App} & H, (\mathtt{rec}\, f(x).e)(v) & \to_0 & H, e[f, x \mapsto \mathtt{rec}\, f(x).e, v] & \\
\text{S0-Let} & H, \texttt{let}\ x = v\ \texttt{in}\ e & \to_0 & H, e[x \mapsto v] & \\
\text{S0-New} & H, \texttt{new} & \to_0 & H[i \mapsto \{\,\}], \vec{i} & i \notin \mathrm{dom}(H) \\
\text{S0-Rd} & H, \vec{i}.a & \to_0 & H, H(i)\$a & i \in \mathrm{dom}(H) \\
\text{S0-Wrt} & H, \vec{i}.a := v & \to_0 & H\{(i)(a) \mapsto v\}, \texttt{udf} & i \in \mathrm{dom}(H)
\end{array}
$$

$$
\text{S0-Let}' \quad \frac{H, s \to_0 H', \mathcal{E}[v]}{H, \texttt{let}\ x = s\ \texttt{in}\ e'' \to_0 H', \mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e'']}
$$

Figure 4.4: $\mathcal{JSC}$ – small-step operational semantics. Inductive. The semantics is well-defined, because the implicit precondition of rule S0-Let′ is always fulfilled. Please consider Lemma 4.1.4 and Lemma 4.1.5 for additional explanation.

4.1.2 Dynamic Semantics

Figure 4.4 presents the small step operational semantics for $\mathcal{JSC}$. The relation $\to_0$ takes a heap H and an expression d and yields a possibly modified heap together with a new expression. A heap $H \in \mathsf{Heap}$ is a finite mapping from integers to property maps ($h \in \mathsf{Object}$), which represent objects. Property maps are finite maps from properties to values. The evaluation makes use of the evaluation context $\mathcal{E}$.

The rule S0-App defines function application. It substitutes the parameter x with the value v and the function name f with the function itself inside of the body e of the function. Hence, on the right hand side of the reduction rule S0-App the new expression is $e[f, x \mapsto \mathtt{rec}\, f(x).e, v]$. The heap stays unchanged.

The second rule (S0-Let) substitutes the variable x inside of the body of the let expression with the value v and does not change the heap.

The rule S-New creates a new object in the heap H at position i if the position is free. The object does not have any properties. Hence, $\{\,\}$, the empty property map, is stored in the heap.

S0-Rd states how property read works. $\vec{i}.a$ reads the property from the object $H(i)$. Property lookup yields a value $H(i)\$a$, which is the new expression of the reduction rule. The lookup in a property map is written $o\$a$ for an object o and a property a. It is defined in Section 1.3. The heap content does not change.

The rule S0-Wrt updates or adds a property. $\vec{i}.a := v$ adds or modifies the property a of the object $H(i)$ to v. Map update works on property maps in exactly the same manner as it works for maps, it changes the binding for a if it exists, and it adds a new binding for a if the binding does not exist. It is defined in Section 1.3. The result of a write operation is always `udf`.

The rule S0-Let′ is the context rule that evaluates the right hand side of let expression s if it is not a value. It makes use of the evaluation context to flatten the let expressions. The following example demonstrates the described behavior.

4.1.2.1 Example

The example shows why the rule S0-Let′ needs to flatten let expressions. The program has no practical purpose but to show how the evaluation works and additionally demonstrate how an infinity loop works. It does not use objects. Hence, the heap is omitted in this example for convenience.

```
let f = rec f(x).
  let o = f(x) in
    o
in
let u = f(udf) in
  u
```

Because the right hand side of the expression, **rec** $f(x)$... o, is a value, the execution substitutes the right hand side of the let expression into the body of the **let** expression. The rule S0-Let is used for this substitution, and the result is:

```
let u = (rec f(x).let o = f(x) in o)(udf) in
  u
```

The value (**rec** $f(x)$.**let** $o = f(x)$ **in** o) is a function. We write F as a shortcut for it. The right hand side of the outer **let** is a function application, which evaluates into **let** o = F(**udf**) **in** o using S0-App to substitute f with its body (F) and x with **udf**. Now notice that F itself is a value, while **let** o = F(**udf**) **in** o is a top expression. Simply changing the right hand side of a let expression from a function application into the body of the function will result in invalid code because on the right hand side of a let, let expressions are forbidden. Here, the evaluation context helps, because we can write every top expression as $\mathcal{E}(w)$ for some variable or value w (see Lemma 4.1.4). In our case it holds:

$$\mathcal{E}[\mathsf{o}] = \textbf{let } \mathsf{o} = \mathsf{F}(\textbf{udf}) \textbf{ in } \mathsf{o}$$

Hence, we can apply S0-Let′ and our top expression evaluates to:

```
let o = F(udf) in
let u = o in
  u
```

Here, the let expressions are flatted, and the resulting expression is a correct top expression for $\mathcal{JSC}$.

$$\begin{aligned}
\text{height}(x) &= 0\\
\text{height}(\texttt{rec}\, f(y).e) &= \text{height}(e) + 1\\
\text{height}(\texttt{udf}) &= \text{height}(\vec{i}) = \text{height}(\texttt{new}) = 0\\
\text{height}(\texttt{let}\ x = s\ \texttt{in}\ e) &= \max(\text{height}(e), \text{height}(s)) + 1\\
\text{height}(w_1(w_2)) &= \max(\text{height}(w_1), \text{height}(w_2))\\
\text{height}(w.a) &= \text{height}(w)\\
\text{height}(w_1.a := w_2) &= \max(\text{height}(w_1), \text{height}(w_2))
\end{aligned}$$

Figure 4.5: $\mathcal{JSC}$ – height of expressions. Inductive.

4.1.3 Formal Properties

4.1.3.1 Well-Defined

Lemma 4.1.4 (Context for top expressions)**.** *For all top expressions e, there exists exactly one evaluation context $\mathcal{E}$ and a value or variable w, such that $e = \mathcal{E}[w]$.*

Proof by contradiction. We assume the lemma is not correct. As a consequence, there exists at least one top expression such that there exits no $\mathcal{E}[w]$ with $\mathcal{E}[w] = e$. Because there may exist more than one of such an expression, let e be an expression with the smallest height (Figure 4.5), for which $\mathcal{E}$ and w do not exist. Either e is a let expression, or a variable, or a value. The two last cases are not possible because for $\mathcal{E} = \Box$ it holds $\mathcal{E}[w] = e$.

Now, assume $e = \texttt{let}\ x = s\ \texttt{in}\ e'$. Since e' is a subexpression of e, the height of e' is smaller then the height of e. Since e is an expression with minimal height that does not have a suitable context, for the expression e' there exists $\mathcal{E}'[w]$ with $\mathcal{E}'[w] = e'$. As a consequence, the context $\mathcal{E} = \texttt{let}\ x = s\ \texttt{in}\ \mathcal{E}'$ exists and it holds $\mathcal{E}[w] = e$. This is a contradiction to our assumption that the set of expression, for which no suitable $\mathcal{E}[w]$ exists, was not empty.

Hence, it is left to prove the uniqueness of $\mathcal{E}$ and w. The uniqueness of $\mathcal{E}$ and w is a direct implication from the definition of the evaluation context. □

Lemma 4.1.5 (Context existence)**.** *For heaps H, H', a simple expression s and a top expression e, if $H, s \to_0 H', e$, then there exists exactly one evaluation context $\mathcal{E}$ and a value or variable w, such that $\mathcal{E}[w] = e$.*

Proof by induction over $\to_0$.

- *Case* S0-App, S0-Let, S0-New, S0-Rd, S0-Wrt: The right hand side of each rule is a top expression. We apply Lemma 4.1.4.
- *Case* S0-Let′: By induction the implicit precondition of the rule S0-Let′ is fulfilled. We have to proof that for $\mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e'']$ there exists exactly one

evaluation context $\mathcal{E}'$ and a value or variable w such that $\mathcal{E}'[w] = \mathcal{E}[\texttt{let } x = v \texttt{ in } e'']$. By definition of $\mathcal{E}$ is holds that $\mathcal{E}[\texttt{let } x = v \texttt{ in } e'']$ is a top level expression. Hence, we apply Lemma 4.1.4.

□

The above lemma ensures that the implicit precondition in the evaluation rule S0-Let′ is always fulfilled and that the evaluation context is unique. This fact is used in a lot of the following proofs implicitly.

Lemma 4.1.6 (Substitution with closed values)**.** *For a closed expression d an evaluation step $H, d \rightarrow_0 H', d'$ yields a closed expresion d'. Substitution applied during evaluation only takes closed values as its parameter.*

Proof by induction over $\rightarrow_0$. A simple case distinction over the rules in Figure 4.4 proves the lemma. □

A consequence of Lemma 4.1.6 is that our definition of simultaneous substitution, even if it is a partial function, is always applicable during evaluation. Hence, the side conditions of the substitution definition are never the reason for an expression not to evaluate further.

4.1.3.2 Determinism

> Determinism (specifically causal determinism) is the concept that events within a given paradigm are bound by causality in such a way that any state (of an object or event) is completely, or at least to some large degree, determined by prior states. [119]

Following the above definition, the semantics of $\mathcal{JSC}$ is deterministic because it is completely predictable what the next step in program execution will be. The only exception is the rule S0-New, which only enforces to pick a reference that is not already used for another object. Hence, it does not specify what concrete reference is used for the new object.

Therefore, we can state that the semantic is deterministic if we ignore memory positions of objects. This is typically a statement that is strong enough and sufficient for a high level programming language.

Definition 4.1.7 (Equivalence modulo memory allocation)**.** *An expression e_1 is equivalent modulo memory allocation to e_2 ($e_1 \equiv e_2$) if there exists a bijection $b : \mathbb{N} \rightarrow \mathbb{N}$ with $e_1 \equiv_b e_2$. (cf. Figure 4.6,* EQ*)*

A heap H_1 is equivalent modulo memory allocation to H_2 if there exists a bijection $b : \mathbb{N} \rightarrow \mathbb{N}$ with $H_1 \equiv_b H_2$. (cf. Figure 4.6, EQ)

Lemma 4.1.8. *The relation $\equiv$ is an equivalence relation.*

Proof. We have to prove that the relation is reflexive, symmetric and transitive.

$$\textsc{EQbVar}\quad x \equiv_b x \qquad \textsc{EQbConst}\quad \texttt{udf} \equiv_b \texttt{udf} \qquad \textsc{EQbRef}\ \frac{i = b(j)}{\vec{i} \equiv_b \vec{j}} \qquad \textsc{EQbFix}\ \frac{e_1 \equiv_b e_2}{\texttt{rec}\, f(x).e_1 \equiv_b \texttt{rec}\, f(x).e_2}$$

$$\textsc{EQbLet}\ \frac{e_1 \equiv_b e_2 \qquad s_1 \equiv_b s_2}{\texttt{let}\ x = s_1\ \texttt{in}\ e_2 \equiv_b \texttt{let}\ x = s_2\ \texttt{in}\ e_2} \qquad \textsc{EQbApp}\ \frac{w_1 \equiv_b w_2 \qquad w'_1 \equiv_b w'_2}{w_1(w'_1) \equiv_b w_2(w'_2)}$$

$$\textsc{EQbRead}\ \frac{w_1 \equiv_b w_2}{w_1.a \equiv_b w_2.a} \qquad \textsc{EQbWrite}\ \frac{w_1 \equiv_b w_2 \qquad w'_1 \equiv_b w'_2}{w_1.a := w'_1 \equiv_b w_2.a := w'_2}$$

$$\textsc{EQbHeap}\ \frac{\begin{array}{c}\forall j \in \mathrm{dom}(H_2) : \exists^{=1} i \in \mathrm{dom}(H_1) : b(i) = j \\ \forall i \in \mathrm{dom}(H_1) : \forall p \in \mathrm{dom}(H_1(i)) : H_1(i)\$p \equiv_b H_2(b(i))\$p\end{array}}{H_1 \equiv_b H_2}$$

$$\textsc{EQ}\ \frac{\exists b : \mathbb{N} \to \mathbb{N}, b \text{ is a bijection} \qquad X \equiv_b X'}{X \equiv X'}$$

Figure 4.6: $\mathcal{JSC}$ – equivalence modulo memory allocation. The inductive definition establishes an equivalence class over expressions and heaps. Please note that $\exists^{=1}$ ensures that there exists exactly one element in the domain of H_1 for each j in the domain of H_2 (cf. Section 1.3). X is an arbitrary syntactic entity.

- *Case* Reflexivity: $\equiv_{id}$ is reflexiv. Hence, there exists a bijection b ($b = id$) that proves the reflexivity of $\equiv$.
- *Case* Symmetry: We prove that for X and X', it holds $X \equiv_b X'$ iff $X' \equiv_{b^{-1}} X$. The proof is immediate by induction over the definition of $\equiv_b$.
- *Case* Transitivity: For $X \equiv_b X'$ and $X' \equiv'_b X''$ it holds $X \equiv_{b' \circ b} X''$. $b' \circ b$ exists, because the function b and b' are both total. The proof is immediate by induction over the definition of $\equiv_b$.

□

Lemma 4.1.9 (Substitution determinism)**.** *For $e_1 \equiv_b e_2$ it holds for all x and $v_1 \equiv_b v_2$:*

$$e_1[x \mapsto v_1] \equiv_b e_2[x \mapsto v_2]$$

Proof by induction over the structure of e_1. Without loss of generality we assume $x \in \mathrm{fv}(e_1)$. All cases are immediate or by induction. □

Lemma 4.1.10 (Determinism)**.** *For $H_1 \equiv_b H_2$ and $e_1 \equiv_b e_2$ with $H_1, e_1 \longrightarrow H_1', e_1'$ and $H_2, e_2 \longrightarrow H_2', e_2'$ then there exists a bijection $b' : \mathbb{N} \to \mathbb{N}$ with*

$$H_1' \equiv_{b'} H_2' \qquad \textit{and} \qquad e_1' \equiv_{b'} e_2'$$

Proof by induction over $\longrightarrow$.

- *Case* S-App, S-Let: We apply Lemma 4.1.9.
- *Case* S-New: Inversion of the definition of $\equiv_b$ yields the desired result, because the side condition of the rule ensures that fresh memory places are chosen.
- *Case* S-Rd, S-Wrt: immediate.
- *Case* S-Let′: immediate by induction.

□

Corollary 4.1.11. *For an closed expression e with $\emptyset, e \longrightarrow^n H_1, e_1$ and $\emptyset, e \longrightarrow^n H_2, e_2$ then e_1 is equivalent modulo memory allocation to e_2 and H_1 is equivalent modulo memory allocation to H_2.*

Proof by induction over $\longrightarrow^*$. A simple consequence from Lemma 4.1.10. □

Convention 4.1.12. *A configuration (H, e) is a pair containing a heap and an expression. From now on, two configurations are considered syntactically equal if their heaps and expressions are equivalent modulo memory allocation (with the same b).*

4.2 JavaScript Core Recency ($\mathcal{JSR}$)

Recency is straightforward to handle in an abstract interpretation setting (cf. Chapter 3), but lifting the concept to types requires some care. There are four key aspects that need to be reflected in the design of the type system.

1. There must be distinct types for recent objects and aged objects: singleton and summary pointer types.
2. Singleton pointer types must be subject to strong update (cf. Chapter 3).
3. Singleton pointer types must be "demotable" to summary types.
4. While an abstract interpreter can demote a singleton pointer to a summary pointer "online" at the respective `new` expressions, a type system must demote more conservatively to stay tractable.

This section adheres to the syntax defined in Section 4.2.2. It is similar to the syntax of $\mathcal{JSC}$.

The main difference between $\mathcal{JSC}$ and $\mathcal{JSR}$ is that $\mathcal{JSR}$ provides additional structure for the heap while $\mathcal{JSR}$ does. The structure $\mathcal{JSR}$ introduces is that each object is grouped by an abstract location from a set $\mathsf{Location}$. Every reference $\vec{i}$ is tagged by an abstract location $l \in \mathsf{Location}$. A reference in $\mathcal{JSR}$ is written $(q\ l, i)$ for $i \in \mathbb{N}$ and $l \in \mathsf{Location}$. The qualifier q is part of the reference for the following reason.

$\mathcal{JSR}$ runs with two heaps, a most recent heap and a summary heap. Inside of the most recent heap only one object is allowed at the same time for each abstract location l. The object that exists in the most recent heap is the most recent object for the abstract location. The summary heap collects all aged objects. Hence, it may contain more than one object for an abstract location l.

q is an additional qualifier that distinguishes two situations. First, if the qualifier is ˜ then the object is a summary object and lies inside of the summary heap. Second, the object is a most recent object, and it lives in the most recent heap. The qualifier for this situation is @.

We use two heaps instead of one and attach the references by a qualifier because based on the qualifier it is now easy to determine if an object is most recent object or an aged object.

Please keep in mind that in $\mathcal{JSR}$ the following invariants hold to keep the system tractable:

Definition 4.2.1 ($\mathrm{INV}_{\mathrm{OH}}$)**.** *The invariant about "most recent **O**bjects in the most recent **H**eap" states that for each abstract location at most one object exists in the most recent heap.*

Definition 4.2.2 ($\mathrm{INV}_{\mathrm{DH}}$)**.** *The invariant about "**D**isjoint **H**eaps" states that the domain of the most recent heap and the domain of the summary heap are disjoint.*

4.2.1 Examples

Before we present the formal system of $\mathcal{JSR}$, we discuss by the use of some examples how we can extend our calculus $\mathcal{JSC}$ to support recency abstraction.

4.2.1.1 Object Demotion

The following example will introduce the idea of recency, and how it could be modeled in a system that is similar to $\mathcal{JSC}$. We make use in this section of some abbreviations as in Section 4.1. We also use some base types like `float` or `boolean`. A heap is a finite function from references to objects, as defined in Figure 4.4. Program 4.2 demonstrates a problem that arises by introducing recency to a system, in which it is possible to distinguish if an object is most recent by looking at the reference. A reference needs to change from $(@l, i)$ into $(\tilde{}l, i)$ because a new object for the abstract location l is created by a new expression.

Program 4.2 $\mathcal{JSC}$ – summary pointers.

```
let x = new^l in x.a := 42.0;
let y = new^l in y.a := "flush";
x.a
```

In the first line a new empty object is created and the property a is set to the float value 42. The next line creates a new object for the same abstract location and adds the property a to the new object. This time the property value is a string. To stay as close as possible to $\mathcal{JSC}$ our goal is to model binding of a value to a variable by substitution.

Let us have a look at the semantics of $\mathcal{JSC}$ and how the program will evaluate in this calculus. The semantics of $\mathcal{JSC}$ yields after three evaluation steps the expression **let** y = new **in** y.a := "flush" **in** $\vec{0}$.a together with a heap that contains an object $\{a : 42\}$ at position 0. One evaluation step further we get **let** y = $\vec{1}$ **in** y.a := "flush" **in** $\vec{0}$.a while the heap is $[0 \mapsto \{a : 42\}, 1 \mapsto \{\,\}]$.

Suppose our semantics for $\mathcal{JSR}$ behaves similar. After one evaluation step we get **let** x = $(@l, 0)$ **in** x.a := 42; **let** y = new^l **in** y.a := "flush"; x.a. The reference is tagged by @ and l, because a new expression always creates a most recent object for its abstract location.

Two steps later the heap is updated, and the resulting expression is **let** y = new^l **in** y.a := "flush" **in** $(@l, 0)$.a. The summary heap is empty and the most recent heap is $[0 \mapsto \{a : 42\}]$. One step later spontaneous guess would lead to the expression **let** y = $(@l, 1)$ **in** y.a := "flush" **in** $(@l, 0)$.a and the most recent heap $[(l, 0) \mapsto \{a : 42\}, (l, 1) \mapsto \{\}]$. But this breaks INV_{OH}. To establish INV_{OH} the system moves the old l object into the summary heap. Hence, the summary heap is $[(l, 0) \mapsto \{a : 42\}]$ and the most recent heap $[(l, 1) \mapsto \{\,\}]$ contains the new object.

The problem caused by the movement is that the reference $(@l, 0)$, which is part of the expression, points to an object in the most recent heap that does not exist after the movement. Therefore, the semantics for $\mathcal{JSR}$ has to adjust all references when it moves objects from the most recent heap into the summary heap.

Our result after four evaluation steps should yield

```
let y = (@l, 1) in y.a := "flush";
 (~l, 0).a
```

To model the movement from the most recent heap into the summary heap, and to adjust the references, such that the system does not get stuck due to wrong qualifiers, we introduce a new top level expression. We denominate the expression demotion and write it $\natural^l e$. It moves the most recent object for the abstract location l into the summary heap and adjusts all references inside of e, such that they point to the appropriate summary heap object.

In $\mathcal{JSR}$ Program 4.2 is enriched with these demotions. This step can be done in

Program 4.3 $\mathcal{JSC}$ – functions.

```
1 ♮^l let x = new^l in
2 let f = λ_.x.a in
3 x.a := 42;
4 ♮^l let y = new^l in
5 let z = f(0) in
6   z
```

a preprocessing step, hence the programmer does not need to write the demotions down. The corresponding program is:

```
♮^l let x = new^l in x.a := 42;
♮^l let y = new^l in y.a := "flush";
  x.a
```

Hence, the first rule that guides the preprocessing step is:

- If the right hand side of a let expression is a new expression, then the let expression is surrounded by a demotion.

  The parameter for the demotion is the abstract label of the new expression.

4.2.1.2 Functions

Consider Program 4.3, in which demotion expressions are inserted around each let, which has a new expression as its right hand side.

The first line creates a new, recent l-object. Two evaluation steps later x is substituted by the reference to the new object. Hence, the lambda expression in the second line contains a precise reference in its body. Some evaluation steps afterwards f is substituted by its closure, which contains the reference to the object that was created in the first line. The evaluation yields:

```
♮^l let y = new^l in
let z = (λ_.(@l,0).a)(0) in
  z
```

The most recent heap is $[(l, 0) \mapsto \{a : 42\}]$ while the summary heap is empty. However, before f is applied, line 4 creates another l-object. Hence, we need demotion as in the first example (Program 4.2) to ensure the invariant INV_{OH}.

The demotion expression adjusts all reference in its body. Hence, one step later the expression is: **let** y = new^l **in let** z = (λ_.(˜l,0).a)(0) **in** z. If we model the demotion expression is such a way, typing of functions becomes arduous. Depending on its use the function may access objects by exact references or by inexact references. To avoid this we treat substitution into a function body special. If a variable is substituted by a precise reference, the reference is converted to a summary reference. Hence, free variables inside of function bodies will never be the source of precise references.

If substitution demotes references inside of function bodies, the expression from Program 4.3 yields after three evaluation steps:

```
let f = λ_.(~l,0).a in
(@l,0).a := 42;
♮^l let y = new^l in
let z = f(0) in
  z
```

Please note that the body of the lambda expression contains an imprecise reference, while in the next line the reference to the object is still precise. The most recent heap after three evaluation steps is $[(l, 0) \mapsto \{\}]$. The example executes without getting stuck with the modified substitution. Later in the execution, if the function f is executed in line 5, the demote expression from line 4 has already moved the object from the most recent heap into the summary heap. Hence, the property read inside of the function body is successful.

Let us now assume that line 3 of Program 4.3 is omitted. The execution also puts the imprecise reference into the function body. But this time, because line 3 is omitted, no demote expression will move the object $(l, 0)$ into the summary heap, and hence the execution of f gets stuck. The body of f tries to access an imprecise object that is part of the most recent heap.

We can avoid the above problem by surrounding each function call with a demote expression that ensures that each object that is accessed by the function using a free variable is demoted before the function is called. The code of Program 4.3 then becomes:

```
1 ♮^l let x = new^l in
2 let f = λ_.x.a in
3 x.a := 42;
4 ♮^l let y = new^l in
5 ♮^l let z = f(0) in
6   z
```

In this example the references and heaps stay in synchrony – independent from the existence of line 4. Our static type system will ensure that the annotation that is put at the demotion in line 5 is correct, and that the demote expression appears in front of a function call. Type inference can compute the annotation set of the demote expression, and our preprocessing step adds a demote expression in front of each function call. So, no additional effort by the programmer is necessary.

Next, think of a function with multiple free variables, such that there is a need to demote a set of objects instead of one object. To demote more than one object we extend the syntax of demote $\natural^l e$ to $\natural^L e$ for a set of locations $L \subseteq \mathsf{Location}$.

The above example yields another rule for preprocessing:

- If the right hand side of a let expression is a function call, the let expression is surrounded by a demote expression.

  The parameter for the demote expression is computed by type inference. The inference ensures that all objects that may be passed to the function by a free variable are demoted before the function is called.

Hint The special substitution does not imply that functions cannot work with precise objects, since parameters can pass precise object references to functions. Section 4.2.3.2 presents a formal definition of the substitution.

4.2.2 Syntax

Figure 4.7 states the formal syntax of $\mathcal{JSR}$. A value is either `udf`, a reference $(q\ell, i)$ or a function $\mathtt{rec}^M f(y).e$ with a parameter y and a body e, where f is the name under which the programmer can access the function itself for recursion. Usually, we ignore the annotation M in the function expression because it is not important for the first presentation of the calculus.[1]

A reference consists of a qualifier q that identifies the heap in which the object is stored, an abstract location l and a natural number. The programmer is not allowed to write down references in programs.

A top expression e is either a value, a let expression $\mathtt{let}\ x = s\ \mathtt{in}\ e$ or a demote expression $\natural^L e$. $\mathtt{let}\ x = s\ \mathtt{in}\ e$ first evaluates the expression s, binds its result to the variable x and executes e with the new binding. The expression $\natural^L e$ demotes all objects with references $\ell \in L$; that means moving all objects with address $(@\ell, i)$, where $\ell \in L$, from the most recent heap into the summary heap, and adjusting the references from $(@\ell, i)$ to $(\tilde{}\ell, i)$.

A simple expression s is either a variable, a value, a function application $w(w)$, a new expression, a property read $w.a$ or a property write $w.a := w$.

The new expression is tagged by an abstract location. If a new object is created by a new expression the object is tagged by the corresponding abstract location of the new expression, and all references to this object that arise during program execution carry this abstract location with them.

Convention 4.2.3. *The sets Location, Variable, Prop and $\mathbb{N}$ are pairwise disjoint.*

4.2.3 Dynamic Semantics

The main difference between the semantics of $\mathcal{JSC}$ and $\mathcal{JSR}$ is the recency information. The other parts are similar.

[1] $\mathcal{JSR}$ uses the special substitution to eliminate precise references of free variables. The annotation stores the abstract location of all objects that have to be demoted before the function is called. Section 4.5.5 proves lemmata that utilizes the annotation.

$$
\begin{array}{rlll}
\textsf{Variable} \ni & x, y, f & ::= & x_1 \mid x_2 \mid \ldots \\
\textsf{Value} \ni & v & ::= & \texttt{rec}^{M} f(y).e \mid \texttt{udf} \mid (q\ell, i) \\
\textsf{TopExpr} \ni & e & ::= & w \mid \texttt{let}\ x = s\ \texttt{in}\ e \mid \natural^{L} e \\
\textsf{Va(r|l)} \ni & w & ::= & x \mid v \\
\textsf{SExpr} \ni & s & ::= & w \mid w(w) \mid \texttt{new}^{\ell} \mid w.a \mid w.a := w \\
\textsf{Qualifier} \ni & q & ::= & \tilde{} \mid @ \\
\textsf{Location} \ni & \ell & ::= & l_1 \mid l_2 \mid \ldots \\
\textsf{Location} \supseteq & L, A, M & & \text{(finite sets)} \\
\textsf{Prop} \ni & a & & \\
\textsf{Expr} \ni & d & ::= & s \mid e
\end{array}
$$

Figure 4.7: $\mathcal{JSR}$ – syntax. The differences between $\mathcal{JSC}$ and $\mathcal{JSR}$ are highlighted in gray. As in $\mathcal{JSC}$, the programmer is not allowed to write down references by himself. They only arise as intermediate values during evaluation. He also has to ensure $M = \emptyset$ for function definitions, because M is an annotation filled during evaluation.

Definition 4.2.4 (Configuration). *A configuration (H, H_0, d) contains a summary heap H, a most recent heap H_0, and an expression d. We also write $(\mathcal{H}, d)$ for the configuration (H, H_0, d) if $\mathcal{H} = (H, H_0)$.*

Figure 4.8 defines the small step operational semantics by induction. The relation $\longrightarrow$ takes a configuration and yields a new one. The rules S-App, S-Let and S-Let$'$ work like their counterparts in $\mathcal{JSC}$, but they rely on a modified substitution $d\{x \Mapsto v\}$ that takes care of demotion of references. Section 4.2.3.2 presents the modified substitution formally.

The demote expression $\natural^{L} e$ (cf. S-Dem) rely on another auxiliary operation, the demote operation $\cdot^{\natural L}$, which is presented in Section 4.2.3.1. The demote expression just applies the demote operation to the pair of heaps to move the appropriate objects into the summary heap and to adjust references inside the heaps. It is not necessary to modify qualifiers inside of the expression e, because substitution already adjusts the modifier of references inside of e. Therefore, the rule returns $e \searrow L$, which is the expression e itself, except annotations. The auxiliary function is defined in Figure 4.12.

The rule S-New creates a new object in the most recent heap and returns its reference $(@\ell, i)$. The side condition of the rule ensures that the most recent heap does not contain a ℓ-object, and that (ℓ, i) is a fresh memory location. Hence, the side condition ensures that INV_{OH} and INV_{DH} stay intact during evaluation.

The rule S-Rd and S-Wrt facilitates map lookup and map update, which are defined in Figure 4.9 to perform property lookup or property update on objects. The map lookup and map update auxiliary functions for references with qualifiers decide depending on the qualifier with which heap they work.

$$\begin{array}{lcll}
H & \in & \mathsf{Heap} & = \mathsf{Location} \times \mathbb{N} \xrightarrow{fin} \mathsf{Object} \\
\mathcal{H} & \in & \mathsf{Heap} \times \mathsf{Heap} & \\
h & \in & \mathsf{Object} & = \mathsf{Prop} \xrightarrow{fin} \mathsf{Value} \\
\mathcal{E} & ::= & \multicolumn{2}{l}{\Box \mid \mathtt{let}\ x = s\ \mathtt{in}\ \mathcal{E} \mid \natural^{L}\mathcal{E}}
\end{array}$$

$$\begin{array}{lll}
\textsc{S-Dem} & \mathcal{H}, \natural^{L} e & \longrightarrow \mathcal{H}^{\natural L}, e \searrow L \\
\textsc{S-App} & \mathcal{H}, (\mathtt{rec}\, f(x).e)(v) & \longrightarrow \mathcal{H}, e\{f, x \Mapsto \mathtt{rec}\, f(x).e, v\} \\
\textsc{S-Let} & \mathcal{H}, \mathtt{let}\ x = v\ \mathtt{in}\ e & \longrightarrow \mathcal{H}, e\{x \Mapsto v\} \\
\textsc{S-New} & H, H_0, \mathtt{new}^{\ell} & \longrightarrow H, H_0\{(\ell, i) \mapsto \{\}\}, (@\ell, i) \\
 & & \quad \text{if } \mathrm{dom}(H_0) \cap (\{\ell\} \times \mathbb{N}) = \emptyset \\
 & & \quad \quad \text{and } (\ell, i) \notin \mathrm{dom}(H) \\
\textsc{S-Rd} & \mathcal{H}, (q\ell, i).a & \longrightarrow \mathcal{H}, \mathcal{H}(q\ell, i)\$a \\
\textsc{S-Wrt} & \mathcal{H}, (q\ell, i).a := v & \longrightarrow \mathcal{H}\{(q\ell, i)(a) \mapsto v\}, \mathtt{udf}
\end{array}$$

$$\textsc{S-Let}' \quad \frac{\mathcal{H}, s \longrightarrow \mathcal{H}', \mathcal{E}[w]}{\mathcal{H}, \mathtt{let}\ x = s\ \mathtt{in}\ e'' \longrightarrow \mathcal{H}', \mathcal{E}[\mathtt{let}\ x = w\ \mathtt{in}\ e'']}$$

Figure 4.8: $\mathcal{JSR}$ – small-step operational semantics. Inductive. The rule S-Dem uses the function $e \searrow L$, which is defined in Figure 4.12. The function adjusts the annotations of lambda expressions.

4.2.3.1 Demotion of References

The semantics of $\mathcal{JSR}$ uses the operation $\cdot^{\natural L}$ for expressions, top level expressions, values and heaps. It adjusts the qualifier of a reference $(q\ell, i)$ if $q = @$ and $\ell \in L$.

Figure 4.10 defines demotion for variables and values, Figure 4.11 presents the definition for expressions and top level expressions. Demotion for values adjusts the qualifier of a reference from @ to ˜ if the abstract location of the reference is element of the set L that is the parameter of the demote operation. Demotion does not modify the body of a function, hence, $(\mathtt{rec}\, f(x).e)^{\natural L}$ is defined as $\mathtt{rec}\, f(x).e$, because the modified substitution adjusts references inside of a lambda expression (cf. Section 4.2.3.2).

For expressions demotion basically demotes all its values. One exception for this rule is the demote expression itself. The demote expression $\natural^{L'}e$ takes care of all references $l \in L'$, hence the demotion only continues with all $l \in L - L'$.

Demotion for heaps and property maps is defined pointwise.

$$\begin{array}{lcll}
H^{\natural L}(\ell, i) & := & H(\ell, i)^{\natural L} & \forall (\ell, i) \in \mathrm{dom}(H) \\
(h^{\natural L})\$a & := & (h\$a)^{\natural L} & \forall a \in \mathrm{dom}(h)
\end{array}$$

Finally, the semantics applies demotion to a pair of heaps by first moving all L-objects from the singleton heap H_0 to the summary heap and then applying heap

$$(H, H_0)(q\ell, i) := \begin{cases} H_0(\ell, i) & \text{if } q = @ \\ H(\ell, i) & \text{if } q = \tilde{} \end{cases}$$

$$(H, H_0)\{(q\ell, i) \mapsto p\} := \begin{cases} H, H_0\{(\ell, i) \mapsto p\} & \text{if } q = @ \\ H\{(\ell, i) \mapsto p\}, H_0 & \text{if } q = \tilde{} \end{cases}$$

Figure 4.9: $\mathcal{JSR}$ – map lookup and map update for references. The figure defines map lookup for a reference $(q\ell, i)$ and a pair of heaps H, H_0 and map update for a reference $(q\ell, i)$, a pair of heaps H, H_0 and a property map p. These definitions facilitate the notation of map lookup and map update (cf. Section 1.3) $(H, H_0)(q, \ell i)$

$$\begin{aligned} x^{\natural L} &:= x \\ (\mathtt{rec}\, f(x).e)^{\natural L} &:= \mathtt{rec}\, f(x).e \\ \mathtt{udf}^{\natural L} &:= \mathtt{udf} \\ (q\ell, i)^{\natural L} &:= \begin{cases} (\tilde{}\ell, i) & \text{if } \ell \in L \\ (q\ell, i) & \text{if } \ell \notin L \end{cases} \end{aligned}$$

Figure 4.10: $\mathcal{JSR}$ – demotion for variables and values.

demotion to both parts individually:

$$\begin{aligned} (H, H_0)^{\natural L} \quad = \quad & (H \cup H_L)^{\natural L}, (H_0 \backslash H_L)^{\natural L} \\ & \text{where } H_L = H_0 \downarrow \{(\ell, i) \mid \ell \in L, i \in \mathbb{N}\} \end{aligned}$$

Demotion for a type environment $\Gamma^{\natural L}$ is defined pointwise.

$$(\Gamma^{\natural L})(x) = (\Gamma(x))^{\natural L} \quad \forall x \in \mathrm{dom}(\Gamma)$$

We also define the shortcut $\cdot^{\natural} := \cdot^{\natural \mathsf{Location}}$ such that if no location is given every abstract location of the program is demoted.

Figure 4.12 defines the function $e \searrow L$. It adjusts the annotation of rec expressions. The purpose of the annotation from the rec expression is to keep track of the set of abstract locations that need to be demoted before the function is called. Every time we execute a demote operation, we adjust these set accordingly.

4.2.3.2 Substitution

Figure 4.13 defines the modified capture avoiding substitution $d\{x \Mapsto v\}$ for $\mathcal{JSR}$. If demotion is the identity, the modified substitution is similar to a standard capture avoiding substitution. The difference is that the value is demoted by the modified version if it is substituted into a demote expression or a rec expression

Top level expressions: $\cdot^{\natural L} : \mathsf{TopExpr} \to \mathsf{TopExpr}$

$$\begin{aligned} v^{\natural L} &:= v^{\natural L} \\ (\texttt{let}\ x = s\ \texttt{in}\ e)^{\natural L} &:= \texttt{let}\ x = s^{\natural L}\ \texttt{in}\ e^{\natural L} \\ (\natural^{L'} e)^{\natural L} &:= \natural^{L'} (e^{\natural (L - L')}) \end{aligned}$$

Expressions: $\cdot^{\natural L} : \mathsf{SExpr} \to \mathsf{SExpr}$

$$\begin{aligned} w^{\natural L} &:= w^{\natural L} \\ (w_1(w_2))^{\natural L} &:= w_1^{\natural L}(w_2^{\natural L}) \\ (\texttt{new}^l)^{\natural L} &:= \texttt{new}^l \\ (w.a)^{\natural L} &:= w^{\natural L}.a \\ (w_1.a := w_2)^{\natural L} &:= w_1^{\natural L}.a := w_2^{\natural L} \end{aligned}$$

Figure 4.11: $\mathcal{JSR}$ – demotion for expressions and top level expressions.

If $e = \big(\texttt{let}\ x = (\mathbf{rec}^M f(x).e_f)(w)\ \texttt{in}\ e_b\big)$, then

$$e \searrow L := \texttt{let}\ x = (\mathbf{rec}^{M \setminus L} f(x).e_f)(w)\ \texttt{in}\ e_b$$

otherwise

$$e \searrow L := e$$

Figure 4.12: $\mathcal{JSR}$ – remove demote annotations.

(cf. the last three cases). If the expression is a demotion, then $(\natural^L e)\{x \Mapsto v\}$ demotes references inside of v with respect to L: $\natural^L(e\{x \Mapsto v^{\natural L}\})$. It ensures that no precise references is able to break the demotion.

Section 4.2.1.2 explains with an example why it is necessary to prevent substitution of free variables by precise references. Substitution inside a function body demotes the value v with respect to $\mathsf{Location}$. As a consequence, no precise reference arises from the function body by free variables. The side condition $M' = M \cup @\mathrm{Locs}(v)$ is a technical detail that makes it simpler to prove the substitution lemma by keeping track of possible substitutions.

$$
\begin{array}{rcll}
y\{x \mapsto v\} & = & y & \\
x\{x \mapsto v\} & = & v & \\
\mathtt{udf}\{x \mapsto v\} & = & \mathtt{udf} & \\
(q\ell, i)\{x \mapsto v\} & = & (q\ell, i) & \\
(w_1(w_2))\{x \mapsto v\} & = & w_1\{x \mapsto v\}(w_2\{x \mapsto v\}) & \\
\mathtt{new}^{\ell}\{x \mapsto v\} & = & \mathtt{new}^{\ell} & \\
(w_1.a)\{x \mapsto v\} & = & w_1\{x \mapsto v\}.a & \\
(w_1.a := w_2)\{x \mapsto v\} & = & w_1\{x \mapsto v\}.a := (w_2\{x \mapsto v\}) & \\
(\mathtt{let}\ x = s\ \mathtt{in}\ e)\{x \mapsto v\} & = & \mathtt{let}\ y = s\{x \mapsto v\}\ \mathtt{in}\ e & \\
(\mathtt{let}\ y = s\ \mathtt{in}\ e)\{x \mapsto v\} & = & \mathtt{let}\ y = s\{x \mapsto v\}\ \mathtt{in}\ (e\{x \mapsto v\}) & \text{if } y \notin \mathrm{fv}(v) \\
(\natural^{L} e)\{x \mapsto v\} & = & \natural^{L}(e\{x \mapsto v^{\natural L}\}) & \\
(\mathbf{rec}^{M} f(z).e)\{x \mapsto v\} & = & \mathbf{rec}^{M} f(z).e & \text{if } x \in \{z, f\} \\
(\mathbf{rec}^{M} f(z).e)\{x \mapsto v\} & = & \mathbf{rec}^{M'} f(z).(e\{x \mapsto v^{\natural}\}) & \text{if } x \notin \{z, f\} \\
 & & & \text{and } \{z, f\} \cap \mathrm{fv}(v) = \emptyset \\
 & & & \text{for } M' = M \cup \mathrm{@Locs}(v) \\
d\{x, y \mapsto v_x, v_y\} & = & (d\{x \mapsto v_x\})\{y \mapsto v_y\} & \text{if } y \notin \mathrm{fv}(v_x)
\end{array}
$$

Figure 4.13: $\mathcal{JSR}$ – modified substitution. The figure defines the capture avoiding substitution inductively. The definition assumes that $x \neq y$.

4.2.4 Invariants of $\mathcal{JSR}$

Lemma 4.2.5 (Context for top expression)**.** *For all top expressions e, there exists one evaluation context $\mathcal{E}$ and a variable or value w, such that $\mathcal{E}[w] = e$.*

Proof by contradiction. Assume there exists at least one top expressions e such that there exits no $\mathcal{E}[w]$ with $\mathcal{E}[w] = e$. Let e be the expression from this set with the smallest height. This expression exists, if the set is not empty, since our expressions all have a finite height.

Either e is a let expression, a demote expression, or a variable, or a value. The last two cases are not possible, since for $\mathcal{E} = \square$ holds $\mathcal{E}[w] = e$.

Use the same ideas as in Lemma 4.1.4 to prove that the case where e is a let expression is not possible. The same argument is applicable for demote expressions. □

Lemma 4.2.6 (Context existence)**.** *For heaps $\mathcal{H}, \mathcal{H}'$, a simple expression s and a top expression e, if $\mathcal{H}, s \longrightarrow \mathcal{H}', e$, then there exists one evaluation context $\mathcal{E}$ and a value or variable w, such that $\mathcal{E}[w] = e$.*

Proof. Analogous to Lemma 4.1.5. The only difference is that we apply Lemma 4.2.5 instead of Lemma 4.1.4. □

Definition 4.2.7 (Closedness)**.** *A value v (or expression d) is* closed*, if $fv(v) = \emptyset$ ($fv(d) = \emptyset$).*

Lemma 4.2.8. *For an expression d it holds that for all variables x*

$$fv(d\{x \mapsto v\}) \subseteq (fv(d) - \{x\}) \cup fv(v)$$

Proof. Analogous to Lemma 4.1.3 by induction over the definition of the substitution function. □

Lemma 4.2.9. *For an expression d with $fv(d) = \{x\}$ and a closed v, $d\{x \mapsto v\}$ is closed.*

For an expression d with $fv(d) = \{x, f\}$ and closed v and d', $d\{x, f \mapsto v, d'\}$ is closed.

Proof. Direct consequence of Lemma 4.2.8. □

Lemma 4.2.10 (Closedness)**.** *For a configuration C with a closed expression d, if $C \longrightarrow C'$, then d' is closed.*

Proof by induction over $\longrightarrow$. We make a case distinction and apply Lemma 4.2.9. □

Next, we restate here the two invariants $\mathrm{INV}_{\mathrm{OH}}$ and $\mathrm{INV}_{\mathrm{DH}}$ formally and prove some lemmata, basically stating that they are valid for each program we are interested in.

Definition 4.2.11 ($\mathrm{INV}_{\mathrm{OH}}$)**.** *For $\mathcal{H} = (H, H_0)$, $\mathrm{INV}_{\mathrm{OH}}(\mathcal{H})$ holds, if $\forall \ell \in dom(H_0)_{\downarrow 1} : \exists^{\leq 1} i : (\ell, i) \in dom(H_0)$.*

$\mathrm{INV}_{\mathrm{OH}}(C)$ *holds for a configuration $C = (H, H_0, d)$, if* $\mathrm{INV}_{\mathrm{OH}}(H, H_0)$ *holds.*

Lemma 4.2.12. $\mathrm{INV}_{\mathrm{OH}}(\emptyset, \emptyset, d)$ *holds.*

Proof. Trivial since the most recent heap is empty. □

Lemma 4.2.13 ($\mathrm{INV}_{\mathrm{OH}}$)**.** *If* $\mathrm{INV}_{\mathrm{OH}}(C)$*, and $C \longrightarrow C'$, then* $\mathrm{INV}_{\mathrm{OH}}(C')$*.*

Proof by induction over $\longrightarrow$. The only two relevant rules are S-Dem and S-New.

- *Case* S-Dem: By the definition of $\mathcal{H}^{\natural L}$, from $\mathrm{INV}_{\mathrm{OH}}(\mathcal{H})$ follows that $\mathrm{INV}_{\mathrm{OH}}(\mathcal{H}^{\natural L})$ holds. Therefore, the case S-Dem is proved.
- *Case* S-New: The object creation case is trivial due to the conditions of the rule S-New.

□

Definition 4.2.14 ($\longrightarrow^*$)**.** *As usual let $\longrightarrow^*$ be the transitive reflexive closure of $\longrightarrow$.*

Corollary 4.2.15 ($\mathrm{INV}_{\mathrm{OH}}$)**.** *For all C with $(\emptyset, \emptyset, d) \longrightarrow^* C$ it holds* $\mathrm{INV}_{\mathrm{OH}}(C)$*.*

Proof by induction over the length of the derivation. In the different cases apply Lemma 4.2.12 and Lemma 4.2.13. □

Definition 4.2.16 ($\mathrm{INV_{DH}}$). $\mathrm{INV_{DH}}(C)$ *for* $C = (H, H_0, d)$ *holds, if* $dom(H) \cap dom(H_0) = \emptyset$.

Lemma 4.2.17. $\mathrm{INV_{DH}}(\emptyset, \emptyset, d)$ *holds.*

Proof. Trivial, because the two heaps have an empty domain. □

Lemma 4.2.18. *If* $\mathrm{INV_{DH}}(C)$*, and* $C \longrightarrow C'$*, then* $\mathrm{INV_{DH}}(C')$*.*

Proof by induction over $\longrightarrow$. Only the rule S-Dem and S-New changes the domains of the heaps. Hence, for all other rules the claim is immediate or holds by induction.

- *Case* S-Dem: The definition of demotion for a pair of heaps just moves objects from the most recent heap into the summary heap (beside adjusting references).
- *Case* S-New: The side condition of the new rule ensures that the reference of the newly created object is not already part of the domain of the summary heap.

□

Corollary 4.2.19 ($\mathrm{INV_{DH}}$). *For all* C *with* $(\emptyset, \emptyset, d) \longrightarrow^* C$ *it holds* $\mathrm{INV_{DH}}(C)$*.*

Proof. Lemma 4.2.17, Lemma 4.2.18 □

Definition 4.2.20 ($\mathrm{INV_H}$). $\mathrm{INV_H}(C)$ *holds, if* $\mathrm{INV_{OH}}(C)$ *and* $\mathrm{INV_{DH}}(C)$*.*

Corollary 4.2.21 ($\mathrm{INV_H}$). *For all* C *with* $(\emptyset, \emptyset, d) \longrightarrow^* C$ *it holds* $\mathrm{INV_H}(C)$*.*

Proof. Corollary 4.2.15, Corollary 4.2.19 □

4.3 Connection between $\mathcal{JSC}$ and $\mathcal{JSR}$

This section presents a proof that the only difference between $\mathcal{JSC}$ and $\mathcal{JSR}$ lies in the recency information. This statement is formulated for expressions and heaps modulo memory allocation (cf. Definition 4.1.7). We omit the corresponding definition for expressions and heaps in $\mathcal{JSR}$ because it is analogous to the definition presented in Figure 4.6.

To make it easier to distinguish between expressions from $\mathcal{JSC}$ and $\mathcal{JSR}$ we use in this section the metavariable f to indicate expression are from $\mathcal{JSR}$, while e indicates that we are dealing with expressions from $\mathcal{JSC}$.

In this section we prove a simulation result, starting from an expression $f \in \mathcal{JSR}$, and show that for an equivalent expression $e \in \mathcal{JSC}$, e and f evaluate in a similar manner. Roughly spoken (for example omitting the heaps), we state that for expressions $e \in \mathcal{JSC}$ and $f \in \mathcal{JSR}$ with $e \sim f$ (e is equivalent to f), if $f \longrightarrow^* f'$, there exists an e', such that $e \to_0^* e'$ and $e' \sim f'$. The relation $\sim$ relates expressions from $\mathcal{JSC}$ to those from $\mathcal{JSR}$, such that their structure is similar. Hence, the theorem allows to transform the safety properties from $\mathcal{JSR}$ to $\mathcal{JSC}$. The result is formally stated in the Theorem 4.3.5.

Definition 4.3.1 ($\mathcal{JSC}$-$\mathcal{JSR}$ Equivalence)**.** *An expression $e \in \mathcal{JSC}$ is* equivalent *to an expression $f \in \mathcal{JSR}$, if $e \sim f$ holds. It is defined as:*

$$x \sim x \qquad \texttt{udf} \sim \texttt{udf} \qquad \vec{i} \sim (q\ell, i) \qquad \frac{e \sim f}{\texttt{rec}\, g(x).e \sim \texttt{rec}^M g(x).f}$$

$$\frac{s \sim s' \qquad e \sim f}{\texttt{let}\ x = s\ \texttt{in}\ e \sim \texttt{let}\ x = s'\ \texttt{in}\ f} \qquad \frac{e \sim f}{e \sim \natural^L f} \qquad \frac{v_1 \sim v_1' \qquad v_2 \sim v_2'}{v_1(v_2) \sim v_1'(v_2')}$$

$$\frac{v \sim v'}{v.a \sim v'.a} \qquad \frac{v_1 \sim v_1' \qquad v_2 \sim v_2'}{v_1.a := v_2 \sim v_1'.a := v_2'}$$

A $\mathcal{JSC}$ heap and a tuple of $\mathcal{JSR}$ heaps are equivalent, iff

$$\frac{\begin{array}{c} dom(H) = (dom(H') \cup dom(H_0'))_{\downarrow 2} \\ dom(H') \cap dom(H_0') = \emptyset \qquad \forall i \in dom(H) : \exists^{=1}\ell : H(i) \sim (H', H_0')(\ell, i) \end{array}}{H \sim H', H_0'}$$

Lemma 4.3.2. *For all $e \in \mathcal{JSC}$ and $f \in \mathcal{JSR}$ with $e \sim f$, it holds*

$$e \sim f^{\natural L} \tag{4.1}$$

for all L.

Proof by induction over the structure of e.

- *Case* $e = x$, $e = \texttt{udf}$, $e = \vec{i}$: trivial.
- *Case* else: Immediate by induction.

□

Lemma 4.3.3 ($\mathcal{JSC}$-$\mathcal{JSR}$ substitution)**.** *For all $e, v \in \mathcal{JSC}$, $f, v' \in \mathcal{JSR}$ and for all L, with $e \sim f$ and $v \sim v'$ it holds:*

$$e[x \mapsto v] \sim f\{x \Mapsto v'^{\natural L}\}$$

Proof by induction over the shape of f.

- *Case* $f = \texttt{udf}$, $f = (q\ell, i)$, $f = \texttt{new}^\ell$: The substitution does not affect f, such that $f\{x \Mapsto v'\} = f$. Inversion of $\sim$ yields also $e[x \mapsto v] = e$.
- *Case* $f = y$: If $y \neq x$, then substitution does not change f. Let us assume $y = x$, then $f\{x \Mapsto v'^{\natural L}\} = v'^{\natural L}$. Inversion of the definition of $\sim$ yields for e: $e = y$. Hence, $e[x \mapsto v] = v$. Because of $v \sim v'$, it holds for all L: $v \sim v'^{\natural L}$ and $e[x \mapsto v] \sim f\{x \Mapsto v'^{\natural L}\}$.

- *Case* $f = \texttt{let}\ y = s'\ \texttt{in}\ f_b$: Inversion of $\sim$ yields for e: $e = \texttt{let}\ y = s\ \texttt{in}\ e_b$. There are two cases, either $x = y$, or $x \neq y$.
  - *Case* $x \neq y$: Then

    $$f\{x \Mapsto v'^{\natural L}\} = \texttt{let}\ y = s\{x \Mapsto v'^{\natural L}\}\ \texttt{in}\ f_b\{x \Mapsto v'^{\natural L}\} \tag{4.2}$$

    For e it holds:

    $$e[x \mapsto v] = \texttt{let}\ y = s[x \mapsto v]\ \texttt{in}\ e_b[x \mapsto v] \tag{4.3}$$

    By induction it holds for all L'':

    $$s[x \mapsto v] \sim s\{x \Mapsto v'^{\natural L''}\} \tag{4.4}$$

    $$e_b[x \mapsto v] \sim f_b\{x \Mapsto v'^{\natural L''}\} \tag{4.5}$$

    Hence, the definition of $\sim$ for let expressions yields the desired result.
  - *Case* $x = y$: The case $x = y$ is analogues, but f_b and e_b are not affected by the substitution.
- *Case* $f = \natural^L f_b$:

  Inversion of $\sim$ yields $e = \natural^L e_b$. Induction yields $e_b[x \mapsto v] \sim f_b\{x \Mapsto v'^{\natural L'}\}$ for all L''.

  This implies that $e_b[x \mapsto v] \sim f_b\{x \Mapsto v'^{\natural L' \cup L}\}$ for all L'. We can always choose an L'', such that $L'' = L' \cup L$.

  Hence, $e[x \mapsto v] \sim (\natural^L f_b)\{x \Mapsto v'^{\natural L'}\} = f\{x \Mapsto v'^{\natural L}\}$ for all L'.
- *Case* $f = (v_1)v_2$, $f = v.a$, $f = v_1.a := v_2$: These cases are straightforward by induction on v or v_1 and v_2.

□

Lemma 4.3.4. *For all heaps $H \in \mathcal{JSC}$ and $\mathcal{H} \in \mathcal{JSR}$ with $H \sim \mathcal{H}$ it holds*

$$H \sim \mathcal{H}^{\natural L}$$

for all L.

Proof. Immediate by Lemma 4.3.3. □

Theorem 4.3.5 ($\mathcal{JSR}$ to $\mathcal{JSC}$ simulation)**.** *For all $e \in \mathcal{JSC}$ and $f, f' \in \mathcal{JSR}$, with $e \sim f$, $H \sim \mathcal{H}$ and $\mathcal{H}, f \longrightarrow^* \mathcal{H}', f'$ there exists e' and H', such that*

$$H, e \rightarrow_0^* H', e' \qquad \textit{and} \qquad e' \sim f' \qquad \textit{and} \qquad H' \sim \mathcal{H}'$$

Proof by induction of $\longrightarrow^*$.

- *Case* S-Dem: Immediate because $\longrightarrow^*$ is reflexive and due to Lemma 4.3.4.

$$
\begin{array}{rlll}
\mathsf{Type}_c \ni & t & ::= \mathtt{obj}(p) \mid (\Delta, t) \xrightarrow{L,A} (\Delta, t) \mid \top \mid \mathtt{udf} & \\
\mathsf{Reference} \ni & p & ::= \tilde{}L \mid @\{\ell\} & \text{with } |L| \geq 1, L \text{ finite} \\
\mathsf{HeapType} \ni & r & ::= \emptyset \mid r[a : t] & \\
\mathsf{SummaryEnv} \ni & \Sigma & ::= \emptyset \mid \Sigma(\ell : r) & \\
\mathsf{SingletonEnv} \ni & \Delta & ::= \emptyset \mid \Delta(\ell : r) & \\
\mathsf{TypeEnv} \ni & \Gamma & ::= \emptyset \mid \Gamma(x : t) &
\end{array}
$$

Figure 4.14: $\mathcal{JSR}$ – type syntax. Co-inductive.

- *Case* S-App, S-Let: By induction and Lemma 4.3.3.
- *Case* S-New, S-Rd, S-Wrt: Immediate.
- *Case* S-Let′: By induction.

□

4.4 Static Semantics of $\mathcal{JSR}$

In this section we present the formal syntax of our static type system in Section 4.4.1. Object types and function types are special in $\mathcal{JSR}$ compared to other type systems [100]. Thus we show how they behave with some examples. We continue by presenting the subtyping relation of the calculus in Section 4.4.2. Another important relation of $\mathcal{JSR}$ is the flow relation (Section 4.4.3), which models the movement between the most recent heap and the summary heap. It ensures that the static semantics stays synchronized with its dynamic counterpart. Because the dynamic semantic demotes references, we define an analogous operation on types in Section 4.4.4. After introducing these relations and functions on types, we present the type rules for values except functions (Section 4.4.5), expressions and top expressions (Section 4.4.6). Because the typing rule for functions is complicated, we present it in Section 4.4.7. The typing rules utilize additional relations, which we introduce afterwards in Section 4.4.8.

4.4.1 Syntax

In Figure 4.14 we define the syntax of types co-inductively. A type is either an object type $\mathtt{obj}(p)$, a function type $(\Delta, t) \xrightarrow{L,A} (\Delta, t)$, the top type $\top$ or the singleton type $\mathtt{udf}$.

In the object type, p refers to a set of heap types through an environment, either using the summary environment (Σ) if $p = \tilde{}L$ or a singleton environment (Δ) if $p = @\{\ell\}$. Hence, the object type itself does not provide complete information about the shape of the object. For this purpose, a heap type r is used additionally. A heap type is a finite mapping from properties to types describing the shape of an object. Section 4.4.1.1 explains object types in more detail. To make the environments more readable we omit the $\emptyset$ entry at the beginning if there are bindings for an environment.

A function type $(\Delta_1, t_1) \xrightarrow{L,A} (\Delta_2, t_2)$ contains a lot of information. Of course it has to state what is the type of the parameter (t_1) and return value (t_2). It also specifies the structure of the most recent heap before invocation (Δ_1), and how the structure will look after execution (Δ_2). The two sets over the arrow of the function type L and A are effects of the function. The first effect collects all locations, for which the function may create new objects, and the second one collects all locations which the function may access (read or write). Section 4.4.1.2 contains a detailed description and motivation for these choices.

Our static type system has a flow-sensitive, and a flow-insensitive part. We treat most recent objects flow-sensitively, while we treat old objects flow-insensitively. The nature of the treatment of most recent objects and summary objects impairs the treatment of the most recent environment and the summary environment.

Our flow sensitive type system uses the judgment

$$\Sigma, \Delta, \Gamma \vdash_{te} e : t \Rightarrow L, A, \Delta, \Gamma$$

to type top expressions, the judgment

$$\Sigma, \Delta, \Gamma \vdash_{s} s : t \Rightarrow L, A, \Delta$$

to type expressions and the judgment

$$\Sigma, \Gamma \vdash_{w} w : t$$

to type values and variables. Each of the three judgments relate their syntactic entity (top expression, expression, value or variable) to a suitable type.

The type judgment for top expressions depends on three environments, the summary environment Σ, the singleton environment Δ and the type environment Γ. Because we treat most recent objects flow-sensitively, the judgment returns a new singleton environment. The type environment is treated flow-sensitively, too, because a demote expression may change the type of variables. The summary environment is not treated flow-sensitively and therefore it is not returned by the judgment. The judgment also computes two effects. L is the allocation effect and A is the access effect. We explain the effects in more detail in Section 4.4.1.2.

Because types of variables only change in our calculus due to the demote expression, which is a top expression, the type judgment for expressions only returns a modified singleton environment along with the two effects.

Variables and values do not change the state of objects or the type of variables. Hence, the judgments for types and variables do not return any environment. It just relates the variable or value to a suitable type.

To establish some basic properties, such that for an abstract location only one binding exists in an environment, well-formedness for types and environment is defined in Figure 4.15. From now on, if we are talking about types and environments, we always assume implicitly that they are well-formedness. Additionally,

$$\textsc{WfTop}\quad \vdash_{\mathfrak{wf}} \top \qquad \textsc{WfUndef}\quad \vdash_{\mathfrak{wf}} \mathtt{udf} \qquad \textsc{WfObj}\quad \vdash_{\mathfrak{wf}} \mathtt{obj}(p)$$

$$\textsc{WfFunction}\quad \frac{\vdash_{\mathfrak{wf}} \Delta_1 \qquad \vdash_{\mathfrak{wf}} \Delta_2 \qquad \vdash_{\mathfrak{wf}} t_1 \qquad \vdash_{\mathfrak{wf}} t_2}{\vdash_{\mathfrak{wf}} (\Delta_1, t_1) \xrightarrow{L,A} (\Delta_2, t_2)}$$

$$\textsc{WfEmptySet}\quad \vdash_{\mathfrak{wf}} \emptyset$$

$$\textsc{WfObject}\quad \frac{\vdash_{\mathfrak{wf}} r \qquad \vdash_{\mathfrak{wf}} t \qquad a \notin \mathrm{dom}(r) \qquad \mathrm{dom}(r)\ \text{finite}}{\vdash_{\mathfrak{wf}} r[a : t]}$$

$$\textsc{WfMostRecentHeap}\quad \frac{\vdash_{\mathfrak{wf}} \Delta \qquad \vdash_{\mathfrak{wf}} r \qquad \ell \notin \mathrm{dom}(\Delta) \qquad \mathrm{dom}(\Sigma)\ \text{finite}}{\vdash_{\mathfrak{wf}} \Delta(\ell : r)}$$

$$\textsc{WfSummaryHeap}\quad \frac{\vdash_{\mathfrak{wf}} \Sigma \qquad \vdash_{\mathfrak{wf}} r \qquad \ell \notin \mathrm{dom}(\Sigma) \qquad \mathrm{dom}(\Delta)\ \text{finite}}{\vdash_{\mathfrak{wf}} \Sigma(\ell : r)}$$

$$\textsc{WfTypeEnvironment}\quad \frac{\vdash_{\mathfrak{wf}} \Gamma \qquad \vdash_{\mathfrak{wf}} t \qquad x \notin \mathrm{dom}(\Gamma) \qquad \mathrm{dom}(\Gamma)\ \text{finite}}{\vdash_{\mathfrak{wf}} \Gamma(x : t)}$$

Figure 4.15: $\mathcal{JSR}$ – well-formedness of types and environments. We interpret the rules for the well-formedness relation co-inductively. Otherwise the relation filters all infinite types out. The relation restricts environments to finite maps that contains at most one binding per key.

we restrict ourself to regular types, which is a usual practice when working with co-inductively defined mathematical entities [100]. Please note that well-formedness environments have a finite number of entries.

Convention 4.4.1. *From now on we assume that all syntactic entities from Figure 4.14 are well-formed (cf. Figure 4.15).*

4.4.1.1 Object Types

To get familiar with the type system and the type syntax of $\mathcal{JSR}$ we take a look at some simple examples that use objects. Please first consider Program 4.4. In the

Program 4.4 $\mathcal{JSC}$ – object initialization.

```
let x = new^l in
  x.a := 42;
  x.name := "Meyer";
  x.a
```

Program 4.5 $\mathcal{JSC}$ – alias.

```
1 let x = new^l in
2 let y = x in
3   y.a := 42;
4   x.a
```

first line a new object is created. The variable x holds a reference to this object. In the next two lines two new properties are added to the object. The property name is set to the value "Meyer" while a is set to the float value 42. The last line of the example returns the float value 42 as the value of the whole expression.

The type of the variable x is in all lines of the program fragment obj$(@\ell)$. As the type judgment for expressions treat the most recent environment flow-sensitive, the environment reflects the state of the most recent heap exactly in this example. Before the first line is executed the most recent heap environment is empty. The judgment that types the new expression returns the most recent environment $(\ell : \emptyset)$. The assignment in line 2 then takes this most recent environment and adjusts it, as the semantics adjusts the heap content and returns $(\ell : [\texttt{a} : \texttt{float}])$.[2] Next, it adjusts the shape another time to $(\ell : [\texttt{a} : \texttt{float}][\texttt{name} : \texttt{string}])$. Therefore, the flow-sensitive part of the type system is capable of returning the type float for the expression x.a.

One interesting question arises when a program introduces aliases. Consider Program 4.5. In the first line a new empty object is created. In the next line an alias is created (y). For a type system it is hard to deal with aliases, and the question is, what happens in the next two lines. In line 3 the object gets a new property a with the value 42. The last line reads the property a of object x. Because x and y are aliased the property read returns 42.

The type system relates the variable x to the type obj$(@\ell)$. The new expression induces – as in the first example – an entry into the most recent heap for the abstract location ℓ. The most recent environment is $(\ell : \emptyset)$ at beginning of line 2. Hence, the variable y gets the type obj$(@\ell)$. In line 3 the type judgment adjusts the most recent environment to $(\ell : [\texttt{a} : \texttt{float}])$. Because of the indirection for object types, the type system is aware of the aliases x and y and the property read in line 4 returns float.

4.4.1.2 Function Types

As already explained in Section 4.2.1.2 at Program 4.3, functions, in some way, may enforce the caller to clean the most recent heap, before the function is invoked. The reason why this is necessary is that it is hairy to allow functions working with most recent objects using free variables.

[2] We assume here the presens of a base type float and string for the example.

Program 4.6 $\mathcal{JSC}$ – calling function with allocation twice.

```
1   let f = λ().
2     $\natural^{l_1}$ let x = $\texttt{new}^{l_1}$ in
3       x
4   in
5   let x1 = f(udf) in
6     // do something with x1, that changes the shape of x1
7   let x2 = f(udf) in
8     // do something with x2, that changes the shape of x2
9   let x3 = f(udf) in
10    ...
```

Additionally, a function may enforce the caller to free the heap before function invocation appropriate. Therefore, the type of a function heavily depends on the structure of the heap. It is obvious that a function needs the structure of all objects that the body may access in some sense (e.g. read, write or allocate). Consider Program 4.6. Even if the function f at first demotes the l_1 object in order to make space for the new l_1 object that is created by the new expression afterward, the function depends on the shape of the l_1 object stored in the most recent heap before invocation. In our example the function type may be

$$(\Delta, \top) \xrightarrow{L,A} ((l_1 : \emptyset), \texttt{obj}(@l_1)) \tag{4.6}$$

for some most recent heap environment Δ and effect L and A. We will first ignore the access effect A. It is explained in Section 4.4.1.3.

The problem with the example is now that the invocation of f in line 9 is not typeable, since the most recent heap contains an l_1 object with another shape as the l_1 object in the most recent heap in line 7.[3]

A solution for this problem is that the function enforces the caller to clean the most recent heap with respect to l_1, before the invocation happens. Then, the function itself does not enforce any restrictions on the structure of the most recent heap for l_1 objects, except that there must be a free place for a new l_1 object the function creates. Therefore, the allocation effect is used to express that a function requires a clean heap for all abstract locations that are part of the effect. Hence, the type of the function is:

$$(\emptyset, \top) \xrightarrow{\{l_1\},A} ((l_1 : \emptyset), \texttt{obj}(@l_1))$$

This type is more flexible than the type from (4.6). The logical type system will support both types for the function.

[3] In the type system, we will be able to always find a local environment Δ describing both objects, but the resulting Δ may be really complicated and ugly, and the worst thing is, that we will lose a lot of information about the structure of the objects, even if this is not necessary.

Program 4.7 $\mathcal{JSC}$ – calling function in different context.

```
1  let f = λ(x).
2    // body of f omitted.
3  in
4  let x = new^l1 in
5  f(x);
6  let z = new^l2 in
7  f(x)
```

4.4.1.3 Access Effect

Until now, our intuition of the most recent environment is that the type system establishes a one-to-one relation between objects in the most recent heap and entries in the most recent environment. Hence, we assume that

$$l \in \mathrm{dom}(\Delta) \longleftrightarrow \exists! i : (l, i) \in \mathrm{dom}(H_0) \tag{4.7}$$

holds for a most recent environment Δ and a suitable most recent heap H_0; that is, if the most recent heap contains an l-object, the most recent heap environment contains an entry for this object, and vice versa. A property like that helps a lot for proving soundness.

A problem with (4.7) is that the type of a function, which contains a most recent environment, has to describe the structure of everything in the most recent heap. Consider Program 4.7 for an example. The first three lines create a function f that takes an object as a parameter and does something with it. Since it is not important what happens inside of f, the function body is omitted in the example. The function is called at two places in the example. The first call is in line 5, the second in line 7. A problem pops up, because in line 6 a new object at abstract location l_2 is created. Typing function f requires the same most recent heap environment at each call site. Since in line 5 the most recent heap environment is $\Delta_5 = (l_1 : \emptyset)$, a possible typing of f, which allows the function call in line 5, is

$$t_f = ((l_1 : \emptyset), t) \xrightarrow{L,A} ((l_1 : \emptyset), t') \tag{4.8}$$

for some types t, t' under the assumption that f does not create new objects. But now have a look at line 7. Here the most recent heap is $\Delta_7 = (l_1 : o_1)(l_2 : \emptyset)$ for a suitable property map o_1. It is easy to see that the most recent heap environments Δ_5 and Δ_7 discern in the existence of the object at abstract location l_2.

We make the function type polymorphic in the part of the most recent heap that is not used by the function body. To achieve the independency we compute an upper bound of abstract locations that are accessed by the function body. We call this upper bound access effect and propagate it in the type system bottom up to the lambda expressions. A type of a function then has the form

$$(\Delta, t) \xrightarrow{L,A} (\Delta, t) \quad , \tag{4.9}$$

$$\textsc{ST-Refl}\quad t <: t \qquad\qquad \textsc{ST-Top}\quad t <: \top \qquad\qquad \textsc{ST-Obj}\quad \frac{L \subseteq L'}{\texttt{obj}(qL) <: \texttt{obj}(qL')}$$

$$\textsc{ST-Fun}\quad \frac{t_1 <: t_1' \qquad t_2' <: t_2 \qquad L \subseteq L' \qquad A \subseteq A'}{(\Delta_2, t_2) \xrightarrow{L,A} (\Delta_1, t_1) <: (\Delta_2, t_2') \xrightarrow{L',A'} (\Delta_1, t_1')}$$

Figure 4.16: $\mathcal{JSR}$ – subtyping. We interpret the subtyping rules co-inductively to establish the subtype relation between types of infinite height.

where A is the access effect of the function. Typing a function call is now possible for all most recent environments Δ' by weakening the condition $\Delta = \Delta'$ to

$$\forall l \in A : \Delta'(l) = \Delta(l) \quad . \tag{4.10}$$

Of course this means that the equivalence from (4.7) is not valid any more, and we can only state that the direction from left to right will hold. We will prove it later in the preservation lemma. It is part of the formulation that the consistency between the most recent heap description and the actual heap is invariant with respect to the semantics.

Allowing the splitting of the most recent heap into two parts is similar to the frame rule used in separation logic [92, 106]. Access effects are a possible way to decide, which part stays unchanged, and which part is passed to the function. Access effect may be seen as some kind of lower bound of the domain of the most recent heap. It is never allowed to split entries with an abstract location away from the most recent heap if the abstract location is part of the access effect. Similarly, the allocation effect may be seen as some kind of negative restriction on the domain of the most recent heap. The type system ensures that for all abstract locations ℓ that are in the allocation effect the most recent heap never contains an entry for such an abstract location. Additionally, if a function gets L as its allocation effect, function application never splits objects with abstract location $\ell \in L$ away. See Section 4.5.3 for more details.

4.4.2 Subtyping

Subtype polymorphism is a feature that most object oriented programming languages support [100]. Figure 4.16 introduces the subtype relation $t_1 <: t_2$ for $\mathcal{JSR}$ by co-induction. To no surprise the relation is reflexive and transitive as usual (cf. Lemma 4.5.6). The rule ST-Top makes $\top$ the supertype of all types. The rule ST-Obj defines the subtyping between object types. A simple example for two object types that are in subtype relation to each other is:

$$\texttt{obj}(\tilde{}\{\ell_1\}) <: \texttt{obj}(\tilde{}\{\ell_1, \ell_2\}) \tag{4.11}$$

$$\textsc{Flow-Env}\quad \frac{\forall \ell \in L : L \vdash_h \Delta(\ell) \lhd \Sigma(\ell) \qquad \Delta' = \Delta^{\natural L} \uparrow L \qquad \Sigma = \Sigma^{\natural L}}{\Sigma, \Delta \rhd^{L} \Delta'}$$

$$\textsc{Flow-ObjRef}\quad \frac{\ell \in L \qquad \ell \in L'}{L \vdash_t \mathtt{obj}(@\{\ell\}) \lhd \mathtt{obj}(\tilde{}L')} \qquad \textsc{Flow-Sub}\quad \frac{t <: t'}{L \vdash_t t \lhd t'} \qquad \textsc{Flow-Obj}\quad \frac{(\forall a \in \mathsf{Prop})\ L \vdash_t r(a) \lhd r'(a)}{L \vdash_h r \lhd r'}$$

Figure 4.17: $\mathcal{JSR}$ – flow. We interpret the flow relation inductively, because the subtype relation already relates types of infinite height.

As the example (4.11) shows, object types with multiple abstract locations are in subtype relation to each other, if they have the same qualifier and if the more general type has a larger set of abstract locations than the more specific type. A consequence of the rule ST-Obj is that our calculus supports union types over summary objects. Because the set L for most recent objects is limited to one element, the calculus does not support union types over most recent objects. In Chapter 6 we discuss an extension of $\mathcal{JSR}$ that supports union types over most recent objects.

The subtype relation treats function arguments contravariant and its return type covariant as usual. It additionally relates the effects of the two function types such that the more general type may have larger effects.

4.4.3 Flow

The flow judgment $\Sigma, \Delta \rhd^{L} \Delta'$ is a relation modeling the movement from most recent objects into the summary heap. It takes the two heap environments Σ, Δ, a set of abstract locations L and yields a new most recent heap environment Δ'.

It ensures that in the summary heap environment there are no objects pointing to precise ℓ objects for $\ell \in L$ by the condition $\Sigma = \Sigma^{\natural L}$. To compute the new most recent heap environment Δ', we have to demote all types inside of Δ with L, and remove all $\ell \in L$ from Δ.

One way to get two types in flow relation is to establish the subtype relation between them. But to be more precise, it is not necessary that the two types are in subtype relation to each other before the objects are moved into the summary heap, but they need to be in subtype relation afterward.

Hence, we would like to support roughly the following implication

$$\Sigma, \Delta \rhd^{L} \Delta' \to (\forall l, a : \Delta(l)(a)^{\natural L} <: \Sigma(l)(a)) \tag{4.12}$$

Instead of using the implication (4.12) as a defining property, we choose a constructive approach for the definition of the flow relation and prove that it fulfills the desired property. The benefit of a constructive approach is that the implementation of the flow relation is simple.

The following example demonstrates that care must be taken by defining the relation. Suppose

$$\Sigma = (\ell_1 : [\texttt{a} : \texttt{string}][\texttt{o} : \texttt{obj}(\tilde{}\ell_3)])(\ell_2 : \emptyset)(\ell_3 : \emptyset)$$

and

$$\Delta = (\ell_1 : [\texttt{a} : \texttt{string}][\texttt{o} : \texttt{obj}(@\ell_3)])(\ell_2 : [b : \texttt{int}])(\ell_3 : \emptyset)$$

In this example the flow relation allows the movement of the ℓ_1 object, if it is moved together with the ℓ_3 object. Moving the object with abstract location ℓ_1 without the ℓ_3 object is forbidden because afterward the ℓ_1 object in the summary heap will have a precise pointer to the ℓ_3 object. This is not allowed since $\Sigma(\ell_1)(o) = \texttt{obj}(\tilde{}\ell_3)$, which is not in subtype relation with $\texttt{obj}(@\ell_3)$. Moving ℓ_1 and ℓ_3 together will relax the constraint of the ℓ_3 object, since then it holds $L \vdash_t \texttt{obj}(@\ell_3) \lhd \texttt{obj}(\tilde{}\ell_3)$ for any L with $\ell_3 \in L$.

Of course it is also allowed to move just the ℓ_3 object since it does not have any properties in both heap environments.

The movement of ℓ_2 is not supported, since the program may read the property of an aged ℓ_2 object. And since map lookup $\emptyset\$\texttt{b}$ returns `udf`, storing an integer in the property `b` would break soundness.

Lemma 4.4.2. *The inductive and co-inductive interpretation of the rules in Figure 4.17 are equivalent.*

Proof. The lemma holds because the domain of heaps and property maps are finite. □

Inductive interpretation of the rules is sufficient because the flow relation does relate function types to each other in the same way as subtyping does. This is due to the fact that demotion does not effect function values and function types.

4.4.4 Demotion of Types and Environments

Figure 4.18 defines the recursive total function $\cdot^{\natural L} : t \to t$ over types. Demotion of `udf` and $\top$ is the identity. The interesting case is to demote precise objects. We change from $\texttt{obj}(@\ell)$ to $\texttt{obj}(\tilde{}\{\ell\})$ if $\ell \in L$. The demote operation on function types is also the identity, since the demote operation on function values is the identity, too. The demote operation naturally expands pointwise to environments.

The intuition behind the demotion on type level is that the following should hold:

$$\Sigma, \Gamma \vdash_w w : t \qquad \text{implies} \qquad \Sigma, \Gamma^{\natural L} \vdash_w w^{\natural L} : t^{\natural L}$$

for all L. The presented definition is sufficient to establish this. The implication is proven in Lemma 4.5.11.

$$
\begin{array}{rcl}
\texttt{udf}^{\natural L} & := & \texttt{udf} \\
\top^{\natural L} & := & \top \\
\texttt{obj}(\tilde{\ }L')^{\natural L} & := & \texttt{obj}(\tilde{\ }L') \\
\texttt{obj}(@\{\ell\})^{\natural L} & := & \begin{cases} \texttt{obj}(\tilde{\ }\{\ell\}) & \text{if } \ell \in L \\ \texttt{obj}(@\{\ell\}) & \text{if } \ell \notin L \end{cases} \\
\left((\Delta_1, t_1) \xrightarrow{L',A} (\Delta_2, t_2)\right)^{\natural L} & := & (\Delta_1, t_1) \xrightarrow{L',A} (\Delta_2, t_2) \\
\emptyset^{\natural L} & := & \emptyset \\
(r[a : \tau])^{\natural L} & := & r^{\natural L}[a : \tau^{\natural L}] \\
(\Delta(\ell : r))^{\natural L} & := & \Delta^{\natural L}(\ell : r^{\natural L}) \\
(\Gamma(x : \tau))^{\natural L} & := & \Gamma^{\natural L}(x : \tau^{\natural L})
\end{array}
$$

Figure 4.18: $\mathcal{JSR}$ – demotion of types and environments. We define the function to demote types and environments recursively. A co-recursive interpretation is not necessary because there is no recursive call for function types.

$$
\begin{array}{ll}
\textsc{T-Undefined} & \textsc{T-Object} \\
\Sigma, \Gamma \vdash_w \texttt{udf} : \texttt{udf} & \Sigma, \Gamma \vdash_w (q\ell, i) : \texttt{obj}(q\ell) \\[2ex]
\textsc{T-Variable} & \\
\dfrac{\Gamma(x) = t}{\Sigma, \Gamma \vdash_w x : t} & \begin{array}{l}\textsc{T-Function} \\ \rightarrow \textit{Figure } 4.22\end{array}
\end{array}
$$

Figure 4.19: $\mathcal{JSR}$ – typing rules for values and variables. Inductive definition of the type judgment for values and variables. Consider Figure 4.22 for the type rule for functions.

4.4.5 Typing of Values and Variables

Figure 4.19 contains the inductive definition of the typing judgment $\vdash_w$. The relation has the form $\Sigma, \Gamma \vdash_w v : t$. Under the summary heap environment Σ and the type environment Γ the value v or the variable x has type t.

The rule T-Undefined types the value `udf` with type `udf` without restricting the environments Γ, Σ in any way. The rule T-Object types object references without looking them up in the most recent/summary heap. It assigns the type $\texttt{obj}(q\ell)$ to an object that has a reference of the form $(q\ell, i)$ for an arbitrary i. The rule T-Variable looks up the type of the variable in the type environment Γ.

Before explaining the type rule for function values (T-Function), we will explain the typing rules for expressions. This makes sense, since the body of a function value is an expression, which is typed during typing the function value itself. Please consider Section 4.4.7 for the explanation of the rule T-Function and have a look at Figure 4.22 for the rule itself. Section 4.4.1.2 contains an introduction to function types.

4.4.6 Typing of Expressions and Top Expressions

The relation $\vdash_s$ types simple expressions, the relation $\vdash_{te}$ types top expressions. Figure 4.20 defines $\vdash_s$ and Figure 4.21 presents the definition of $\vdash_{te}$. Both, $\Sigma, \Delta, \Gamma \vdash_s s : t \Rightarrow L, A, \Delta'$ and $\Sigma, \Delta, \Gamma \vdash_{te} e : t \Rightarrow L, A, \Delta', \Gamma'$ relate an (top) expression s (e) to a type t under a type environment Γ, a global heap environment Σ and a local heap environment Δ. The type relation also produces two effects L and A. The first is the allocation effect, basically collecting the set of all abstract locations, for which the expression may create new objects. The second is the access effect, which collects all abstract locations, for which the expression may access the most recent heap. Both relations produce a new most recent heap environment Δ, because both of them can modify the most recent heap (e.g. by performing function calls or creating new objects). The relation $\vdash_{te}$ also produces a new type environment Γ'. Hence, typing of variables and most recent objects is flow sensitive.

Inside of an expression s there is no possibility to directly change the type of a variable. Only the demote expression can change types of variables, and since it is a top level expression, $\vdash_s$ does not need to handle type of variables flow sensitive. Therefore, the judgment does not return a modified type environment.

4.4.6.1 Expressions

The rule T-Function Call in Figure 4.20 types function calls. The condition $\Delta, \Gamma \vdash_c L$ cleans the most recent heap for all objects with an abstract location $\ell \in L$. The condition $\Delta, A \vdash_S \Delta_1, \Delta_2$ splits the most recent heap Δ into two parts Δ_1 and Δ_2. The access effect is used to decide, which objects are put into which heap. The most recent heap description Δ_1 contains all object with an abstract location ℓ with $\ell \in A$. Δ_2 contains all other objects. The condition

T-Value

$$\frac{\Sigma, \Gamma \vdash_w w : t \qquad t <: t'}{\Sigma, \Delta, \Gamma \vdash_s w : t' \Rightarrow \emptyset, \emptyset, \Delta}$$

T-Function Call

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash_c L \qquad \Delta, A \vdash_S \Delta_1, \Delta_2 \\ \Delta', A \vdash_S \Delta_1', \Delta_2 \qquad \Sigma, \Gamma \vdash_w w_2 : t_2 \qquad \Sigma, \Gamma \vdash_w w_1 : (\Delta_1, t_2) \xrightarrow{L,A} (\Delta_1', t_1)\end{array}}{\Sigma, \Delta, \Gamma \vdash_s w_1(w_2) : t_1 \Rightarrow L, A, \Delta'}$$

T-New

$$\frac{\Delta, \Gamma \vdash_c \{\ell\} \qquad \ell \in \mathrm{dom}(\Sigma)}{\Sigma, \Delta, \Gamma \vdash_s \mathtt{new}^{\ell} : \mathtt{obj}(@\ell) \Rightarrow \{\ell\}, \emptyset, \Delta(\ell \mapsto \{\})}$$

T-Read

$$\frac{\Sigma, \Gamma \vdash_w w : \mathtt{obj}(p) \qquad \Sigma, \Delta \vdash_{\mathfrak{r}} p.a : t \qquad A \vdash_a p}{\Sigma, \Delta, \Gamma \vdash_s w.a : t \Rightarrow \emptyset, A, \Delta}$$

T-Write

$$\frac{\Sigma, \Gamma \vdash_w w : \mathtt{obj}(p) \qquad \Sigma, \Gamma \vdash_w w' : t' \qquad \Sigma, \Delta \vdash_{\mathfrak{w}} p.a := t' \Rightarrow \Delta' \qquad A \vdash_a p}{\Sigma, \Delta, \Gamma \vdash_s w.a := w' : \mathtt{udf} \Rightarrow \emptyset, A, \Delta'}$$

Figure 4.20: $\mathcal{JSR}$ – typing rules for simple expressions ($\vdash_s$). Inductive definition of the type judgement for simple expressions.

$\Delta', A \vdash_S \Delta'_1, \Delta_2$ then later merges the two heap Δ_2 and the heap Δ'_1, which is returned by the function together to a most recent heap description Δ'. The idea behind the splitting is that this allows the function call without passing the complete most recent heap description. Hence, the function is polymorphic in the part of the most recent heap that is not used inside of the function. The two conditions $\Sigma, \Gamma \vdash_w w_2 : t_2$ and $\Sigma, \Gamma \vdash_w w_1 : (\Delta_1, t_2) \xrightarrow{L,A} (\Delta'_1, t_1)$ do the usual check of parameter and function value.

The rule T-New types the new expression. The expression $\texttt{new}^\ell$ gets type $\texttt{obj}(@\ell)$, and the most recent heap environment returned by the type rule contains a new empty object at abstract location ℓ. The condition $\Delta, \Gamma \vdash_c \{\ell\}$ ensures that the most recent heap Δ is empty for the corresponding abstract location, while $\ell \in \mathrm{dom}(\Sigma)$ enforces that there exists a summary heap description for objects of abstract location ℓ. The allocation effect is $\{\ell\}$ and the access effect is empty.

The expression $w.a$ is correctly typed, if the value w has an object type $\texttt{obj}(p)$ and if the property lookup in the static heap environment yields a type t. This lookup is outsourced to the relation $\vdash_{\mathfrak{r}}$; consider Section 4.4.8 for the definition. The condition $A \vdash_a p$ ensures that the access effect of the type rule T-Read contains all abstract locations, for which the reading may be performed. The allocation effect is empty, because reading a property does not create a new object in the heap. The two environments Γ and Δ stay unchanged.

The rule T-Write types the two values w and w' using $\vdash_w$, such that $\texttt{obj}(p)$ is the type of the left hand side (without the label), while t' is the type of the right hand side of the assignment. The condition $\Sigma, \Delta \vdash_{\mathfrak{w}} p.a := t' \Rightarrow \Delta'$ uses the relation $\vdash_{\mathfrak{w}}$ to ensure that the write operation is reflected correctly either in the most recent heap environment, or that the write operation is allowed with respect to the global heap environment. Which one is necessary depends on the precession of the object, which is reflected in the pointer reference p. Both cases are handled by $\vdash_{\mathfrak{w}}$. Since an assignment may change the type of the corresponding object, if p is a precise reference type, the relation $\vdash_{\mathfrak{w}}$ returns a modified most recent heap environment Δ'. Therefore, the type rule T-Write returns Δ' to reflect the possible type change. If p is an imprecise reference type, it holds $\Delta = \Delta'$. Like in the rule for reading properties, the condition $A \vdash_a p$ ensures the correct effect calculation. The allocation effect L is empty, since writing a property does not create new objects in the heap. The type environment Γ stays unchanged.

4.4.6.2 Top Expressions

Figure 4.21 contains the typing rules of the top expressions. Since each value is also an expression, the rule T-Value deals with the values by using $\vdash_w$ to type the value. Of course a value does not have any effects, nor does it change the flow sensitive parts of the type judgment.

The rule T-Demote type the demotion expression. It ensures that the movement of all objects with abstract location $\ell \in L$ is possible with respect to the type environments with the condition $\Sigma, \Delta \rhd^L \Delta'$. The rule also enforces that the

T-Value

$$\frac{\Sigma, \Gamma \vdash_w w : t \qquad t <: t'}{\Sigma, \Delta, \Gamma \vdash_{te} w : t' \Rightarrow \emptyset, \emptyset, \Delta, \Gamma}$$

T-Demote

$$\frac{\begin{array}{c}\Sigma, \Delta \rhd^{L} \Delta' \\ \Sigma, \Delta', \Gamma^{\natural L} \vdash_{te} e : t \Rightarrow L', A, \Delta'', \Gamma''\end{array}}{\Sigma, \Delta, \Gamma \vdash_{te} \natural^{L} e : t \Rightarrow L' - L, A \cup L, \Delta'', \Gamma''}$$

T-Let

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash_c L_1 \qquad \Sigma, \Delta, \Gamma \vdash_s s_1 : t_1 \Rightarrow L_1, A_1, \Delta_1 \\ \Sigma, \Delta_1, \Gamma(x : t_1) \vdash_{te} e_2 : t_2 \Rightarrow L_2, A_2, \Delta_2, \Gamma'(x : t_1')\end{array}}{\Sigma, \Delta, \Gamma \vdash_{te} \texttt{let}\ x = s_1\ \texttt{in}\ e_2 : t_2 \Rightarrow L_1 \cup L_2, A_1 \cup (A_2 - L_1), \Delta_2, \Gamma'}$$

Figure 4.21: $\mathcal{JSR}$ – typing rules for top expressions ($\vdash_{te}$). Inductive.

T-Function

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta) \cap L = \emptyset \qquad L' \cup L'' \cup M \subseteq L \\ \Gamma' = \Gamma \downarrow \mathrm{fv}(\mathbf{rec}^{M} f(x).e) \qquad L' = @\mathrm{Locs}(\Gamma') \qquad \Gamma'' = (\Gamma')^{\natural L'} \\ t_f = (\Delta, t) \xrightarrow{L,A} (\Delta', t') \qquad \Sigma, \Delta, \Gamma''(f : t_f)(x : t) \vdash_{te} e : t' \Rightarrow L'', A, \Delta', \Gamma'''\end{array}}{\Sigma, \Gamma \vdash_w \mathbf{rec}^{M} f(x).e : t_f}$$

Figure 4.22: $\mathcal{JSR}$ – typing rule for function values. Inductive.

body of the demotion expression is typeable under the modified environments Δ' and $\Gamma^{\natural L}$. The allocation effect of a demotion expression is $L' - L$, since the demote operation itself moves all most recent object with an abstract location $\ell \in L$ into the summary heap. Hence, there is no reason to enforce surrounding code to clean the heap for abstract locations in L, which is the purpose of the allocation effect. But it is also important to ensure that surrounding code is not allowed to split away object with abstract locations $\ell \in L$. If that is allowed, the type system cannot establish $\mathrm{INV}_{\mathrm{OH}}$. Hence, we extend the access effect with every abstract location $\ell \in L$.

The let expression (T-Let) basically evaluates the two expressions in sequence and binds the value of the first expression to the variable on the left hand side of the assignment. Hence, the type rule passes the new variable to the second expression with the type that the first expression gets. It also handles the effects with the two conditions $L = L_1 \cup L_2$ and $A = A_1 \cup (A_2 - L_1)$.

4.4.7 Typing of Functions

The rule T-Function (Figure 4.22) enforces a lot of restrictions. This rule is the one that forbid a lot of the problems described in the introduction of the formal calculus. It is not a surprise that the type of a recursive lambda expression is a function type τ_f, and that the function body is typed with an extended type

$$
\begin{array}{lcl}
@\mathrm{Locs}(\_ \xrightarrow{L,A} \_) & = & @\mathrm{Locs}(\mathtt{udf}) = @\mathrm{Locs}(\emptyset) = @\mathrm{Locs}(\mathtt{obj}(\tilde{}L)) = \emptyset \\
@\mathrm{Locs}(\mathtt{obj}(@\{\ell\})) & = & \{\ell\} \\
@\mathrm{Locs}(\Gamma(x:t)) & = & @\mathrm{Locs}(\Gamma) \cup @\mathrm{Locs}(t) \\
@\mathrm{Locs}(\top) & = & \mathsf{Location}
\end{array}
$$

Figure 4.23: $\mathcal{JSR}$ – locations in types and environments.

environment, in which the function parameter and the function itself is bound to their types.

The rule also ensures that a precise singleton pointer is not accessible using free variables and that the most recent heap is cleaned for all objects that may be created during function evaluation. This is done by the following conditions:

1. The allocation effect L'' of the function body is the set of abstract location for which the function body e may create new objects, without handling the demotion itself. Hence, to establish $\mathrm{INV}_{\mathrm{OH}}$ the type rule enforces that all $\ell \in L''$ are not part of the domain of the most recent heap by $\mathrm{dom}(\Delta) \cap L = \emptyset$ and $L'' \subseteq L$.

2. The set L' is used to ensure that free variables only point to aged objects. This is done by $\Gamma' = \Gamma \downarrow \mathrm{fv}(\mathtt{rec}\, f(x).e)$ and $L' = @\mathrm{Locs}(\Gamma')$ and $\Gamma'' = (\Gamma')^{\natural L'}$.

   Since our non-standard substitution demotes singleton references before transporting them into the body of a function this behavior is fulfilled in the dynamic semantics if the correct demotion expressions are inserted before function calls. The demotion is conservative and it causes most of the complication in the typing rule, but it seems that this complication is unavoidable.

   The function @Locs is defined in Figure 4.23. For all types it is the empty set, except for object types. For a type environment Γ it is computed by building the union of the entries.

3. The annotation M of functions exists only in intermediate expressions. In a program that a programmer writes all function expressions have $M = \emptyset$.

   The purpose of the M is to keep track of all abstract locations that were part of the allocation effect of the function before substitution changes free variables into values.

   One important step to prove soundness is, as usual a substitution lemma. It is easier to formulate and prove if the annotation M does ensure that the allocation effect is not reduced by substitution.

Read-Exact
$$\frac{\Delta(\ell)(a) = t}{\Sigma, \Delta \vdash_{\mathfrak{r}} @\ell.a : t}$$

Read-Inexact
$$\frac{\Sigma(\ell)(a) = t}{\Sigma, \Delta \vdash_{\mathfrak{r}} {}^{\sim}\ell.a : t}$$

Read-Union
$$\frac{\forall \ell \in L(\Sigma, \Delta \vdash_{\mathfrak{r}} q\ell.a : t'_\ell \wedge t'_\ell <: t)}{\Sigma, \Delta \vdash_{\mathfrak{r}} qL.a : t}$$

Figure 4.24: $\mathcal{JSR}$ – reading properties.

Write-Inexact
$$\frac{(\forall \ell \in L)\ t <: \Sigma(\ell)(a)}{\Sigma, \Delta \vdash_{\mathfrak{w}} {}^{\sim}L.a := t \Rightarrow \Delta}$$

Write-Exact
$$\frac{\Delta' = \Delta[\ell \mapsto \Delta(\ell)[a \mapsto t]]}{\Sigma, \Delta \vdash_{\mathfrak{w}} @\ell.a := t \rightarrow \Delta'}$$

Figure 4.25: $\mathcal{JSR}$ – writing properties.

4.4.8 Auxiliary Relations

- $\Sigma, \Delta \vdash_{\mathfrak{r}} p.a : t$

  The relation $\vdash_{\mathfrak{r}}$ takes the global heap environment Σ and the most recent environment Δ together with an reference type $p = q\ell$ and the property a and relates them to the type t. It is defined in Figure 4.24. The first two rules state that if the qualifier is precise $q = @$, the type t is equal to the type of $\Delta(\ell)(a)$, and if $q = {}^{\sim}$ then t is equal to $\Sigma(\ell)(a)$.

  The third rule handles the cases, where $p = qL$, with $|L| > 1$. Reading from such a union type is realized by reading each $\ell \in L$, and computing a super type of all the results.

- $\Sigma, \Delta \vdash_{\mathfrak{w}} p.a := t \Rightarrow \Delta$

  The relation $\vdash_{\mathfrak{w}}$ takes the global heap environment Σ, the most recent environment Δ, a reference type $p = q\ell$, the property a and the type of the right hand side of the assignment t and relates it to a new most recent environment if the assignment is valid.

  Two rules define the behavior of the relation $\vdash_{\mathfrak{w}}$. They are presented in Figure 4.25. The first rule handles the case, where $p = {}^{\sim}L$. A write access is only allowed if the type that should be stored inside the object property is a subtype of all the property types. Hence, the condition $t <: \Sigma(\ell)(a)$ is enforced for all $\ell \in L$. The second rule defines how write access to most recent objects is realized. Because an object type with a precise reference only allows one abstract location, the rule just has to update the most recent heap appropriately. Because of $\mathrm{INV}_{\mathrm{OH}}$ it is ensured that this type change does not break soundness.

- $\Delta, \Gamma \vdash_c L$

$$\textsc{T-Clean}\quad \frac{\Gamma = \Gamma^{\natural L} \qquad \Delta = \Delta^{\natural L} \qquad \mathrm{dom}(\Delta) \cap L = \emptyset}{\Delta, \Gamma \vdash_c L}$$

$$\textsc{T-AccessEx}\quad A \cup L \vdash_a @L \qquad\qquad \textsc{T-AccessIn}\quad A \vdash_a \tilde{}L$$

Figure 4.26: $\mathcal{JSR}$ – cleaning the heap, computing access effects.

$$\textsc{Split-Base}\quad \emptyset, A \vdash_S \emptyset, \emptyset \qquad \textsc{Split-Left}\quad \frac{l \in A \qquad \Delta, A \vdash_S \Delta_1, \Delta_2}{\Delta(l : r), A \vdash_S \Delta_1(l : r), \Delta_2} \qquad \textsc{Split-Right}\quad \frac{l \notin A \qquad \Delta, A \vdash_S \Delta_1, \Delta_2}{\Delta(l : r), A \vdash_S \Delta_1, \Delta_2(l : r)}$$

Figure 4.27: $\mathcal{JSR}$ – splitting the most recent heap.

The relation $\Delta, \Gamma \vdash_c L$ ensures that the most recent heap and the type environment are cleaned for the set of abstract locations L. This implies that the domain of Δ does not contain any objects from L, and that there are no references in Δ and Γ pointing to a precise $\ell \in L$. The relation is defined in Figure 4.26. It uses the demote operations on heaps and on the type environment.

- $A \vdash_a p$

  The relation $A \vdash_a p$ collects all abstract locations of p if p is a precise reference. It is defined in Figure 4.26. The first rule handles the situation where $p = @L$, the second rule handles the cases where $p = \tilde{}L$.

- $\Delta, A \vdash_S \Delta, \Delta$

  The split relation $\Delta, A \vdash_S \Delta_1, \Delta_2$ allows splitting the heap Δ with respect to the set of abstract locations A into two parts. It is defined in Figure 4.27. The rule Split-Base states that splitting the empty environment yields two empty environments. The rules Split-Left and Split-Right splits up one entry to the left/right side, depending on if the condition $\ell \in A$ is fulfilled.

  Because splitting is modeled as a relation it is also used to merge the two heaps after the function call.

4.5 Soundness

In order to prove soundness, we take the typical approach, proving progress and preservation. Usually, progress is the easy part, while preservation is the real hard work. In Section 4.5.1 we will introduce the notation of co-induction, and prove some auxiliary lemmas for our type judgments, subtyping, etc.

In $\mathcal{JSR}$ it turns out that progress is not easy to prove. We have to prove that the system does not get stuck. Looking at the rules of the dynamic semantics easily shows that the only possibility to get stuck is due to the conditions in the rules S-New, S-Rd and S-Wrt.

The rule S-New has the condition that the most recent heap does not contain any other objects for the abstract location, for which the rule will create a new object. This condition ensures that INV_{OH} holds in $\mathcal{JSR}$. The type system ensures that the most recent heap environment Δ does not contain ℓ objects if an expression $\texttt{new}^{\ell}$ is typed. Hence, the direct application of the rule is not a problem. This means in the case distinction of the progress lemma, the case of new expressions is trivial.

To apply the rule S-Rd and S-Wrt, the implicit conditions – the reference of the object that should be accessed is actually in the domain of the heap – must be ensured. Interesting is here that the qualifier q, which determines which heap is used for the lookup, is synchronized with the actual position of the object. If the qualifier states that the object should be aged, and the object is part of the most recent heap, the rules are not applicable, and our progress lemma will fail. Hence, we have to prove that this mismatch is not possible.

By proving the absence of the mismatch we prove that it is possible to apply S-Rd and S-Wrt in all situations. The critical issue in the proof is that due to the modified substitution, some references are modified in advance to ensure that for example all free variables of a closure will not be substituted by a precise reference. As a consequence, it is not possible to prove that for all $(\tilde{\ell}, i)$, which are part of an expression, the summary heap contains an object for address (ℓ, i).

Because of these complications we have to face proving progress, we cannot hope to prove it without some invariants that heavily depend on typing information.

Hence, instead of proving progress before preservation as usual, we begin with the substitution lemma, which is proved in Section 4.5.2. In Section 4.5.3 we formulate INV_{OH} together with similar information about the heap and the heap environments.

Next, we have to deal with the issues of the existing references inside of the expression and the heaps. Therefore Section 4.5.5 defines the notation of unblock contexts. With this notation we can formulate an appropriate invariant. The invariant INV_{cls} states that the references inside of the expression and the heaps stay consistent such that program execution is never stopped because an object is part of the wrong heap. Of course we can prove this only for correctly typed programs.

Another precondition for the invariant to hold is that we are talking about correctly preprocessed programs. The preprocessing step has to ensure that each let expression with a new expression or a function call as the right hand side of the assignment is surrounded by a demote expression. Since the let expression has the precondition $\Delta, \Gamma \vdash_c L_1$, if L_1 is the effect for the right hand side of the let expression, typing of programs without demote expressions around these let expressions is not possible for all let expressions inside of the program.

4.5.1 Auxiliary Lemmas

Our presentation in this section rephrases the basic concepts similar to Chapter 21 from „Types and Programming Languages" [100].

Convention 4.5.1. *Let X be a meta variable that ranges over values, expressions, types and environments.*

For the proofs we assume here that we have an universe $\mathcal{U}$ and a monotone function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$.

Definition 4.5.2 (Greatest Fixed Point)**.** *νF denotes the greatest fixed point of F, which is defined by*

$$\nu F := \max\{X \mid X = F(X)\}.$$

Definition 4.5.3 (F-consistent)**.** *For a monotone function F the set $X \subseteq \mathcal{U}$ is F-consistent if $X \subseteq F(X)$.*

Theorem 4.5.4 (Knaster-Tarski [112])**.** *The union of all F-consistent sets is the greatest fixed point of F.*

Definition 4.5.5 (Transitivity)**.** *For an universe $\mathcal{U}$ a relation $R \subseteq \mathcal{U} \times \mathcal{U}$ is transitive if R is closed under the monotone function $TR(R) = \{(x, y) \mid \exists z \in \mathcal{U} : (x, z) \in R \wedge (z, y) \in R\}$ – i.e. if $TR(R) \subseteq R$.*

Lemma 4.5.6. *The subtyping relation <: is reflexive and transitive.*

Proof. **Reflexivity** per definition (ST-Refl).

Transitivity Let S be the monotone function presented in Figure 4.16. Since νS is a fixed point, $\nu S = S(\nu S)$. Hence, $TR(\nu S) = TR(S(\nu S))$. Under the assumption

$$TR(S(\nu S)) \subseteq S(TR(\nu S)) \tag{4.13}$$

we can conclude $TR(\nu S) \subseteq S(TR(\nu S))$. Hence, $TR(\nu S)$ is S-consistent and Theorem 4.5.4 implies $TR(\nu S) \subseteq \nu S$. Hence, νS is transitive.

It remains to prove equation (4.13). We show the stronger fact that for all $R \subseteq \mathcal{U} \times \mathcal{U}$ it holds $TR(S(R)) \subseteq S(TR(R))$.

Let $(t_1, t_2) \in TR(S(R))$. The definition of transitivity yields that there exists a type t' such that $(t_1, t') \in S(R)$ and $(t', t_2) \in S(R)$.

Now we prove $(t_1, t_2) \in S(TR(R))$.

Case distinction over shape of t'.

- *Case* $t' = \top$: Since $(t', t_2) \in S(R)$, the definition of S implies $t_2 = \top$. Hence, $(t_1, t_2) = (t_1, \top) \in S(Q)$ for all Q and therefore $(t_1, t_2) \in S(TR(R))$.
- *Case* $t' = \texttt{obj}(p)$: Since $(t', t_2) \in S(R)$ the definition of S implies $t_2 = \top$, $t_2 = \texttt{obj}(p)$ or $t_2 = \texttt{obj}(p')$.

  Case distinction over shape of t_2.

  - *Case* $t_2 = \top$: This is analogous to the case $t' = \top$.
  - *Case* $t_2 = t' = \texttt{obj}(p)$: Then $(t_1, t_2) \in S(R)$. The definition of TR yields $(t_1, t_2) \in TR(S(R))$ by setting $z = y = t_2$ and $x = t_1$.
  - *Case* $t_2 = \texttt{obj}(p')$: The definition of S yields $p = qL$ and $p' = qL'$ with $L \subseteq L'$. Further, $t_1 = \texttt{obj}(p'')$ holds and we get $p'' = qL''$ with $L'' \subseteq L$. Hence, $L'' \subseteq L'$. By the definition of S $(t_1, t_2) \in S(Q)$ for all Q, hence $(t_1, t_2) \in S(TR(R))$.

  End case distinction over shape of t_2.
- *Case* $t' = \texttt{udf}$: Analog to the object case.
- *Case* $t' = (\Delta_p, t'_p) \xrightarrow{L',A'} (\Delta_r, t'_r)$: The case $t_2 = \top$ is analogous to the case $t' = \top$.

  Otherwise $(t', t_2) \in S(R)$ implies $t_2 = (\Delta_p, t^2_p) \xrightarrow{L_2,A_2} (\Delta_r, t^2_r)$ with $(t^2_p, t'_p) \in R$, $(t'_r, t^2_r) \in R$, $L' \subseteq L_2$ and $A' \subseteq A_2$.

  Similarly, $t_1 = (\Delta_p, t^1_p) \xrightarrow{A_1,L_1} (\Delta_r, t^1_r)$ with $(t'_p, t^1_p) \in R$, $(t^1_r, t'_r) \in R$, $L_1 \subseteq L'$ and $L_2 \subseteq A'$.

  Hence, $L_1 \subseteq L_2$ and $A_1 \subseteq A_2$. The definition of TR implies $(t^1_r, t^2_r) \in TR(R)$ and $(t^2_p, t^1_p) \in TR(R)$. The definition of S implies $(t_1, t_2) \in S(TR(R))$.

End case distinction over shape of t'.

□

Lemma 4.5.7. *For all L and $L' \subseteq L$*

$$(X^{\natural L})^{\natural L'} = X^{\natural L} = (X^{\natural L'})^{\natural L}$$

Proof by induction over the definition of the recursive function $\cdot^{\natural L}$. Let L and L' be fixed with $L' \subseteq L$.

- *Case* X is a value:

  Case distinction over $X = v$.

  - *Case* $X = x, X = \mathtt{rec}\, f(x).e, X = \texttt{udf}, X = (\tilde{\ell}, i)$: trivial, since the demote operation is the identity function.

- *Case* $X = (@\ell, i)$: If $\ell \in L$, it holds $X^{\natural L} = (\tilde{}\ell, i)$, such that another demotion with L' yields the desired result.

  If $\ell \notin L$, then we can conclude $\ell \notin L'$, since $L' \subseteq L$. This is sufficient to prove the desired equation.

  The second part of the equation is analogous.

End case distinction over $X = v$.

- *Case* X is an expression or top level expression: By induction
- *Case* X is a type: A simple case distinction over the structure of types yields the desired result. The interesting case is $X = \texttt{obj}(@\ell)$. It is analogous to the case $X = (@\ell, i)$.
- *Case* X is an environment: For heaps, object maps, type environments, most recent environments and summary environments just apply induction hypothesis.

□

Lemma 4.5.8. *It holds*

$$X^{\natural\emptyset} = X$$

Proof. Trivial by definition of $\cdot^{\natural}$. □

Lemma 4.5.9. *If* $t_1 <: t_2$ *then* $t_1^{\natural L} <: t_2^{\natural L}$.

Proof by induction over the definition of $\cdot^{\natural L}$. First, please notice that the demote function is total.

- *Case* $t_1 = \texttt{obj}(@\ell_1)$: There are two cases for t_2, either $t_2 = \top$ or $t_2 = t_1$. Both are immediate.
- *Case* other: immediate.

□

Lemma 4.5.10. *If* $\Sigma, \Delta, \Gamma \vdash_{te} e : t \Rightarrow L, A, \Delta', \Gamma'$, *then* $dom(\Gamma) = dom(\Gamma')$.

Proof by induction over the type judgment.

- *Case* T-Value: immediate.
- *Case* T-Let: immediate by induction.
- *Case* T-Demote: immediate by induction and the fact that the demote operation on type environment is defined point-wise.

□

Lemma 4.5.11. *If* $\Sigma, \Gamma \vdash_w w : t$ *then* $\Sigma, \Gamma^{\natural L} \vdash_w w^{\natural L} : t^{\natural L}$.

Proof. Case distinction over the typing judgment $\vdash_w$.

- *Case* T-Undefined: immediate
- *Case* T-Variable: immediate
- *Case* T-Object, $t = \mathtt{obj}(qL')$: If $q = \tilde{}$, then $t^{\natural L} = t$. Inversion of $\vdash_w$ yields $v = (\tilde{}\ell, i)$ for $\ell \in L'$. Hence, it also holds $v^{\natural L} = v$. Applying T-Object to the demoted value and type is allowed for an arbitrary type environment.

  If $q = @$, then $L' = \{\ell\}$. Hence, inversion of $\vdash_w$ yields $v = (@\ell, i)$. There are two cases:
  - *Case* $\ell \in L$: It holds $(@\ell, i)^{\natural L} = (\tilde{}\ell, i)$, and $t^{\natural L} = \mathtt{obj}(\tilde{}\{\ell\})$.
  - *Case* $\ell \notin L$: The type and the value do not change and hence the desired result is trivial.
- *Case* T-Function: nothing happens, trivial.

End case distinction over the typing judgment $\vdash_w$. $\square$

Definition 4.5.12 (INV$_{\text{lam}}$)**.** INV$_{\text{lam}}$ *is fulfilled by an expression* e *(*INV$_{\text{lam}}(e)$*), if for all subexpressions* e' *of the form* $e' = \mathbf{rec}^M f(x).e''$*, no exact reference occurs in* e''.

Lemma 4.5.13. *If* INV$_{\text{lam}}(e)$ *and* $H, H_0, e \longrightarrow H', H'_0, e'$*, then* INV$_{\text{lam}}(e')$.

Proof. trivial by induction of $\longrightarrow$. $\square$

Lemma 4.5.14. *If* $C = (\emptyset, \emptyset, e)$ *is a configuration and* INV$_{\text{lam}}(e)$*, then for all* $C' = (H', H'_0, e')$*, with* $C \longrightarrow^* C'$*,* e' *fulfills* INV$_{\text{lam}}(e')$.

Proof. immediate by induction and Lemma 4.5.13. $\square$

4.5.2 Substitution

This subsection contains the proof of the substitution lemma. It is an important step towards soundness, because many rules in the dynamic semantics use substitution.

We prove the lemma in three flavors, once for values, once for expressions, and once for top level expressions. The proof is based on induction over the structure of the value, expression of top expression.

Definition 4.5.15 (Type Environment Abbreviation)**.** *Let* x *range over all variables,* t_x *over all types and* Γ *over all type environments. Then* Γ^x *is a abbreviation for* $\Gamma(x : t_x)$ *and* Γ_x *for* $\Gamma \downarrow \{y \mid y \in \mathit{dom}(\Gamma), y \neq x\}$.

Lemma 4.5.16 (Substitution)**.** *For a type environment* Γ*, a global heap environment* Σ *and a closed* v *with*

$$\Sigma, \Gamma \vdash_w v : t_x, \tag{4.14}$$

it holds:

1. *Substitution on values:*

$$\Sigma, \Gamma(x : t_x) \vdash_w v_0 : t \tag{4.15}$$

implies

$$\Sigma, \Gamma \vdash_w v_0\{x \mapsto v\} : t$$

2. *Substitution on expressions*

$$\begin{gathered}\Sigma, \Delta, \Gamma(x : t'_x) \vdash_s s : t_0 \Rightarrow L, A, \Delta' \\ t_x <: t'_x\end{gathered}$$

implies

$$\Sigma, \Delta, \Gamma \vdash_s s\{x \mapsto v\} : t_0 \Rightarrow L, A, \Delta'$$

3. *Substitution inside of top expressions*

$$\begin{gathered}\Sigma, \Delta, \Gamma(x : t'_x) \vdash_{te} e : t_0 \Rightarrow L, A, \Delta', \Gamma' \\ t_x <: t'_x\end{gathered}$$

implies

$$\Sigma, \Delta, \Gamma \vdash_{te} e\{x \mapsto v\} : t_0 \Rightarrow L, A, \Delta', \Gamma'$$

Closedness of v is needed. Otherwise substitution may change the set of free variables of a lambda abstraction, which in turn may enlarge the set of exact references passed into the body of the lambda abstraction, and thus enlarge the set L' in the typing rule for abstraction (T-Function).

Proof. By induction on the structure of the value, expressions and top level expressions. We make a case distinction over the last applied type rule, to make it better readable.

1. Values

   Case distinction over the structure of the value v_0.

- *Case* $v_0 = y$: Inversion of (4.15) with T-Variable yields

  $$\Gamma(x : t_x)(y) = t$$

  There are two cases, either $x = y$ or $x \neq y$. Both are trivial.

- *Case* $v_0 = \mathtt{rec}^M f(y).e_l$ for a set of locations M: If $x \notin \mathrm{fv}(\mathtt{rec}^M f(y).e_l)$, then the desired result is immediate because v_0 is then not affected by substitution.

  Now assume that $x \in \mathrm{fv}(\mathtt{rec}^M f(y).e_l)$. Hence, $x \in \mathrm{fv}(e_l)$ and $x \neq y$ and $x \neq f$.

  Then, inversion of (4.15) using T-Function yields:

  $$t = t_f = (\Delta_2, t_y) \xrightarrow{L,A} (\Delta_1, t_r) \tag{4.16}$$

  $$\mathrm{dom}(\Delta_2) \cap L = \emptyset \tag{4.17}$$

  $$L' \cup L'' \cup M \subseteq L \tag{4.18}$$

  $$\Gamma' = \Gamma(x : t_x) \downarrow \mathrm{fv}(\mathtt{rec}\, f(y).e_l)$$

  $$L' = @\mathrm{Locs}(\Gamma') \tag{4.19}$$

  $$\Gamma'' = (\Gamma')^{\natural L'}$$

  $$\Sigma, \Delta_2, \Gamma''^{yf} \vdash_{te} e_l : t_r \Rightarrow L'', A, \Delta_1, \Gamma''' \tag{4.20}$$

  $x \in \mathrm{dom}(\Gamma''^{y})$ holds because $x \in \mathrm{fv}(e_l)$. That is why Γ'' from equation (4.20) contains an assignment for x, and we can write $\Gamma'' = \Gamma''_x(x : t_x)$. Rewriting judgment (4.20) to highlight on the important variable x yields (since $x \neq f, x \neq y$):

  $$\Sigma, \Delta_2, \Gamma''^{yf}_x(x : t_x) \vdash_{te} e_l : t_r \Rightarrow L'', A, \Delta_1, \Gamma'''_x(x : t'_x)$$

  The environment $\Gamma'''_x(x : t'_x)$ contains a binding for x because of Lemma 4.5.10.

  Next we show:

  $$\Sigma, \Gamma''^{yf}_x \vdash_w v^\natural : t^\natural_x \tag{4.21}$$

  To figure out (4.21) we make use of (4.14) and Lemma 4.5.11. We are able to change the Γ because v is closed. Notice that if v is closed, then $v^\natural$ is closed, too. So we can apply induction on e_l and $v^\natural$, which yields

  $$\Sigma, \Delta_2, \Gamma''^{yf}_x \vdash_{te} e_l\{x \Mapsto v^\natural\} : t_r \Rightarrow L'', A, \Delta_1, \Gamma'''_x$$

  Thus, the preconditions for T-Function are fulfilled and yields for some M' and M'':

  $$\Sigma, \Gamma \vdash_w (\mathtt{rec}^{M'} f(y).e_l)\{x \Mapsto v\} : (\Delta_2, t_y) \xrightarrow{M'',A} (\Delta_1, t_r) \tag{4.22}$$

Next, we have to show that $M'' = L$. This is trivial, since substitution sets $M' = M \cup @\text{Locs}(v)$, which will compensate the possible change of L' due to removing x from $\Gamma(x : t_x)$.

- *Case* $v_0 = (q\ell, i)$, $v_0 = \texttt{udf}$: The value is not affected by the substitution.

End case distinction over the structure of the value v_0.

2. Top Expressions:

Case distinction over the structure of e, or $\vdash_{te}$.

- *Case* T-Value, $e = v_0$: It must be that $L = \emptyset$, $A = \emptyset$, $\Delta = \Delta'$, and $\Gamma(x : t_x) = \Gamma'$. Inversion of T-Value yields:

$$\begin{gathered}\Sigma, \Gamma(x : t_x) \vdash_w v_0 : t_0' \\ t_0' <: t_0\end{gathered}$$

By induction we can substitute inside of v_0. Hence, the claim holds.

- *Case* T-Let: Suppose that

$$\Sigma, \Delta, \Gamma(x : t_x) \vdash_{te} \texttt{let}\ y = s\ \texttt{in}\ e : t_0 \Rightarrow L, A, \Delta_0, \Gamma_0(x : t_{x0}) \tag{4.23}$$

and $\Sigma, \Gamma \vdash_w v : t_x$.

We can assume $y \neq x$, because if $x = y$ holds, the substitution does not affect the expression, which implies that the lemma holds.

Inversion of (4.23) yields:

$$\begin{gathered}\Delta, \Gamma \vdash_c L_1 \qquad (4.24)\\ L = L_1 \cup L_2 \\ A = A_1 \cup (A_2 - L_1) \\ \Sigma, \Delta, \Gamma(x : t_x) \vdash_s s : t_1 \Rightarrow L_1, A_1, \Delta_1 \qquad (4.25)\\ \Sigma, \Delta_1, \Gamma(x : t_x)(y : t_1) \vdash_{te} e : t_2 \\ \Rightarrow L_2, A_2, \Delta_2, \Gamma_2(x : t_{x0})(y : t_1') \qquad (4.26)\end{gathered}$$

Induction is applicable to (4.25) and yields:

$$\Sigma, \Delta, \Gamma(x : t_x) \vdash_s s\{x \mapsto v\} : t_1 \Rightarrow L_1, A_1, \Delta_1 \tag{4.27}$$

Induction is applicable to (4.26) and yields that:

$$\begin{gathered}\Sigma, \Delta_1, \Gamma(y : t_1) \vdash_{te} e\{x \mapsto v\} : t_2 \\ \Rightarrow L_2, A_2, \Delta_2, \Gamma_2(y : t_1')\end{gathered} \tag{4.28}$$

Finally, the let rule is applicable to (4.27) and (4.28) to yield the desired:

$$\begin{aligned}\Sigma, \Delta, \Gamma \vdash_{te} \texttt{let}\ y = s\{x \Mapsto v\}\ \texttt{in}\ e\{x \Mapsto v\} : t_2 \\ \Rightarrow L_1 \cup L_2, A_1 \cup (A_2 - L_1), \Delta_2, \Gamma_2\end{aligned}$$

That is:

$$\begin{aligned}\Sigma, \Delta, \Gamma \vdash_{te} (\texttt{let}\ y = s\ \texttt{in}\ e)\{x \Mapsto v\} : t_2 \\ \Rightarrow L, A, \Delta_2, \Gamma_2\end{aligned}$$

- *Case* $e = \natural^{L'} e_l$:

  We can assume that $x \in \mathrm{dom}(\Gamma)$, since otherwise the substitution will not affect the expression.

  Please note that compared to the type rule, L and L' is swapped, since L is the effect and L' is the annotation at the demotion expression. Inversion of the type judgment yields:

  $$\Sigma, \Delta \triangleright^{L'} \Delta' \tag{4.29}$$

  $$\Sigma, \Delta', \Gamma^{\natural L'} \vdash_{te} e_l : t \Rightarrow L, A, \Delta'', \Gamma'' \tag{4.30}$$

  $$L' \subseteq L$$

  Because of Lemma 4.5.11 and (4.14):

  $$\Sigma, \Gamma^{\natural L'} \vdash_w v^{\natural L'} : t_x^{\natural L'} \tag{4.31}$$

  Hence, induction hypothesis is applicable to (4.30), x and $v^{\natural L'}$ such that:

  $$\Sigma, \Delta', \Gamma^{\natural L'} \vdash_{te} e_l\{x \Mapsto v^{\natural L'}\} : t_0 \Rightarrow L, A, \Delta'', \Gamma'' \tag{4.32}$$

  Applying T-Demote to (4.29), (4.32) yields the desired result.

End case distinction over the structure of e, or $\vdash_{te}$.

3. Simple Expressions:

   Case distinction over $\vdash_s$.

   - *Case* T-Value: analogous to the case T-Value of the relation $\vdash_{te}$.
   - *Case* T-Function Call, $s = v_1(v_2)$: Immediate by induction.
   - *Case* T-New, $s = \texttt{new}^{\ell}$: Immediate.
   - *Case* T-Read, $s = v_0.a$: Immediate by induction.
   - *Case* T-Write, $s = v_{01}.a := v_{02}$: Immediate by induction.

   End case distinction over $\vdash_s$.

□

$$\frac{\Sigma,\Delta,L,A \Vdash H,H_0 \qquad \Sigma,\Delta,\emptyset \vdash_{te} e : t \Rightarrow L,A,\Delta',\emptyset}{\Sigma,\Delta \Vdash_e H,H_0,e : t \Rightarrow L,A,\Delta'} \qquad \emptyset \Vdash_\Delta H_0$$

$$\frac{\begin{array}{c} L \cap \mathrm{dom}(H_0)_{\downarrow 1} = \emptyset \qquad A \subseteq \mathrm{dom}(\Delta) \\ \forall \ell \in \mathrm{dom}(\Delta) : \exists^{=1} i \ : (\ell,i) \in \mathrm{dom}(H_0) \wedge \Sigma,\Delta \Vdash_o H_0(\ell,i) : \Delta(\ell) \end{array}}{\Delta,L,A \Vdash_\Delta H_0}$$

$$\frac{\begin{array}{c} \mathrm{dom}(H) \cap \mathrm{dom}(H_0) = \emptyset \\ \Delta,L,A \Vdash_\Delta H_0 \qquad \forall (\ell,i) \in \mathrm{dom}(H)\ \ell \in \mathrm{dom}(\Sigma) \wedge \Sigma,\Delta \Vdash_o H(\ell,i) : \Sigma(\ell) \end{array}}{\Sigma,\Delta,L,A \Vdash H,H_0}$$

$$\frac{(\forall a \in \mathrm{dom}(h))\ a \in \mathrm{dom}(r) \wedge \Sigma,\emptyset \vdash_w h(a) : r(a)}{\Sigma,\Delta \Vdash_o h : r}$$

Figure 4.28: $\mathcal{JSR}$ – typing of heaps and configurations.

4.5.3 Heap Consistency

We will now start to make connections between the dynamic and the static semantics. We define the consistency between the static type environments and the dynamic heaps in Figure 4.28. This relation also ensures that for each abstract location at most one object is part of the most recent heap.

There are two other properties the relation establishes. First, it ensures that for all abstract locations $\ell \in L$ there exists no element in the most recent heap H_0 and that for all abstract locations $\ell \in A$ there exists an entry in Δ. So the set A is a lower bound on the domain of Δ. The following lemmata about the consistency relation state some properties:

Lemma 4.5.17. *If* $\Sigma,\Delta,L,A \Vdash H,H_0$ *then* $L \times \mathbb{N} \cap dom(H_0) = \emptyset$.

Proof. Immediate by inversion of the definitions. □

Lemma 4.5.18. *If* $\Sigma,\Delta,L,A \Vdash H,H_0$ *then* $\forall \ell \in A \exists i \in \mathbb{N} : (\ell,i) \in dom(H_0)$.

Proof. Immediate by inversion of the definitions. □

Lemma 4.5.19. *If* $\Sigma,\Delta,L,A \Vdash H,H_0$ *then* $\Sigma,\Delta,L',A' \Vdash H,H_0$ *for* $L' \subseteq L$ *and* $A' \subseteq A$.

Proof. Trivial by inversion of the definition of the heap consistency relation. □

Lemma 4.5.20. *If* $\Sigma,\Delta,L,A \Vdash H,H_0$ *then* $\mathrm{INV_H}(H,H_0)$

Definition 4.5.21 (Type respecting configurations)**.** *A configuration C is* respecting the environments Σ, Δ with type t*, if there exists L, A, Δ' with:*

$$\Sigma, \Delta \Vdash_e C : t \Rightarrow L, A, \Delta' \quad . \tag{4.33}$$

C is a type respecting configuration*, if there exists Σ, Δ and t, such that C is respecting Σ, Δ with t.*

4.5.4 Structure of Function and Demotion Expressions

Definition 4.5.22 (INV$_{\text{df}}$)**.** *For an expression e it holds* INV$_{\text{df}}$ *(*INV$_{\text{df}}$*(e)), if*

1. *for all subexpressions s of e with the form:*

   $$s = (\mathtt{rec}^{M} f(x).e_f)(w)$$

   $M \neq \emptyset$ implies there exists a subexpression e'' of e, such that

   $$e'' = \natural^{L}\mathtt{let}\ x = s\ \mathtt{in}\ e_b$$

   with $M \subseteq L$.

2. *for all subexpressions e' of e of the form:*

   $$e' = \natural^{L} e_b$$

   then

   $$e_b = \mathtt{let}\ x = s\ \mathtt{in}\ e_l \tag{4.34}$$

   for an expression s and top level expressions e_l.

Definition 4.5.23 (Expression written by a programmer)**.** *An expression is written down by the programmer if it fulfills the following constraints:*

1. *The expression does not contain references.*
2. *The M-annotations of function expressions are empty.*
3. *The bodies of all demote expressions are let expressions.*

Lemma 4.5.24 (INV$_{\text{df}}$)**.** *If $C = (\emptyset, \emptyset, e)$ is a type respecting configuration, and e is a program that a programmer is allowed to write down, then e fulfills* INV$_{\text{df}}$*.*

Proof. trivial by induction over the type judgment □

Lemma 4.5.25 (INV$_{\text{df}}$)**.** *If* INV$_{\text{df}}(e)$ *and*

$$\Sigma, \Delta \Vdash_e H, H_0, e : t \Rightarrow L, A, \Delta' \tag{4.35}$$

and $H, H_0, e \longrightarrow H', H_0', e'$, then INV$_{\text{df}}(e')$.

Proof by induction over $\longrightarrow$.

- *Case* S-Dem, $e = \natural^{L} e_m$:

  The only interesting case occurs, if

  $$e_m = \texttt{let } x = \texttt{rec}^{M} f(x).e_f(v) \texttt{ in } e_b$$

  because this is the only subexpression, for which the existence of the surrounding demote expression may change. Due to (4.35) it holds $M \subseteq L$. From the definition of $e \searrow L$ we can conclude $e' = \texttt{let } x = \texttt{rec}^{\emptyset} f(x).e_f(v) \texttt{ in } e_b$.

- *Case* S-App,S-Let,S-New,S-Rd,S-Wrt: use Lemma 4.5.16.

- *Case* S-Let$'$: by induction

□

4.5.5 Unblocked Contexts

To prove soundness, an essential part is to prove that during program execution all references inside of an expression, and all references inside of the heaps, point to valid objects. Since the demote expression moves objects from the most recent heap into the summary heap in order to ensure that at each time only one object per abstract location is part of the most recent heap (INV$_{\mathrm{OH}}$), the crucial point is to prove that if we move an object from the most recent heap to the summary heap, all references pointing to this object are adjusted correctly.

Since our semantics models this movement also during substitution (e.g. $\lambda x.e\{x \mapsto v\}$ demotes everything in v, such that x is substituted by $v^{\natural}$), another important aspect is to prove that the corresponding objects for these references are moved into the summary heap, before the lambda expression is executed. In order to deal with the second aspect, we introduce the notation of unblocked contexts.

Definition 4.5.26 (Unblocked context for references)**.** *An unblocked context $\mathcal{U}^{\ell}$ for an imprecise reference $(\tilde{\ell}, i)$ is defined for all L such that $\ell \notin L$ as*

$$\begin{array}{rcl}
\mathcal{U}^{\ell} & ::= & \Box.a \mid \Box.a := w \\
 & \mid & \texttt{rec}^{L} f(x).\mathcal{U}^{\ell} \mid \natural^{L}\mathcal{U}^{\ell} \\
 & \mid & \texttt{let } x = \mathcal{U}^{\ell} \texttt{ in } e \mid \texttt{let } x = s \texttt{ in } \mathcal{U}^{\ell} \\
 & \mid & \mathcal{U}^{\ell}(w) \mid w(\mathcal{U}^{\ell}) \mid w.a := \mathcal{U}^{\ell}
\end{array}$$

This definition ensures that all references inside of a lambda or a demote expression with $\ell \in L$ are not in an unblocked context.

Definition 4.5.27 (Reference occurrence)**.** *A reference $(q\ell, i)$ occurs in a value v, expression s or top level expression e, if in the abstract syntax tree of v, s or e there exists a leaf of the form $(q\ell, i)$.*

Definition 4.5.28 (Reference correctness)**.** *A reference $(q\ell, i)$ is correct with respect to the heaps H and H_0, $H, H_0 \Vdash (q\ell, i)$, if all of the following holds:*

- $H, H_0 \Vdash (\tilde{}\ell, i)$ *iff* $(\ell, i) \in dom(H) \cup dom(H_0)$ *and*
- $H, H_0 \Vdash (@\ell, i)$ *iff* $(\ell, i) \in dom(H_0)$.

The invariant $\mathrm{INV}_{\mathrm{cls}}$ states closedness of a configuration with respect to the heap. Any reference contained in one of the heaps or in the expression is defined in one of the heaps.

Definition 4.5.29 ($\mathrm{INV}_{\mathrm{cls}}$)**.** $\mathrm{INV}_{\mathrm{cls}}$ *is fulfilled by a configuration $C = (H, H_0, e)$ ($\mathrm{INV}_{\mathrm{cls}}(C)$), if all of the following holds:*

1. *e is closed.*
2. *If $(q\ell, i)$ occurs in e, then $H, H_0 \Vdash (q\ell, i)$.*
3. *If $e = \mathcal{U}^\ell[(\tilde{}\ell, i)]$, then $(\ell, i) \in dom(H)$.*
4. *For all $(\ell, i) \in dom(H) \cup dom(H_0)$, if $h = (H \cup H_0)(\ell, i)$ then, for all $a \in dom(h)$,*
   a) *if $(q\ell', i')$ occurs in $h(a)$, then $H, H_0 \Vdash (q\ell', i')$ and*
   b) *if $h(a) = (\tilde{}\ell', i')$, then $(\ell', i') \in dom(H)$.*

Lemma 4.5.30 ($\mathrm{INV}_{\mathrm{cls}}$)**.** *If $\mathrm{INV}_{\mathrm{df}}(e)$, $\mathrm{INV}_{\mathrm{cls}}(H, H_0, e)$, $H, H_0, e \longrightarrow H', H_0', e'$ and*

$$\Sigma, \Delta \Vdash_e H, H_0, e : t \Rightarrow L, A, \Delta'$$

then $\mathrm{INV}_{\mathrm{cls}}(H', H_0', e')$.

Proof. The first item of the invariant holds trivially.
Case distinction on the definition of the the reduction $\longrightarrow$.

- *Case* S-Dem:

  $$H, H_0, \natural^L e \longrightarrow (H, H_0)^{\natural L}, e \searrow L$$

  Thus with $H_L = H_0^{\natural L} \downarrow \{(\ell, i) \mid \ell \in L, i \in \mathbb{N}\}$, $H' = H^{\natural L} \cup H_L$ and $H_0' = H_0^{\natural L} \backslash H_L$.

  - item 2 holds, because e and $e \searrow L$ have the same set of references and $\mathrm{dom}(H') \cup \mathrm{dom}(H_0') = \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$.
  - Item 3 holds as follows. Suppose that $e \searrow L = \mathcal{U}^\ell[(\tilde{}\ell, i)]$. We have to prove that $(\ell, i) \in \mathrm{dom}(H')$.
    * *Case $\ell \notin L$*: If $\ell \notin L$, then $\mathcal{U}_1^\ell = \natural^L \mathcal{U}^\ell$ is an unblocking context for (ℓ, i) and the original expression $\natural^L e$. Since $\mathrm{INV}_{\mathrm{cls}}$ holds for $\natural^L e$, $(l, i) \in \mathrm{dom}(H)$. This implies $(l, i) \in \mathrm{dom}(H')$.

        - *Case* $\ell \in L$: item 2 yields that $(\ell, i) \in \mathrm{dom}(H') \cup \mathrm{dom}(H_0')$. Since $\ell \in L$, $(\ell, i) \in H_L$. Hence, $(\ell, i) \in \mathrm{dom}(H')$.

    - For item 4, take some $(\ell, i) \in \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$. It holds $(\ell, i) \in \mathrm{dom}(H') \cup \mathrm{dom}(H_0')$. Let $h = (H \cup H_0)(\ell, i)$ and $h' = (H' \cup H_0')(\ell, i)$ with $\mathrm{dom}(h) = \mathrm{dom}(h')$. Let $a \in \mathrm{dom}(h)$ be an arbitrary property.
        - *Case* item 4a:
            - *Case* $(@\ell, i')$ occurs in $h'(a)$: First, $\ell' \notin L$ due to the definition of the demote operation on heaps. Second, because $\mathrm{INV_{cls}}(H, H_0, e)$, it holds $(\ell', i') \in \mathrm{dom}(H_0)$. By definition of H_0' and L it follows $(\ell', i') \in \mathrm{dom}(H_0')$.
            - *Case* $(\tilde{}\ell', i')$ occurs in $h'(a)$: $(\ell', i') \in \mathrm{dom}(H') \cup \mathrm{dom}(H_0') = \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$.

          Thus, item 4a holds.
        - *Case* item 4b: Let $h'(a) = (\tilde{}\ell', i')$. There are two cases:
            - *Case* $\ell' \notin L$: It holds $(\ell', i') \in \mathrm{dom}(H) \subseteq \mathrm{dom}(H')$.
            - *Case* $\ell' \in L$: It holds $(\ell', i') \in \mathrm{dom}(H_L) \subseteq \mathrm{dom}(H_0)$ so that $(\ell', i') \in \mathrm{dom}(H_L) \subseteq \mathrm{dom}(H')$.

          Thus, item 4b holds.

  Hence, $\mathrm{INV_{cls}}(H', H_0', e')$ holds.

- *Case* S-App:

$$H, H_0, (\mathtt{rec}^M f(x).e)(v) \longrightarrow H, H_0, e\{f, x \Mapsto \mathtt{rec}^M f(x).e, v\}$$

    - item 2 holds because substitution only changes precise to imprecise references, which is covered by the definition of $H, H_0 \Vdash (q\ell, i)$.
    - item 3 Suppose $e' = \mathcal{U}^\ell[(\tilde{}\ell, i)]$.

      Due to $\mathrm{INV_{df}}(e)$ we can conclude $M = \emptyset$, which implies $\ell \notin M$. Then the references was unblocked before the execution, which implies $(l, i) \in \mathrm{dom}(H)$. Since the heaps do not change, item 3 is shown.
    - Since the heaps do not change, item 4 is trivial.

- *Case* S-Let, S-New, S-Rd, S-Wrt: trivial

- *Case* S-Let′: by induction

End case distinction on the definition of the the reduction $\longrightarrow$. □

Lemma 4.5.31. *If* $C = (\emptyset, \emptyset, e)$ *is a type respecting configuration and* e *is a program that a programmer is allowed to write down, then* C *fulfills* $\mathrm{INV_{cls}}$.

Proof. The proof is trivial, since programs, written down by the programmer, do not contain references. □

4.5.6 Progress

The invariant $\mathrm{INV}_{\mathrm{cls}}$ is the most important information used to prove progress. Due to Lemma 4.5.30 and Lemma 4.5.31, we prove progress for programs that fulfills the invariant $\mathrm{INV}_{\mathrm{cls}}$. We can than conclude progress holds for all programs the programmer can write down.

Lemma 4.5.32 (Progress)**.** *For a type respecting configuration* $C = (H, H_0, e)$ *that fulfills* $\mathrm{INV}_{\mathrm{cls}}(C)$ *either*

- *there exists* H', H'_0, e'*, such that*

$$H, H_0, e \longrightarrow H', H'_0, e'$$

- *or* e *is a value*

Proof. There are two critical parts in the proof. The first is to ensure that the side conditions of S-Rd and S-Wrt are fulfilled. The second is establish the side condition of the S-New rule. In all other cases, the evaluation rules of $\mathcal{JSR}$ do not have any side condition, which means, if we are able to show that the expression has a structure suitable for the evaluation rule, there is no additional reason that will disallow the application.

The side condition of S-Rd and S-Wrt demands that there is an object for the reference in the heap. $\mathrm{INV}_{\mathrm{cls}}$ ensures that the reference of the expression and the heaps are synchronized correctly.

The side condition of S-New demands that the most recent heap is empty for the abstract location for which a new object should be created. Typing ensures that if an expression has an allocation effect L the most recent heap is not allowed to contain objects with abstract location $\ell \in L$. Therefore, the side condition of S-New is fulfilled.

The proof is by induction over the structure of e. As usual we can omit all cases where e is a value or a variable (the first due to the second item of the progress lemma, the second due to the fact, that $\mathrm{INV}_{\mathrm{cls}}(e)$ ensures that e is closed.)

- *Case* $e = \texttt{let}\ x = s\ \texttt{in}\ e'$: If s is a value, then we can apply S-Let. If s is not a value, then we use induction hypothesis, the rule S-Let$'$ and Lemma 4.2.5.

- *Case* $e = \natural^{L} e'$: We can apply T-Demote.

- *Case* $e = (q\ell, i).a$: From $\mathrm{INV}_{\mathrm{cls}}$ we can conclude that the reference is part of the corresponding heap, which will allow map look up. Hence, S-Rd is applicable.

- *Case* $e = (q\ell, i).a := v$: Analogous, use S-Wrt.

- *Case* $e = \texttt{new}^{\ell}$: The fact that C is a type respecting configuration ensures that the most recent heap does not contain an ℓ object because the allocation effect of the new expression is $\{\ell\}$.

- *Case* else: All other cases are trivial.

□

4.5.7 Preservation

Lemma 4.5.33 (Synchronized demotion)**.** *If*

$$\begin{gathered}\Sigma, \Delta, L', A \Vdash H, H_0 \\ \text{and } \Sigma, \Delta \rhd^L \Delta' \\ \text{and } L \subseteq A\end{gathered}$$

then

$$\Sigma, \Delta', L' \cup L, A - L \Vdash (H, H_0)^{\natural L} \qquad . \tag{4.36}$$

Proof. Let $(H', H_0') = (H, H_0)^{\natural L}$. Inversion of $\Sigma, \Delta \rhd^L \Delta'$ yields:

$$\begin{gathered}\Sigma = \Sigma^{\natural L} \\ \Delta' = \Delta^{\natural L} \uparrow L \\ \forall \ell \in L : L \vdash_h \Delta(\ell) \lhd \Sigma(\ell)\end{gathered}$$

Inversion of $\Sigma, \Delta, L', A \Vdash H, H_0$ yields:

$$\operatorname{dom}(H) \cap \operatorname{dom}(H_0) = \emptyset \tag{4.37}$$
$$\begin{gathered}\Delta, L', A \Vdash H_0 \\ \forall (\ell, i) \in \operatorname{dom}(H) : \ell \in \operatorname{dom}(\Sigma) \wedge \Sigma, \Delta \Vdash H(\ell, i) : \Sigma(\ell)\end{gathered}$$

It holds $\operatorname{dom}(H') \cap \operatorname{dom}(H_0') = \emptyset$ by definition of H and H_0' and (4.37), which is the first condition of (4.36). By inversion of $\Delta, L', A \Vdash H_0$ it holds:

$$\begin{gathered}L' \cap \operatorname{dom}(H_0)_{\downarrow 1} = \emptyset \\ A \subseteq \operatorname{dom}(\Delta) \\ \forall \ell \in \operatorname{dom}(\Delta) : \exists^{=1} i : (\ell, i) \in \operatorname{dom}(H_0) \wedge \Sigma, \Delta \Vdash_o H_0(\ell, i) : \Delta(\ell)\end{gathered}$$

By the definition of H_0' it holds $L \cap \operatorname{dom}(H_0')_{\downarrow 1} = \emptyset$. As a consequence it holds $(L \cup L') \cap \operatorname{dom}(H_0')_{\downarrow 1} = \emptyset$, and hence

$$\Delta', L \cup L', A - L \Vdash H_0' \quad ,$$

because of Lemma 4.5.11. Hence, the second condition of (4.36) is valid and it is left to prove that

$$\forall (\ell, i) \in \operatorname{dom}(H') : \ell \in \operatorname{dom}(\Sigma) \wedge \Sigma, \Delta \Vdash H'(\ell, i) : \Sigma(\ell)$$

holds.

For an arbitrary $(\ell, i) \in \operatorname{dom}(H')$, there are two cases:

- *Case* $(\ell, i) \in \mathrm{dom}(H)$: We can apply Lemma 4.5.11.
- *Case* $(\ell, i) \notin \mathrm{dom}(H)$: It holds by the definition of H' that $(\ell, i) \in \mathrm{dom}(H_0)$, $H'(\ell, i) = H_0(\ell, i)^{\natural L}$ and $\ell \in L$, which implies $\ell \in \mathrm{dom}(\Delta)$. Hence, we apply Lemma 4.5.11.

□

Lemma 4.5.34 (Extension of most recent environment)**.** *If*

$$\begin{gathered}\Sigma, \Delta, \Gamma \vdash_s e : t \Rightarrow L, A, \Delta', \Gamma', \\ \Delta_1 = \Delta(\ell_1 : r_1) \ldots (\ell_n : r_n) \\ \textit{when } \forall i \in \{1, \ldots n\} : \ell_i \notin L \textit{ and } \ell_i \notin \textit{dom}(\Delta)\end{gathered}$$

then

$$\Sigma, \Delta_1, \Gamma \vdash_s e : t \Rightarrow L, A, \Delta'_1, \Gamma'$$

with $\Delta'_1 = \Delta'(\ell_1 : r_1) \ldots (\ell_n : r_n)$.

Proof. By induction over the type judgment. □

Theorem 4.5.35 (Preservation)**.** *For configurations* $C = (H, H_0, e)$ *and* $C' = (H', H'_0, e')$ *with* $C \longrightarrow C'$ *and*

$$\Sigma, \Delta \Vdash_e C : t \Rightarrow L, A, \Delta'$$

there exists L_n, A_n, Δ_n *such that*

$$\Sigma, \Delta_n \Vdash_e C' : t \Rightarrow L_n, A_n, \Delta' \tag{4.38}$$

and $L_n \cup A_n \subseteq L \cup A$.

Proof by induction over $\longrightarrow$.

- *Case* S-Dem:

  $$\begin{gathered}H, H_0, \natural^{L_d} e \longrightarrow (H, H_0)^{\natural L_d}, e \searrow L_d \\ \Sigma, \Delta \Vdash_e H, H_0, \natural^{L_d} e : t \Rightarrow L, A, \Delta'\end{gathered}$$

  Inversion of the typing consistency judgments yields

  $$\begin{gathered}\Sigma, \Delta, L, A \Vdash H, H_0 \\ \Sigma, \Delta, \emptyset \vdash_{te} \natural^{L_d} e : t \Rightarrow L, A, \Delta', \emptyset\end{gathered} \tag{4.39}$$

Now inversion of (4.39) using T-Demote yields that there exits Δ_n, L' and A' with

$$\begin{aligned} \Sigma, \Delta \triangleright^{L_d} \Delta_n \\ L_d \subseteq L \\ \Sigma, \Delta_n, \emptyset \vdash_{te} e : t \Rightarrow L', A', \Delta', \emptyset \qquad (4.40) \\ L = L' - L_d \\ A = A' \cup L_d \end{aligned}$$

It therefore holds $L_d \subseteq A$, $L' = L \cup L_d$ and $A' = A - L_d$. Let $(H', H'_0) = (H, H_0)^{\natural L_d}$. By Lemma 4.5.33, we get

$$\Sigma, \Delta_n, L \cup L_d, A - L_d \Vdash H', H'_0$$

which is

$$\Sigma, \Delta_n, L', A' \Vdash H', H'_0$$

$\Sigma, \Delta_n \vdash_{te} e \searrow L_d : t \Rightarrow L_n, A_n, \Delta'$ follows from (4.40) and the definition of $e \searrow L_d$. $e \searrow L_d$ only affects the M-annotation on a function, which does not affect typing (see T-Function). Hence, (4.38) is shown.

- *Case* S-App:

$$\begin{aligned} H, H_0, (\mathtt{rec}^M f(x).e)(v) \longrightarrow H, H_0, e\{f, x \Mapsto \mathtt{rec}^M f(x).e, v\} \\ \Sigma, \Delta \Vdash_e H, H_0, (\mathtt{rec}^M f(x).e)(v) : t \Rightarrow L, A, \Delta' \end{aligned}$$

Inversion yields

$$\Sigma, \Delta, L, A \Vdash H, H_0 \qquad (4.41)$$

$$\Sigma, \Delta, \emptyset \vdash_s (\mathtt{rec}^M f(x).e)(v) : t \Rightarrow L, \Delta', \emptyset \qquad (4.42)$$

Further inversion of (4.42) with the T-Function Call rule yields

$$\begin{aligned} \Delta = \Delta^{\natural L} \\ \Gamma = \Gamma^{\natural L} \\ \mathrm{dom}(\Delta) \cap L = \emptyset \qquad (4.43) \\ \Delta, A \vdash_S \Delta_1, \Delta_2 \qquad (4.44) \\ \Delta', A \vdash_S \Delta'_1, \Delta_2 \qquad (4.45) \\ \Sigma, \emptyset \vdash_w v : t_2 \qquad (4.46) \\ \Sigma, \emptyset \vdash_w \mathtt{rec}^M f(x).e : (\Delta_1, t_2) \xrightarrow{L,A} (\Delta'_1, t) \qquad (4.47) \end{aligned}$$

Inversion of (4.47) with the T-Function rule yields

$$\begin{aligned}
&\operatorname{dom}(\Delta_1) \cap L = \emptyset \\
&L' \cup L'' \cup M \subseteq L \qquad (4.48)\\
&\Gamma' = \emptyset = (\Gamma \downarrow \operatorname{fv}(\lambda(y,x).e)) \\
&\Gamma'' = \emptyset = \Gamma'^{\natural L'} \\
&\Sigma, \Delta_1, \emptyset(x : t_2) \vdash_s e : t \Rightarrow L'', A'', \Delta'_1, \emptyset(x : t'_2) \qquad (4.49)
\end{aligned}$$

Because of (4.46) v is closed. The function expression itself is closed, too. We apply Lemma 4.5.16 (substitution) on (4.49), and (4.46) and get

$$\begin{aligned}
&\Sigma, \Delta_1, \emptyset \vdash_s e\{f, x \Mapsto \mathtt{rec}^M f(x).e, v\} : t \Rightarrow L'', A'', \Delta'_1, \emptyset \qquad (4.50)\\
&A'' \subseteq A
\end{aligned}$$

Next, we apply Lemma 4.5.34 to (4.50) to get a typing with Δ and Δ'. This is possible due to (4.48), (4.43), (4.44) and (4.45). The heap consistency is valid due to (4.41). Thus it is (4.38).

- *Case* S-Let:

  $$H, H_0, \mathtt{let}\ x = v\ \mathtt{in}\ e \longrightarrow H, H_0, e\{x \Mapsto v\}$$

  Inversion of the heap typing rule for the left-hand side yields

  $$\begin{aligned}
  &\Sigma, \Delta, L, A \Vdash H, H_0 \qquad (4.51)\\
  &\Sigma, \Delta, \emptyset \vdash_s \mathtt{let}\ x = v\ \mathtt{in}\ e : t \Rightarrow L, A, \Delta', \emptyset \qquad (4.52)
  \end{aligned}$$

  Inverting (4.52) with the T-Let rule and the T-Value rule yields

  $$\begin{aligned}
  &\Sigma, \Delta, \emptyset \vdash_s v : t_1 \Rightarrow \emptyset, \emptyset, \Delta, \emptyset \\
  &\Sigma, \emptyset \vdash_w v : t'_1 \qquad (4.53)\\
  &t'_1 <: t_1 \qquad (4.54)\\
  &\Sigma, \Delta, \emptyset(x : t_1) \vdash_{te} e : t \Rightarrow L, A, \Delta_2, \emptyset(x : t'_1) \qquad (4.55)
  \end{aligned}$$

  Applying Lemma 4.5.16 to (4.53) and (4.55) and (4.54) yields the desired

  $$\Sigma, \Delta, \emptyset \vdash_s e\{x \Mapsto v\} : t \Rightarrow L, A, \Delta_2, \emptyset$$

  Because heaps do not change, heap consistency is trivial.

- *Case* S-New: We know that

  $$\begin{aligned}
  &H, H_0, \mathtt{new}^l \longrightarrow H, H_0[(\ell, i) \mapsto \{\}], (@l, i) \qquad (4.56)\\
  &\operatorname{dom}(H_0) \cap (\{\ell\} \times \mathbf{Z}) = \emptyset \\
  &(\ell, i) \notin \operatorname{dom}(H) \\
  &\Sigma, \Delta \Vdash_e H, H_0, \mathtt{new}^l : \mathtt{obj}(@l) \Rightarrow \{l\}, \emptyset, \Delta'' \qquad (4.57)
  \end{aligned}$$

where $\Delta'' = \Delta(l : \{\})$. Now we have to show that there exists Δ' so that

$$\Sigma, \Delta' \Vdash_e H, H_0', (@l, i) : \texttt{obj}(@l) \Rightarrow \emptyset, \emptyset, \Delta'' \tag{4.58}$$

Inversion of (4.57) yields

$$\Sigma, \Delta, \{\ell\}, \emptyset \Vdash H, H_0 \tag{4.59}$$

$$\Sigma, \Delta, \emptyset \vdash_s \texttt{new}^l : \texttt{obj}(@l) \Rightarrow \{l\}, \emptyset, \Delta(\ell \mapsto \{\}), \emptyset \tag{4.60}$$

$$\Delta'' = \Delta(\ell \mapsto \{\})$$

From the definition of $\vdash_s$ and $\vdash_w$ we get

$$\Sigma, \Delta'', \emptyset \vdash_s (@l, i) : \texttt{obj}(@l) \Rightarrow \emptyset, \emptyset, \Delta'', \emptyset$$

(4.59) implies

$$\Sigma, \Delta(\ell \mapsto \{\}), \emptyset, \emptyset \Vdash H, H_0[(\ell, i) \mapsto \{\}]$$

- *Case* S-RD: For $q = @$ or $q = \tilde{}$ it holds

  $$\begin{gathered} H, H_0, (ql, i).a \longrightarrow H, H_0, (H \cup H_0)(l, i)\$a \\ \Sigma, \Delta \Vdash_e H, H_0, (ql, i).a : t \Rightarrow \emptyset, A, \Delta \\ L = \emptyset \end{gathered} \tag{4.61}$$

  We have to show that

  $$\begin{gathered} \Sigma, \Delta \Vdash_e H, H_0, (H \cup H_0)(l, i)\$a : t \Rightarrow L_n, A_n, \Delta \\ L_n \cup A_n \subseteq A \end{gathered} \tag{4.62}$$

  Inverting (4.61) gives us consistency of heaps and their typings and the typing of the expression

  $$\Sigma, \Delta, \emptyset, A \Vdash H, H_0 \tag{4.63}$$

  $$\Sigma, \Delta, \emptyset \vdash_s (ql, i).a : t \Rightarrow \emptyset, A, \Delta, \emptyset$$

  and after another inversion of the second part

  $$\begin{gathered} \Sigma, \emptyset \vdash_w (ql, i) : \texttt{obj}(qL_r) \\ \Sigma, \Delta \vdash_{\mathfrak{r}} ql.a : t \\ A \vdash_a qL_r \\ l \in L_r \end{gathered} \tag{4.64}$$

  Case distinction over the precision of the pointer.

- *Case* $ql = @l, \Rightarrow (l,i) \in \mathrm{dom}(H_0)$:
  Since (4.64) $l \in \mathrm{dom}(\Delta)$. Because of (4.63) we know

  $$\Sigma, \Delta \Vdash_o H_0(l,i) : \Delta(l)$$

  Cause of (4.64) we get $\Delta(l)(a) = t$. Inverting the line above yields

  $$\Sigma, \emptyset \vdash_w H_0(l,i)\$a : t$$

- *Case* $ql = \tilde{}l, \Rightarrow (l,i) \in \mathrm{dom}(H)$:
  The inversion of (4.64) yields $\Sigma(l)(a) = t', t' <: t$. From (4.63) we get $\Sigma, \Delta \Vdash_o H(l,i) : \Sigma(l)$, hence

  $$\Sigma, \emptyset \vdash_w H(l,i)\$a : t \quad .$$

  As in the other case we fullfill (4.62).

End case distinction over the precision of the pointer. In both cases $L_n = \emptyset$ and $A_n = \emptyset$. Therefore, $L_n \cup A_n = \emptyset \subseteq L \cup A$.

- *Case* S-WRT: Hence, $e = (q\ell, i).a := v$. Inversion of the configuration typing rule yields:

$$\begin{gathered} L = \emptyset \\ \Sigma, \Delta, \emptyset \vdash_s (q\ell, i).a := v : \mathtt{udf} \Rightarrow \emptyset, A, \Delta', \emptyset \qquad (4.65) \\ \Sigma, \Delta, \emptyset, A \Vdash H, H_0 \end{gathered}$$

Further inversion of (4.65) yields

$$\begin{aligned} &\Sigma, \emptyset \vdash_w (q\ell, i) : \mathtt{obj}(qL') && (4.66) \\ &\Sigma, \emptyset \vdash_w v : t \\ &\Sigma, \Delta \vdash_{\mathfrak{w}} qL'.a := t \Rightarrow \Delta' && (4.67) \end{aligned}$$

(4.66) yields : $\ell \in L'$.

Case distinction over q.

- *Case* $q = @$: We can conclude that the update results in a change to the most recent heap and in a new local environment description where a strong update happens. The new heap is consistent with the new local environment description. The expression `udf` is typed.
- *Case* $q = \tilde{}$: S-WRT changes the property of the object in the summary heap. The inversion of the judgment $\vdash_{\mathfrak{w}}$ ensures that the value that is written to the property has a type that is a subtype of $\Sigma(\ell)(a)$. Hence, heap consistency is ensured after the update. Typing the expression `udf` is immediate.

End case distinction over q.

- *Case* S-LET$'$: Inversion of the rule yields

$$H, H_0, \texttt{let } x = s \texttt{ in } e'' \longrightarrow H', H_0', \mathcal{E}[\texttt{let } x = w \texttt{ in } e'']$$
$$H, H_0, s \longrightarrow H', H_0', \mathcal{E}[w]$$

By induction it holds there exists L_n', A_n' and Δ_n' such that

$$\Sigma, \Delta_n' \Vdash_e (H', H_0', \mathcal{E}[w]) : t \Rightarrow L_n', A_n', \Delta'$$

which implies

$$\Sigma, \Delta_n', \emptyset \vdash_{te} \mathcal{E}[w] : t \Rightarrow L_n', A_n', \Delta', \emptyset \tag{4.68}$$

From (4.68) we can conclude that the sequence of let and demote expressions building $\mathcal{E}$ does type. These typing judgments, the typing rule T-LET, especially how it propagates its environments, and the fact that the original configuration is type respecting is sufficient to prove that the new configuration C' is type respecting, too.

□

Corollary 4.5.36 (Extended preservation). *For configurations $C = (H, H_0, e)$ and $C' = (H', H_0', e')$ with $C \longrightarrow^* C'$ and*

$$\Sigma, \Delta \Vdash_e C : t \Rightarrow L, A, \Delta'$$

there exists L_n, A_n, Δ_n such that

$$\Sigma, \Delta_n \Vdash_e C' : t \Rightarrow L_n, A_n, \Delta'$$

and $L_n \cup A_n \subseteq L \cup A$.

Proof. An induction on the length of the evaluation sequence, Theorem 4.5.35 yields the desired claim. □

4.5.8 Soundness

Theorem 4.5.37 (Soundness). *Let $C = (H, H_0, e)$ be a type respecting configuration that fulfills* $\mathrm{INV}_{\mathrm{cls}}(C)$*.*

Let $\mathcal{C} = \{C' \mid C \longrightarrow^* C'\}$ be the list of reachable configurations from C. All configurations from $\mathcal{C}$ are type respecting and it holds either:

- *for all values v there exists no $n \in \mathbb{N}$ such that $C \longrightarrow^n (H^n, H_0^n, v)$ or*

- *there exists an $n \in \mathbb{N}$ and a value v with $C \longrightarrow^n C^n$ and $e^n = v$ and there does not exists C' with $C^n \longrightarrow C'$.*

Proof. The claim follows by induction using Lemma 4.5.32 and Corollary 4.5.36. □

Corollary 4.5.38 (Soundness for programs)**.** *For a type respecting configuration $C = (\emptyset, \emptyset, e)$ that the programmer is allowed to write down, all reachable configurations are type respecting, and either contain a value or can be further evaluated.*

Proof. The formulation in natural language is just a direct consequence of Theorem 4.5.37, Lemma 4.5.31 and Corollary 4.2.21. □

4.6 Decidability

An interesting property of the type system of $\mathcal{JSR}$ is if its logical type system is decidable; that is to prove that there exists a Turing machine, which decides for given $\Sigma, \Delta, \Gamma, e, t, L, A, \Delta', \Gamma'$ if

$$\Sigma, \Delta, \Gamma \vdash_{te} e : t \Rightarrow L, A, \Delta', \Gamma'$$
$$\Sigma, \Delta, \Gamma \vdash_{s} s : t \Rightarrow L, A, \Delta'$$
$$\Sigma, \Gamma \vdash_{w} w : t$$

holds. A necessary, but not sufficient, condition for the decidability of a type system is the decidability of its auxiliary relations. Especially the subtype relation is important in this context because it is often the reason for a type system to become undecidable.

4.6.1 Co-inductive and Regular Types

For well-formed co-inductive types $t \in \mathsf{Type}$, as defined in Figure 4.14 and Figure 4.15, it turns out that the type system is not decidable.

Definition 4.6.1 (Well-formed types)**.** $\mathsf{Type}_{\mathfrak{wf}} := \{t \mid t \in \mathsf{Type} \wedge \vdash_{\mathfrak{wf}} t\}$.

The reason for that is that the set $\mathsf{Type}_{\mathfrak{wf}}$ is not countable. Because there is no bijection between $\mathsf{Type}_{\mathfrak{wf}}$ and $\mathbb{N}$, there cannot exist a Turing machine that decides any form of relation over types.

Lemma 4.6.2. $\mathsf{Type}_{\mathfrak{wf}}$ *is uncountable.*

Proof. Let be $\mathsf{Location} = \{0, \ldots, 9\}$. Consider the function $T : [0,1] \to \mathsf{Type}$:

$$T(r) := (\emptyset, \mathtt{obj}(r_1)) \longrightarrow (\emptyset, \mathtt{obj}(r_2)) \longrightarrow \ldots \text{ for } r = 0.r_1 r_2 \ldots$$

For each $r, r' \in [0,1]$, if $r \neq r'$, then $T(r) \neq T(r')$. Hence, T is an injective function. Since $[0,1]$ is uncountable, Type is uncountable, too. □

$$
\begin{array}{rll}
\mathsf{types}(t) & := \{t\} \cup \bigcup_i(\mathsf{types}'(\Delta_i, t_i)) & t = (\Delta_1, t_1) \xrightarrow{L,A} (\Delta_2, t_2) \\
\mathsf{types}(t) & := \{t\} & \text{otherwise} \\
\mathsf{types}(p) & := \emptyset & \\
\mathsf{types}(\emptyset) & := \emptyset & \\
\mathsf{types}(r[a : t]) & := \mathsf{types}'(r, t) & \\
\mathsf{types}(\Delta(\ell : r)) & := \mathsf{types}'(\Delta, r) & \\
\mathsf{types}(\Gamma(x : t)) & := \mathsf{types}'(\Gamma, t) & \\
\mathsf{types}'(Y_1, Y_2) & := \mathsf{types}(Y_1) \cup \mathsf{types}(Y_2) &
\end{array}
$$

Figure 4.29: $\mathcal{JSR}$ – definition of the co-recursive function types. types(Y) computes the set of all types that are part of Y. Y is a wildcard for all elements from $\mathcal{U}$.

Corollary 4.6.3. *The type system of $\mathcal{JSR}$ is undecidable.*

Proof. As a consequence from Lemma 4.6.2 there exists no finite encoding for all types. □

Due to Corollary 4.6.3 we have to restrict ourselfs to a smaller set of types. As usual, we restrict the type system of $\mathcal{JSR}$ to regular types [100].

Definition 4.6.4 ($\mathcal{JSR}_r$)**.** *Let $\mathcal{JSR}_r$ be the regular subset of $\mathcal{JSR}$.*

The next paragraphs prove the decidability of $\mathcal{JSR}_r$.

Convention 4.6.5. *We define the universe $\mathcal{U}$ to contain all elements of interest:*

$$\mathcal{U} := \mathit{Type} \cup \mathit{Reference} \cup \mathit{HeapType} \cup \mathit{SummaryEnv} \cup \mathit{SingletonEnv} \cup \mathit{TypeEnv}$$

The co-recursive function types : $\mathcal{U} \rightarrow 2^{\mathsf{Type}}$ is defined in Figure 4.29. It computes the set of all types that are part of a structure $Y \in \mathcal{U}$.

Definition 4.6.6 (regular type)**.** *A type $t \in$ Type is regular if types(t) is finite. The set of regular types is written Type_r.*

Theorem 4.6.7. *The set Type_r is countable.*

Proof. Obviously, Type_r contains infinite many members. Section 4.6.2 provides a finite representation for each element in Type_r in form of a μ-type. That proves the countability of Type_r. □

Theorem 4.6.8. *The type system of $\mathcal{JSR}_r$ is decidable.*

Proof. The only critical relation for the decidability of $\mathcal{JSR}_r$ is the subtype relation. All other relations and functions are defined inductively or by recursion and are therefore straightforward to implement. Section 4.6.2 presents an inductive algorithm that decides the subtype relation of μ-types. Hence, a Turing machine exists that decides the subtype relation for regular types. As a consequence, the type system of $\mathcal{JSR}_r$ is decidable. □

$$\begin{array}{rcl}
\mathsf{Type}_\mu \ni & t & ::= \mathtt{obj}(p) \mid (\Delta, t) \xrightarrow{L,A} (\Delta, t) \mid \top \mid \mathtt{udf} \mid \mu X.t \mid X \\
\mathsf{Reference} \ni & p & ::= \tilde{}L \mid @\{\ell\} \qquad\qquad \text{with } |L| \geq 1 \\
\mathsf{HeapType}_\mu \ni & r & ::= \emptyset \mid r[a : t] \\
\mathsf{SummaryEnv}_\mu \ni & \Sigma & ::= \emptyset \mid \Sigma(\ell : r) \\
\mathsf{SingletonEnv}_\mu \ni & \Delta & ::= \emptyset \mid \Delta(\ell : r) \\
\mathsf{TypeEnv}_\mu \ni & \Gamma & ::= \emptyset \mid \Gamma(x : t) \\
\mathsf{TypeVariable} \ni & X &
\end{array}$$

Figure 4.30: $\mathcal{JSR}_\mu$ – type syntax. Inductive. The difference between the inductive definition in this figure and the definition from Figure 4.14 is marked in gray.

$$\begin{array}{rcl}
\mathsf{tree}(\mu X.t) & := & \mathsf{tree}(t[X \mapsto \mu X.t]) \\
\mathsf{tree}((\Delta_1, t_1) \xrightarrow{L,A'} (\Delta_2, t_2)) & := & (\mathsf{tree}(\Delta_1), \mathsf{tree}(t_1)) \xrightarrow{L,A'} (\mathsf{tree}(\Delta_2), \mathsf{tree}(t_2)) \\
\mathsf{tree}(t) & := & t \qquad \text{otherwise} \\
\mathsf{tree}(p) & := & p \\
\mathsf{tree}(\emptyset) & := & \emptyset \\
\mathsf{tree}(r[a : t]) & := & \mathsf{tree}(r)[a : \mathsf{tree}(t)] \\
\mathsf{tree}(\Delta[\ell : r]) & := & \mathsf{tree}(\Delta)[\ell : \mathsf{tree}(r)] \\
\mathsf{tree}(\Sigma[\ell : r]) & := & \mathsf{tree}(\Sigma)[\ell : \mathsf{tree}(r)] \\
\mathsf{tree}(\Gamma[x : t]) & := & \mathsf{tree}(\Gamma)[x : \mathsf{tree}(t)]
\end{array}$$

Figure 4.31: $\mathcal{JSR}_\mu$ – function tree for closed types. Inductive.

4.6.2 μ-Types

This paragraph contains a sketch how to map recursive types to μ-types to establish a connection between $\mathcal{JSR}_r$ and $\mathcal{JSR}_\mu$. The presentation is rough because this mapping is well understood in the literature. For example "Types and Programming Languages" [100] explains the concepts in detail in Part IV, Recursive Types.

Figure 4.30 presents the set Type_μ. We assume the μ-Types are contractive (see [100] for details). It holds that Type_μ has the same cardinality as Type_r. Please note, that the interpretation of the rules in the version with μ-types is inductive. Figure 4.31 defines a function $\mathsf{tree} : \mathsf{Type}_\mu \to \mathsf{Type}_r$ by induction. Roughly spoken, it maps μ-types to recursive types by unfolding the variables. Figure 4.32 presents the well-formedness relation for μ-types. The only difference between the relation for μ-types and regular types is that we add two new rules for μ-types and for type variables. A type $t' \in \mathsf{Type}_\mu$ is well-formed if it fulfills the relation $\emptyset \vdash^*_{\mathfrak{wf}} t'$. For

$$A \vdash^*_{\mathfrak{wf}} \top \qquad\qquad A \vdash^*_{\mathfrak{wf}} \texttt{udf} \qquad\qquad A \vdash^*_{\mathfrak{wf}} \texttt{obj}(p)$$

$$\frac{A \vdash^*_{\mathfrak{wf}} \Delta_1 \qquad A \vdash^*_{\mathfrak{wf}} \Delta_2 \qquad A \vdash^*_{\mathfrak{wf}} t_1 \qquad A \vdash^*_{\mathfrak{wf}} t_2}{A \vdash^*_{\mathfrak{wf}} (\Delta_1, t_1) \xrightarrow{L,A} (\Delta_2, t_2)} \qquad \frac{X \in A}{A \vdash^*_{\mathfrak{wf}} X} \qquad \frac{A \cup \{X\} \vdash^*_{\mathfrak{wf}} t}{A \vdash^*_{\mathfrak{wf}} \mu X.t}$$

$$A \vdash^*_{\mathfrak{wf}} \emptyset \qquad\qquad \frac{A \vdash^*_{\mathfrak{wf}} r \qquad A \vdash^*_{\mathfrak{wf}} t \qquad a \notin \mathrm{dom}(r)}{A \vdash^*_{\mathfrak{wf}} r[a : t]}$$

$$\frac{A \vdash^*_{\mathfrak{wf}} \Delta \qquad A \vdash^*_{\mathfrak{wf}} r \qquad \ell \notin \mathrm{dom}(\Delta)}{A \vdash^*_{\mathfrak{wf}} \Delta(\ell : r)} \qquad \frac{A \vdash^*_{\mathfrak{wf}} \Gamma \qquad A \vdash^*_{\mathfrak{wf}} t \qquad x \notin \mathrm{dom}(\Gamma)}{A \vdash^*_{\mathfrak{wf}} \Gamma(x : t)}$$

Figure 4.32: $\mathcal{JSR}_\mu$ – well-formedness types and environments. Inductive.

$t' \in \mathsf{Type}_\mu$ and $t = \mathsf{tree}(t')$ it holds:

$$\vdash_{\mathfrak{wf}} t \Leftrightarrow \emptyset \vdash^*_{\mathfrak{wf}} t' \tag{4.69}$$

We continue by defining a subtype relation over μ-types (Figure 4.33), such that for a μ-type t'_1, t'_2 and $t_i = \mathsf{tree}(t'_i), i \in \{1, 2\}$ it holds:

$$t_1 <: t_2 \Leftrightarrow t'_1 <:^* t'_2 \tag{4.70}$$

For all other definitions, we just have to adjust the relation that is used when we are ensuring subtyping between two types. Consider the definition of the flow relation as an example (Figure 4.34). The only change is highlighted with a gray box. If we define $t <: t'$ as a shortcut for $t <:^* t'$, we can literally use the definition of all relations and functions as they were presented and just decide which version we use by looking at the types. If we use types from Type_r, we use the co-inductive definition restricted to regular types, and if we use types form Type_μ, we will use the finite representation as μ-types.

From Figure 4.33 we can deduce an algorithm that decides the subtype relation by following the approach from “Types and Programming Languages” [100].

$$\textsc{ST-Refl}\quad t <:^* t \qquad\qquad \textsc{ST-Top}\quad t <:^* \top \qquad\qquad \textsc{ST-Obj}\quad \frac{L \subseteq L'}{\texttt{obj}(qL) <:^* \texttt{obj}(qL')}$$

$$\textsc{ST-Fun}\quad \frac{t_1 <:^* t_1' \qquad t_2' <:^* t_2 \qquad L \subseteq L' \qquad A \subseteq A'}{(\Delta_2, t_2) \xrightarrow{L,A} (\Delta_1, t_1) <:^* (\Delta_2, t_2') \xrightarrow{L',A'} (\Delta_1, t_1')} \qquad \textsc{ST-}\mu\textsc{Left}\quad \frac{t[X \mapsto \mu.t] <:^* t'}{\mu X.t <:^* t'}$$

$$\textsc{ST-}\mu\textsc{Right}\quad \frac{t <:^* t'[X \mapsto \mu.t']}{t <:^* \mu X.t'}$$

Figure 4.33: $\mathcal{JSR}_\mu$ – subtyping relation. Inductive. The subtype relation for μ-types shares the first four inference rules with the subtyping relation <: from Figure 4.16. The relation is extended by two rules (ST-μLeft, ST-μRight) to handle μ-types.

$$\frac{\forall \ell \in L : L \vdash_h \Delta(\ell) \lhd^* \Sigma(\ell) \qquad \Delta' = \Delta^{\natural L} \uparrow L \qquad \Sigma = \Sigma^{\natural L}}{\Sigma, \Delta \rhd^L \Delta'}$$

$$\frac{\ell \in L \qquad \ell \in L'}{L \vdash_t \texttt{obj}(@\{\ell\}) \lhd^* \texttt{obj}(\tilde{}L')} \qquad \frac{t <:^* t'}{L \vdash_t t \lhd^* t'} \qquad \frac{(\forall a \in \mathsf{Prop})\ L \vdash_t r(a) \lhd^* r'(a)}{L \vdash_h r \lhd^* r'}$$

Figure 4.34: $\mathcal{JSR}_\mu$ – flow relation for μ-types. Inductive. The difference between the definition of this figure and the definition of the flow relation from Figure 4.17 is marked in gray.

5 Type Inference

The chapter about type inference first proves the decidability of type inference for $\mathcal{JSR}$. Secondly, it argues that a type inference based on guessing annotations of new expressions and demote expressions is not feasible, because the annotation space is to large. Therefore, it sketches an approach based on constraint generation that is useful in practice.

The prototype implementation of type inference consists of roughly 9000 lines of OCaml code. It uses a syntax-directed version of the type system to generate constraints (essentially equality, subtyping, flow, and map constraints, where the latter constrain the domain of a singleton environment). The solver for these constraints builds a hyper-graph where variables (e.g. type variables α, location set variables μ) are vertices and constraints are hyperedges.

The implementation of the inference algorithm is available on the web at `http://proglang.informatik.uni-freiburg.de/JavaScript/`. The implementation infers the types and locations for all examples presented in our papers [56, 57].

5.1 Decidability

Before we present the type inference algorithm we discuss in this section the decidability of type inference. Before we can discuss the decidability of a type inference algorithm we define what the task of a type inference algorithm actually is. Please first recapitulate the syntax definition of $\mathcal{JSR}$ (Figure 4.7). One can think of type inference for $\mathcal{JSR}$ as the following:

For a given top expression e, a simple expression s or for a given value v compute $\Gamma, \Gamma', \Delta, \Delta', \Sigma, t, L, A$ such that

$$\Sigma, \Delta, \Gamma \vdash_{te} e : t \Rightarrow L, A, \Delta', \Gamma$$
$$\Sigma, \Delta, \Gamma \vdash_{s} s : t \Rightarrow L, A, \Delta'$$
$$\Sigma, \Gamma \vdash_{w} v : t$$

holds.

The definition of the type judgments and their auxiliary relations directly shows that the computation of these values is not more complicated than type checking itself. Therefore, it is simple to see that type inference is decidable for $\mathcal{JSR}$, too.

We will not invest a lot of effort in proving this statement, because for practice a far more important question arises; that is, can we find an algorithm that is capable of computing the types *and the demote expressions with their annotations* for a expression from $\mathcal{JSC}$. So, our central question in this section is, can we infer additionally to the types, effects and environments the position and annotation of the demote expressions?

First, we formulate this question more mathematically. For a given expression $e \in \mathcal{JSC}$, does there exits an expression $f \in \mathcal{JSR}$ with $e \sim f$, such that there exists $\Gamma, \Gamma', \Delta, \Delta', \Sigma, t, L, A$ with

$$\Sigma, \Delta, \Gamma \vdash_{te} f : t \Rightarrow L, A, \Delta', \Gamma' \tag{5.1}$$

Another difference between expressions from $\mathcal{JSC}$ and $\mathcal{JSR}$ is that new expressions are annotated with abstract locations. To infer the annotation of new expressions is also the task of type inference for expressions in $\mathcal{JSC}$. Hence type inference, additionally to the traditional task of type inference, has to address the following questions:

1. What is the annotation of a new expression?

2. Where to insert demote expressions into the expression?

3. What is the annotation of a demote expression?

Question 1 Because the source code of the program is finite, the number of new expressions of the program is finite, too. If we think about all possible annotations for these new expressions using the full set of abstract locations Location, of course there are infinite many possibilities.

But if we are interested in answering the question, if there exists an assignment of annotations for the new expressions, such that the program type checks, we do not need to check all these possibilities. The actual abstract location is irrelevant for type checking. The only information of value is, which new expressions create objects of the same abstract locations, and which will create different objects. Hence, we can easily define an equivalence relation over programs that abstract from the concrete abstract locations. Now it is sufficient for type checking a program, to find an equivalence class, such that a randomly picked representative from the class does pass type checking, or to prove that all equivalence classes do not pass type checking. Because the number of equivalence classes is finite, for decidability we may assume that we will guess the correct class if it exists.

Question 2 The syntax of $\mathcal{JSR}$ does enforce strong restriction about where to introduce demote expressions. Our first step is to put around each let expression a demote expression with an unknown location set. It is sufficient to only put one demote expression there, because of the following obvious law:

$$(e^{\natural L_2})^{\natural L_1} \equiv e^{\natural L_1 \cup L_2}$$

The equivalence relation $\equiv$ means there exists n, m, such that if we perform n steps on the left side, we will end in the same configuration, as if we execute

m steps on the right side. For these two expressions, obviously it holds that for $n = 2$ and $m = 1$ the desired configuration is found, because it holds:

$$\begin{gathered}(H, H_0, (e^{\natural L_2})^{\natural L_1}) \longrightarrow^2 H', H_0', e' \\ (H, H_0, e^{\natural L_1 \cup L_2}) \longrightarrow H', H_0', e'\end{gathered}$$

Reconsidering the type judgment also leads to the insight, that if the right side of the equation is not typeable, then the left side is also not typeable. Hence, for type inference, we do not have to introduce two demote operations that are directly nested. From this fact we can conclude, that the number of possible places to insert demote expressions is also finite.

Question 3 Because the set of abstract locations of a program with already annotated new expressions is finite, and because all possible positions of demote expressions is finite, we just have to guess for each of the demote expressions a set of abstract location $L \subseteq \mathsf{Location}^*$.

Hence, the decidability of type inference depends only one the question, whether we can compute the type and the environments for a given program that is fully annotated; that means, its new expressions are annotated and suitable demote expressions with abstract location sets are inserted, too.

Because for μ-types the subtype relation is decidable, and because the logical type system is syntax directed, we can conclude that type inference is decidable, too. The argument exactly follows the argumentation proving that type inference for the simply typed lambda calculus is decidable.

We will not follow this path any further because the approach of guessing the annotations does not lead to an algorithm that is useful in practice.

5.2 Complexity of Annotation Inference

In the implementation we cannot rely on some oracle guessing the correct assignments to all the constructor calls. The reason why that is not possible is easy, there exists too many different possibilities to assign abstract locations to constructors.

Let us consider a program with N new expressions. For such a program, the set of abstract locations is at most the interval $[1, \ldots, N]$. To find a tight lower bound for the possible assignments, we have to think about a clever way of picking abstract locations for the different new expressions.

Obviously, it does not make sense to choose any other abstract location than 1 for the first new expression. But for the next, we can consider the 1 another time, or we can pick the 2. For the third, we have either already picked $1, 1$ or $1, 2$. If we picked $1, 1$, it makes sense to choose from the set $\{1, 2\}$, if we have picked $1, 2$, we will choose from the set $\{1, 2, 3\}$. Hence, if we already have picked $x_1, \ldots, x_k$ values, we will choose x_{k+1} from the set $[1, \ldots, \max_{1 \leq i \leq k} x_i + 1]$.

A recursive formula $F(n)$ for the number of different possible assignments yields:

$$\begin{aligned}
F(n) &= f_n(n) \\
f_n(m) &= \sum_{i=1}^{n} \varphi_i(m) \\
\varphi_1(m) &= 1 \\
\varphi_n(m) &= \varphi_{n-1}(m-1) + n\varphi_n(m-1) && \text{if } 1 < m \wedge n \leq m \\
\varphi_n(m) &= 0 && \text{if } 1 < m < n
\end{aligned}$$

where the helper function $\varphi_n(m)$ takes two arguments and counts how many sequences with length m and maximum n do exist, if we have to choose the elements, such that no number is skipped. Because we are not allowed to skip numbers, we cannot create a sequence with m elements that does have a maximum greater than m. Hence, the last case, where $m < n$ yields a zero. The other base case, if the maximum we are interested in, is 1, yields a count of 1, since there exists only one sequence of length m with maximum 1, which is $\underbrace{1, \ldots, 1}_{m \text{ times}}$.

The recursive formula $\varphi_n(m)$ for $1 < m$ and $n \leq m$ reduces the computation to sequences of the length $m-1$. We can use two different kind of sequences of length $m-1$ and extend them to sequences of length m with maximum n. The first are the sequences with a maximum of $n-1$, the second the sequences with maximum n. If the maximum is smaller that $n-1$, we cannot reach a maximum of n by adding one number at the end, because we are not allowed to skip numbers. And if the maximum is greater than n, it will be obviously to large after the extension.

For a sequence with length $m-1$ and maximum $n-1$, we have to increase the maximum of the sequence by picking n. Due to the fact, that the number of sequences of length $m-1$ with maximum $m-1$ is $\varphi_{n-1}(m-1)$, we will get the same number of sequences.

The second possibility to create a sequence with a maximum of n and length m from a sequence of length $m-1$ is, we take all sequences that have already a maximum of n. In this case, we can extend each sequence by a number chosen from $[1, \ldots, n]$. For this case, $n \cdot \varphi_n(m-1)$ yields the number of possible sequences.

The nice thing with these two parts is, that they are disjoint. Hence, we can easily add up the two parts in our formula to compute $\varphi_n(m)$. See Table 5.1 for some computed values.

Will it be a practical approach to just try out all possibilities in an implementation? To answer this question we have to discuss how the function F is growing by presenting a closed function F', such that $F \in \Theta(F')$. But this complicated search is not necessary, since it is very easy to find a lower bound that is growing too fast for the brute force approach. Consider the following sequences (for an even n)

$$1, 2, 3, \ldots, \frac{n}{2}, x_1, \ldots, x_{\frac{n}{2}} \quad .$$

$F(n)$	$\varphi_n(m)$	m						
		1	2	3	4	5	6	7
1	$n = 1$	1	0	...				
2	2	1	1	0	...			
5	3	1	3	1	0	...		
15	4	1	7	6	1	0	...	
52	5	1	15	25	10	1	0	...
203	6	1	31	90	65	15	1	0
877	7	1	63	301	350	140	21	1

Table 5.1: Number of possible assignments of abstract locations to new expressions.

We will choose the x_i from the set $[1, \ldots, \frac{n}{2}]$. This yields sequences of length n with maximum $\frac{n}{2}$, which are all valid sequences with respect to the restriction discussed above. The number of sequences can be easily computed with the formula:

$$F'(n) = \left(\frac{n}{2}\right)^{\frac{n}{2}} \tag{5.2}$$

Obviously this is a lower bound we cannot accept in a setting of trying out all possible values, even if the number of new expression in a program is only around 1% of all the expressions.

Lemma 5.2.1. *For a program with n new expressions, the number of different possible annotations for these new expressions is $F(n)$.*

A lower bound for the recursively defined function $F(n)$ is $F'(n) = \left(\frac{n}{2}\right)^{\frac{n}{2}}$.

5.3 A Practical Approach

The last section motivates, why just guessing the abstract locations for new expressions is not an option. Hence, instead of ensuring that there does not exists a possible assignment for the abstract locations of new expressions, we will use some additional information available in JavaScript programs to make type inference applicable in practice.

There are two different possibilities to create objects in JavaScript.

First, by a new expression. A new expression in JavaScript calls a function and hence we annotate each function with an abstract label. For the new expression the annotation label of the function is used to determine the type of the created object. This approach groups objects with the same constructor into the same abstract location. Second, object literals are used to create objects. We abstract each object created by an object literal based on its allocation site. Hence, two objects created by different object literals are never abstracted to the same abstract location. An alternative approach for object literals might be to follow an idea used by Google Chrome and Firefox to group objects in order to make property read faster. They create classes for objects such that objects with the same properties are abstracted to the same class.

Both alternatives for object literals have drawbacks. In rare cases it might be the case that grouping objects into the same abstract location, just because they are created by the same constructor might reject a program that is type safe if the objects are grouped more accurate. Hence, we also allow the programmer to manually provide the abstract locations at each new expression and object literal if he needs the additional flexibility.

Consequentially, our type inference algorithm is not complete; that is, if the type inference algorithm rejects a program, it is not valid to conclude that for all possible annotations the program is rejected by the type checker.

5.4 Constraint Based System

Figure 5.1 presents the syntax of types, variables and constraints. Type variables (X) are mapped to types by type inference. During inference we map them to two types (indicated by $[\mapsto t, t]$). The first type is a lower bound, the second one is an upper bound (with respect to the subtyping relation). Location variables (μ) and precision variables (χ) are mapped to suitable values during constraint simplification. Object variables (ω) are assigned to maps from properties to type variables. The singleton environment variables (σ) are assigned to maps from abstract locations to object type variables. The need for the upper and lower bound for variables is explained in the section on constraint simplification (Section 5.4.2).

Types in the inference algorithm are from Type_i. Type_i is the subset of Type_c that contains only finite types (hence Type_i is the inductive interpretation of the type syntax). Together with the type variables this is sufficient to encode all regular types, similar to the approach followed in Section 4.6.2.

5.4.1 Constraint Generation

The following three judgments

$$\Gamma \vdash_{te} e : \tau \Rightarrow \mu, \sigma, \Gamma \mid C$$
$$\Gamma \vdash_{s} s : \tau \Rightarrow \mu, \sigma \mid C$$
$$\Gamma \vdash_{w} w : \tau \mid C$$

generate the constraints for type inference. They are defined in Figure 5.2 and Figure 5.3.

Typically, the constraint generation rules call the constraint generation algorithm for all of their parts, create some additional constraints and build a new constraint by conjunction. Some of the rules need to create new fresh variables, which is indicated for example by the notation "σ' fresh" in the rule C-Demote.

$$
\begin{array}{lll}
X & [\, \mapsto t, t \,] & \text{type variable} \\
\mu & [\, \mapsto L, L \,] & \text{location variable} \\
\chi & [\, \mapsto q, q \,] & \text{precision variable} \\
\omega & [\, \mapsto \mathsf{Prop} \xrightarrow{\mathit{fin}} X \,] & \text{object variable} \\
\sigma & [\, \mapsto \mathsf{Location} \xrightarrow{\mathit{fin}} \omega \,] & \text{local environment variable}
\end{array}
$$

$$
\begin{array}{llll}
\tau & ::= & t \mid X & t \in \mathsf{Type}_i \\
S & \in & \{\in, \notin\} & \text{basic set operations} \\
Y & \in & \{\tau, \mu, \chi, \omega, \sigma\} & \text{wild-card} \\
C & & & \text{constraints}
\end{array}
$$

$$
\begin{array}{lll}
C ::= & \tau <: \tau & \text{subtype} \\
& \sigma, \tau \lhd \tau & \text{flow on type} \\
& \Sigma, \sigma \rhd^{\mu} \sigma & \text{flow on environment} \\
& l \; S \; \mu & \text{set operation on location variable} \\
& \mu \subseteq \mu & \text{subset of location variable} \\
& \mu = \mu - \mu & \text{difference of location variable} \\
& \mathrm{Locs}(\tau) \subseteq \mu & \text{all pointers part of } \tau \text{ are in } \mu \\
& \mu \cap \mu = \emptyset & \text{disjoint sets} \\
& \sigma =^{\#}_{\mu} \sigma & \text{demotion on local environment variable} \\
& X =^{\#}_{\mu} X & \text{demotion on type variable} \\
& \omega =^{\#}_{\mu} \omega & \text{demotion on object variable} \\
& \sigma \vdash_{\mathfrak{r}} X.a : X & \text{property read} \\
& \sigma \vdash_{\mathfrak{w}} X.a := X \Rightarrow \sigma & \text{property write} \\
& Y = Y & \text{equality} \\
& C \wedge C & \text{conjunction} \\
& \overline{C_i} & \text{conjunction} \\
& \texttt{False} & \text{type error} \\
& \texttt{True} \text{ or } \emptyset & \text{success}
\end{array}
$$

Figure 5.1: Constraint syntax.

$$
\textsc{C-Undefined}\quad \frac{}{\Sigma, \Gamma \vdash_w \texttt{udf} : \texttt{udf} \mid \emptyset}
\qquad
\textsc{C-Object}\quad \frac{}{\Sigma, \Gamma \vdash_w (q\ell, i) : \texttt{obj}(q\ell) \mid \emptyset}
\qquad
\textsc{C-Variable}\quad \frac{x : X \in \Gamma}{\Sigma, \Gamma \vdash_w x : X \mid \emptyset}
$$

$$
\textsc{C-Value}_{te}\quad \frac{\Sigma, \Gamma \vdash_w w : \tau \mid C_1 \qquad C_2 = \tau <: \tau'}{\Sigma, \Gamma, \sigma \vdash_{te} w : \tau' \Rightarrow \emptyset, \sigma, \Gamma \mid \overline{C_i}}
\qquad
\textsc{C-Value}_{s}\quad \frac{\Sigma, \Gamma \vdash_w w : \tau \mid C_1 \qquad C_2 = \tau <: \tau'}{\Sigma, \Gamma, \sigma \vdash_{s} w : \tau' \Rightarrow \emptyset, \sigma \mid \overline{C_i}}
$$

Figure 5.2: Type rules for constraint generation.

Most of the rules are straightforward (Figure 5.2). Interesting is the rule C-Function. It creates a fresh type variable X, two location set variables μ and μ' and a local environment variable σ_2. Local environment variables carry a location set variable to encode the domain of the map. The domain location set variable of σ_2 is named μ_s, expressed by the notation $\mu_s \leftarrow \mathrm{dom}(\sigma_2)$. Due to the fact that $\mathcal{JSC}$ and $\mathcal{JSR}$ have static scoping, the set of free variables of the function $\mathtt{rec}\, f(x).e.$ is computed statically, which allows to compute Γ' from Γ by a simple domain restriction. Then, the constraint generation algorithm calls the algorithm for constraint generation for top expressions with Γ' and σ_2, yielding the new variables μ'', σ_1, a type environment Γ''' and a set of constraints C_5 and the type or type variable τ.

The three following constraints C_1 to C_3 express simple restrictions on the location set variables. The constraint C_4 is build from a set of constraints. For each $x \in \mathrm{dom}(\Gamma')$ a constraint is generated. These constraints are combined by a conjunction (expressed by $\wedge$ over the equation sign).

The correctness of the constraint generation is not proved formally, since we omit also formal definitions for the semantics of the constraints. We decide to omit them, because the semantic of all constraints is either clear, since we use basic set operations or logical operations, or the semantics of a constraint is easily defined by the corresponding relation from the logical type system.

5.4.2 Constraint Solving

Unfortunately, constraint simplification has to deal with negative information like $\ell \notin \mu$. Such a constraint is implicitly generated from $\sigma' = \sigma[\ell \mapsto \{\}]$[1] in the constraint generation for `new`, for example. As a consequence, using a monotone framework to solve the constraints does not work. Hence, we combine two monotone frameworks where one is collecting positive information like $\ell \in \mu$ and the other one is collecting the negative information $\ell \notin \mu$. The positive information raises the lower bounds for location variables (and other variables) and the negative information lowers their upper bounds. We obtain an initial upper bound for the location variables by typing the program under a closed world assumption.

Following the constraint generation phase, all constraints are inserted into a work list and subjected to simplification. Simplifying a constraint can cause one or more of the following actions:

1. Create a new constraint,
2. Raise the lower bound of a variable,
3. Lower the upper bound of a variable,
4. Remove the constraint from the constraint set.

[1] The map update requires $\ell \notin \mathrm{dom}(\sigma)$.

C-Function

$$\frac{\begin{array}{c} X, \mu, \mu', \sigma_2 \text{ fresh} \qquad \mu_s \leftarrow \mathrm{dom}(\sigma_2) \\ \Gamma' = \Gamma \downarrow \mathrm{fv}(\mathtt{rec}\, f(x).e) \qquad \Sigma, \Gamma'(x : X), \sigma_2 \vdash_{te} e : \tau \Rightarrow \mu'', \sigma_1, \Gamma''' \mid C_5 \\ C_1 = \mu_s \cap \mu = \emptyset \qquad C_2 = \mu' \subseteq \mu \qquad C_3 = \mu'' \subseteq \mu \\ \forall x \in \mathrm{dom}(\Gamma') : C_4 \triangleq \mathrm{Locs}(\Gamma'(x)) \subseteq \mu' \wedge \Gamma''(x) =^{\#}_{\mu'} \Gamma'(x) \wedge \sigma_1, \Gamma'(x) \lhd \Gamma''(x) \end{array}}{\Sigma, \Gamma \vdash_w \mathtt{rec}\, f(x).e : (\sigma_2, X) \xrightarrow{\mu} (\sigma_1, \tau) \mid \overline{C_i}}$$

C-Demote

$$\frac{\begin{array}{c} \sigma' \text{ fresh} \qquad C_1 = \Sigma, \sigma \rhd^{\mu} \sigma' \qquad \forall x : \mathrm{dom}(\Gamma) : C_2 \triangleq \Gamma'(x) =^{\#}_{\mu} \Gamma(x) \\ \Sigma, \sigma', \Gamma' \vdash_{te} e : \tau \Rightarrow \mu', \sigma'', \Gamma'' \mid C_3 \qquad C_4 = \mu \subseteq \mu' \qquad C_5 = \mu'' = \mu' - \mu \end{array}}{\Sigma, \sigma, \Gamma \vdash_{te} \natural^{\mu} e : \tau \Rightarrow \mu'', \sigma'', \Gamma'' \mid \overline{C_i}}$$

C-Let

$$\frac{\begin{array}{c} \mu_s \leftarrow \mathrm{dom}(\sigma) \qquad \Sigma, \sigma, \Gamma \vdash_s s_1 : \tau_1 \Rightarrow \mu_1, \sigma_1 \mid C_1 \qquad C_2 = \mu_s \cap \mu_1 = \emptyset \\ \Sigma, \sigma_1, \Gamma(x : \tau_1) \vdash_{te} e_2 : \tau_2 \Rightarrow \mu_2, \sigma_2, \Gamma'(x : \tau_1') \mid C_3 \qquad C_4 = \mu \subseteq \mu_1 \wedge \mu \subseteq \mu_2 \end{array}}{\Sigma, \sigma, \Gamma \vdash_{te} \mathtt{let}\ x = s_1\ \mathtt{in}\ e_2 : \tau_2 \Rightarrow \mu, \sigma_2, \Gamma' \mid \overline{C_i}}$$

C-Function Call

$$\frac{\begin{array}{c} \Sigma, \Gamma \vdash_w w_2 : \tau_2 \mid C_1 \\ \Sigma, \Gamma \vdash_w w_1 : (\sigma, \tau_2) \xrightarrow{\mu} (\sigma', \tau_1) \mid C_2 \end{array}}{\Sigma, \sigma, \Gamma \vdash_s w_1(w_2) : \tau_1 \Rightarrow \mu, \sigma' \mid \overline{C_i}}$$

C-New

$$\frac{C_1 = \ell \notin \mathrm{dom}(\sigma) \qquad C_2 = \sigma' = \sigma[\ell \mapsto \emptyset]}{\Sigma, \sigma, \Gamma \vdash_s \mathtt{new}^{\ell} : \mathtt{obj}(@\ell) \Rightarrow \{\ell\}, \sigma' \mid \overline{C_i}}$$

C-Read

$$\frac{\Sigma, \Gamma \vdash_w w : \tau \mid C_1 \qquad C_2 = \tau = \mathtt{obj}(p) \qquad C_3 = \sigma \vdash_{\mathfrak{r}} \tau.a : \tau'}{\Sigma, \sigma, \Gamma \vdash_s w.a : \tau' \Rightarrow \emptyset, \sigma \mid \overline{C_i}}$$

C-Write

$$\frac{\begin{array}{c} \Sigma, \Gamma \vdash_w w : \tau \mid C_1 \\ \Sigma, \Gamma \vdash_w w' : \tau' \mid C_2 \qquad C_3 = \tau = \mathtt{obj}(p) \qquad C_4 = \sigma \vdash_{\mathfrak{w}} \tau.a := \tau' \Rightarrow \sigma' \end{array}}{\Sigma, \sigma, \Gamma \vdash_s w.a := w' : \mathtt{udf} \Rightarrow \emptyset, \sigma' \mid \overline{C_i}}$$

Figure 5.3: Type rules for constraint generation.

For most cases the simplification rule is straightforward. Let us have a look at the constraint simplification rule for $l \in \mu$:

$$l \in \mu \qquad \rightarrow \qquad \texttt{delete}, \underline{\mu} := \underline{\mu} \cup \{l\}$$

Executing this rule raises the lower bound of μ (denoted by $\underline{\mu}$) by adding l. Afterwards $\underline{\mu}$ contains all the information that was expressed by the constraint, so the constraint itself is deleted.

The operation $\cup$ for a location variable does some additional things. First, it checks if l is not contained in the upper bound of μ, denoted $l \notin \overline{\mu}$. If that is the case, then the constraint $l \in \mu$ is not solvable because it contradicts the already computed upper bound. Second, if the lower bound of μ changes (viz., if $\underline{\mu}$ did not contain l before the update), then every constraint which depends on μ is added to the work list.

Another, more complicated example is the simplification of a read constraint. Let us assume we have

$$\sigma \vdash_{\mathfrak{r}} \xi\mu_1.a : \alpha$$

and that $\underline{\mu_1} = \{l_1\}$ and $\overline{\mu_1} = \{l_1, l_2, l_3\}$, the precision variable $\xi = @$, $\sigma(l_1) = o_1$, $o_1 = []$. The constraint simplification finds that ξ is equal to @ and that implies that only one location is allowed for μ_1. Because the lower bound of μ_1 contains one element, simplification sets $\overline{\mu_1} = \{l_1\}$. After changing $\overline{\mu_1}$ every constraint that depends on μ_1 is added to the work list. In particular, the read constraint is visited once more, because it depends on μ_1.

The next visit notices that μ is equal to a location, and that the precision variable is set. Hence, $\sigma(l)$ contains information about the shape of the object. Reading the property a of the object enforces that the object has a property that is a subtype of α. Hence, simplification extends $o_1 = [a \mapsto \alpha_a]$ and generates a new constraint $\alpha_a <: \alpha$. If one of these operations is not allowed the constraint `False` is generated, which terminates constraint simplification immediately and rejects the program. Of course, the newly generated subtype constraint is added to the work list, as well as each constraint that depends on o_1.

The algorithm considers many other cases. Discussing them does not lead to additional insights. Correctness of the transformation is typically straightforward to see. Hence, the only reason to present all simplification rules is to prove a completeness result for contraint simplification. But such a proof is not of much value in our situation. The inference algorithm is not complete because of the heuristic used to infer annotations for new expressions and object literals. Experience with our prototype implementation indicates that the combination of two monotone frameworks enables the algorithm to collect sufficient information to infer a valid typing.

Currently, the implementation is not optimized for speed. For example, it propagates the structure of the most recent heap to every program point. A possibility to make the algorithm faster by reducing the amount of data computed is to use lazy propagation [41, 65].

6 Extensions

This chapter discusses how to extend the formal system of Chapter 4 with expressions and statements of JavaScript that are not part of the calculus. Section 6.1 presents an extension to support functions with multiple parameters. Next, in Section 6.2, we present two approaches to include conditionals into the calculus. An aspect of JavaScript that is special to the language is its realization of inheritance. It turns out that the recency-aware calculus can easily be extended to support the prototype mechanism of JavaScript (cf. Section 6.3).

6.1 Multiple Parameters

Functions in $\mathcal{JSR}$ take only one parameter. To simulate functions with two parameters, the most common approach is to define a shortcut:

$$\lambda \mathsf{xy}.\mathtt{e} ::= \lambda \mathsf{x}.\lambda \mathsf{y}.\mathtt{e}$$

But this approach does not work well in our recency-aware calculus, because the calculus used a modified substitution. Consider the following example:

$\textbf{let}\ \mathsf{o} = \mathtt{new}^{l_1}\ \textbf{in}$
$\textbf{let}\ \mathsf{p} = \mathtt{new}^{l_2}\ \textbf{in}$
$\textbf{let}\ \mathsf{f} = \lambda \mathsf{xy}.\ \mathsf{x.a} := \mathsf{y}\ \textbf{in}$
$\quad \mathsf{f\ o\ p}$

In this example f has two parameters and the programmer assumes that passing precise object references to functions by parameters is allowed. But after applying the shortcut, the code extends to:

$\textbf{let}\ \mathsf{o} = \mathtt{new}^{l_1}\ \textbf{in}$
$\textbf{let}\ \mathsf{p} = \mathtt{new}^{l_2}\ \textbf{in}$
$\textbf{let}\ \mathsf{f} = \lambda \mathsf{x}.\ \lambda \mathsf{y}.\ \mathsf{x.a} := \mathsf{y}\ \textbf{in}$
$\quad \mathsf{f\ o\ p}$

Hence, x is a free variable in $\lambda \mathsf{y}.\ \mathsf{x.a} := \mathsf{y}$, and the modified substitution will replace x not by $(@l_1, 0)$ but with $(\tilde{\ }l_0, 0)$ during evaluation of the example. To avoid this problem we can introduce tuples in our system and pass multiple function parameters to the function as a tuple. We define a shortcut:

$$\lambda \overline{\mathtt{x}}.\mathtt{e} ::= \lambda \mathtt{t}.\mathtt{e}[\mathtt{x_i} \mapsto \mathtt{t_i}]$$

where $\overline{x} := x_1, \ldots, x_n$, t is a tuple and t_i is code to access the i-th component of t.

We can choose to simulate tuples by objects with specific property names for the different components. The benefit from that approach is that the extension of the static type system is just syntactic sugar. For example, the function type $(\Delta, (\texttt{int}, \texttt{int})) \to (\Delta, \texttt{int})$ is a shortcut for $(\Delta, \texttt{obj}(\tilde{\ell})) \to (\Delta, int)$ with $\Sigma(\ell) = (\texttt{fst} : \texttt{int})(\texttt{snd} : \texttt{int})$. The code to implement the lookup for the first component is $x.\texttt{fst}$.

6.2 Conditionals

6.2.1 Conditionals in $\mathcal{JSC}$

The integration of conditional expressions into $\mathcal{JSC}$ is simple. We extend the syntax from $\mathcal{JSC}$ (c.f. Figure 4.1) by a new expression:

$$s \quad ::= \quad \ldots \mid \texttt{if}\ w\ s\ s$$

The condition of the conditional expression is either a value or a variable. To support only variables and values as conditions is not a restriction, because a programmer can branch on the result of an arbitrary computation by using a let expression in front of the conditional. We extend substitution (Figure 4.3):

$$(\texttt{if}\ w\ s_1\ s_2)[x_i \mapsto v] = \texttt{if}\ w[x_i \mapsto v]\ s_1[x_i \mapsto v]\ s_2[x_i \mapsto v]$$

We also extend dynamic semantics (c.f. Figure 4.4) by the two rules:

$$\begin{array}{llll} \text{S0-I{\scriptsize F}T} & H, \texttt{if}\ v\ s_1\ s_2 & \to_0 & H, e_1 \text{ if } v \neq \texttt{udf} \\ \text{S0-I{\scriptsize F}F} & H, \texttt{if}\ \texttt{udf}\ s_1\ s_2 & \to_0 & H, s_2 \end{array}$$

They define that the conditional evaluates to either the “then” branch, or the “else” branch, depending on the value of the condition. If the value is a “falsy” value (in our calculus udf is the only “falsy” value), the “else” branch is chosen (S0-IfF), otherwise (S0-IfT) evaluation continues with the “then” branch. The heap does not change in both rules. If the calculus is also extended by other primitive values, for example boolean values or strings, a rule

$$\begin{array}{lllll} \text{S0-I{\scriptsize F}C} & H, \texttt{if}\ v\ s_1\ s_2 & \to_0 & H, \texttt{if}\ v'\ s_1\ s_2 & \text{if } v \text{ not a boolean value} \\ & & & & \text{and } v \to_0^b v' \end{array}$$

is used to convert the value v into a boolean value v' according to a conversion relation $\to_0^b$. In this case, the side conditions of S0-IfT and S0-IfF will change into $v = \texttt{true}$ and $v = \texttt{false}$, and it will hold for example that $\texttt{udf} \to_0^b \texttt{false}$. For the converting relation $\to_0^b$ it should hold that:

$$\forall x : x \to_0 y \quad \to \quad y = \texttt{true} \lor y = \texttt{false}$$

This is a way to model the automatic value conversion of JavaScript in the core calculus $\mathcal{JSC}$, while we do take the details of the conversion into account.

6.2.2 Conditionals in $\mathcal{JSR}$

We extend the syntax of $\mathcal{JSR}$ in the same way as in $\mathcal{JSC}$

$$s \quad ::= \quad \ldots \mid \texttt{if}\ w\ s\ s$$

and extend the dynamic semantics similarly:

$$\begin{array}{llcl} \text{S-I{\scriptsize F}T} & \mathcal{H}, \texttt{if}\ v\ s_1\ s_2 & \longrightarrow & \mathcal{H}, s_1 \text{ if } v \neq \texttt{udf} \\ \text{S-I{\scriptsize F}F} & \mathcal{H}, \texttt{if}\ \texttt{udf}\ s_1\ s_2 & \longrightarrow & \mathcal{H}, s_2 \end{array}$$

It is also necessary to define demotion

$$(\texttt{if}\ w\ s_1\ s_2)^{\natural L} := \texttt{if}\ w^{\natural L}\ s_1^{\natural L}\ s_2^{\natural L}$$

and substitution

$$(\texttt{if}\ w\ s_1\ s_2)\{x \Mapsto v\} = \texttt{if}\ w\{x \Mapsto v\}\ s_1\{x \Mapsto v\}\ s_2\{x \Mapsto v\}$$

for conditional expressions.

The most critical question is, how to define the type rule for the conditional. There is a large design space here, because the condition adds nodes to the control flow graph in such a way that there is a join of multiple edges. Care must be taken here to ensure that the important invariant INV_H is still valid. The easiest possibility to model the conditional is to be restrictive:

$$\text{T-I{\scriptsize F}R} \quad \frac{\Sigma, \Gamma \vdash_w w : t_v \qquad \forall i \in \{1,2\} : \Sigma, \Delta, \Gamma \vdash_s s_i : t_i \Rightarrow L, A, \Delta \qquad t_i <: t}{\Sigma, \Delta, \Gamma \vdash_{te} \texttt{if}\ w\ s_1\ s_2 : t \Rightarrow L, A, \Delta, \Gamma} \tag{6.1}$$

This rule ensures that the structure of the most recent heaps of the two branches are the same. Of course this yields a sound system, but maybe it is too restrictive. For example it forbids most recent objects to escape one branch of the conditional. It is possible to create new objects – even with different abstract locations – inside of the conditional branches, but they must be demoted before the conditional is finished.

As an example consider Program 6.1. In this program the two branches of the conditional create different objects, and the result is bound to the variable `x`. Due to the demote expressions the new objects are returned old, resulting in a situation that `x` has the type $\texttt{obj}(\tilde{}\{l_1, l_2\})$. If the programmer initializes the two objects independently from each other completely inside the conditional branches, the type rule T-IfR is perfectly fine and works as expected.

But what happens, if the program attaches a new method after the execution of the conditional? The method attachment will not pass the type checker, because changes to the shape of old objects are not supported by $\mathcal{JSR}$ and it is not possible to keep the object most recent after the conditional, if the rule T-IfR is used to type conditional expressions.

Program 6.1 $\mathcal{JSR}$ – conditionals.

```
let c = ⋯ in
$\natural^{\{\ell_1,\ell_2\}}$ let x =
  if c
    let tmp = new$^{\ell_1}$ in $\natural^{\ell_1}$ tmp
    let tmp = new$^{\ell_2}$ in $\natural^{\ell_2}$ tmp
in
  x
```

Thus, our goal is to find a solution that keeps the objects longer most recent[1], even if they are created in different conditional branches with eventually different shapes. A first consequence of removing the demote expressions is that the variable x needs a type that subsumes $\texttt{obj}(@\ell_1)$ and $\texttt{obj}(@\ell_2)$. In the current type language of $\mathcal{JSR}$ the only supertype that fulfills the requirement is $\top$. Since this results in a situation, in which the programmer is unable to use the variable sensibly, the type language of $\mathcal{JSR}$ needs to be extended by union types of most recent objects. Only with union types it makes sense to support the typing of conditional expressions in such a way that most recent objects may escape the subexpressions of the conditional expression. The type of the variable x is $\texttt{obj}(@\{l_1, l_2\})$ in such an extended system if the demote expressions are omitted.

6.2.2.1 Unit Types of Most Recent Objects

In the following paragraphs we create a system that supports the union of most recent objects. As a consequence, the example in the previous section (without the demote operations) will pass the type checker, and the type of the variable x is $\texttt{obj}(@\{\ell_1, \ell_2\})$.

The type syntax from Figure 4.14 is extended by

$$p \quad ::= \quad \cdots \mid @L$$

to express an union type over most recent objects. To support union types of most recent objects, please consider the inference rule

$$\begin{array}{l}\text{ST-O{\small BJ}}\\ \dfrac{L \subseteq L'}{\texttt{obj}(qL) <: \texttt{obj}(qL')}\end{array} \tag{6.2}$$

from Figure 4.16. Since the type syntax is extended to contain precise object types with location sets that are no singleton sets the rule becomes more powerful by expressing that $\texttt{obj}(@\{\ell_1\}) <: \texttt{obj}(@\{\ell_1, \ell_2\})$ holds. Hence, without any additional adjustments it seems now possible to type the variable x in the example with the

[1] Hence, there will be no demote expressions in the conditional branches.

precise union type $\texttt{obj}(@\{\ell_1, \ell_2\})$. But, at a second look, the type rule T-IfR is also restrictive with respect to the structure of the most recent heap. The rule demands for two equal most recent heaps for the two branches of the conditional expression.

6.2.2.2 Heap Alternatives

Even if union types over most recent objects allow to express that a variable may contain a most recent object of an abstract locations L, the type rule for a conditional expression needs additionally a way to handle the case that the structure of the most recent heap is different for the different parts of the conditional. Hence, a way to combine different most recent heap structures is needed. For this purpose we introduce heap alternatives. We extend the syntax of most recent heap types by:

$$\Delta \quad ::= \quad \cdots \mid \Delta \parallel \Delta$$

Heap alternatives express that the most recent heap may either have the structure provided by the left most recent environment, or given by the right most recent environment. With these two syntax extensions the type rule for conditionals

$$\text{T-If}\quad \frac{\begin{array}{c}\Sigma, \Gamma \vdash_w w : t_v \qquad \Delta_1, \Delta_2 \vdash_J \Delta' \\ \forall i (t_i <: t \wedge \Sigma, \Delta, \Gamma \vdash_s s_i : t_i \Rightarrow L_i, A_i, \Delta_i)\end{array}}{\Sigma, \Delta, \Gamma \vdash_{te} \texttt{if}\ w\ s_1\ s_2 : t \Rightarrow L_1 \cup L_2, A_1 \cup A_2, \Delta', \Gamma} \tag{6.3}$$

supports the escape of different most recent objects from the different branches of the conditional expression. If the two conditions $\Sigma, \Delta, \Gamma \vdash_{te} e_i : t_i \Rightarrow L_i, A_i, \Delta_i$ for $i \in \{1, 2\}$ are fulfilled, each branch of the conditional has an access effect A_i, an allocation effect L_i, a new most recent environment Δ_i and a return type t_i. The return type of the conditional is a super type of the type of the two branches, which is ensured by $t_i <: t$. The two effects are just combined by the union operation.

Combining the most recent heap environments is delegated to the relation $\Delta_1, \Delta_2 \vdash_J \Delta'$, which is defined in Figure 6.1. The first two rules states that the join of a environment and an empty environment is just the former. A more complicated situation arises if both environments contain objects. The next rule defines how to merge environments if one of them contains an object that the other one does not contain; that is, if the domain of the two environments is not equal. The objects that are part of one of the environments are not modified. They are collected inside the heap alternatives. The merge relation ensures that it holds for all $\Delta_1 \parallel \Delta_2$:

$$\Delta_1 \parallel \Delta_2 \rightarrow \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset \tag{6.4}$$

Objects for which both environments contains entries are treated by computing the smallest supertype for each property. That is done by the last rule of the merge relation.

$$\Delta, \emptyset \vdash_J \Delta \parallel \emptyset \qquad\qquad \emptyset, \Delta \vdash_J \Delta \parallel \emptyset$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta_1) \neq \mathrm{dom}(\Delta_2) \\ \mathrm{dom}(\Delta_1) \neq \emptyset \qquad \mathrm{dom}(\Delta_2) \neq \emptyset \qquad L = \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) \qquad \Delta_2' = \Delta_2 \downarrow L \\ \Delta_1' = \Delta_1 \downarrow L \qquad \Delta_1'' = \Delta_1 \uparrow L \qquad \Delta_2'' = \Delta_2 \uparrow L \qquad \Delta_1', \Delta_2' \vdash_J \Delta_M\end{array}}{\Delta_1, \Delta_2 \vdash_J \Delta_M(\Delta_1'' \parallel \Delta_2'')}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2) \neq \emptyset \qquad l \in \mathrm{dom}(\Delta_1) \\ \Delta_1 \uparrow \{l\}, \Delta_2 \uparrow \{l\} \vdash_J \Delta \qquad (\forall a \in \mathsf{Prop})\ (\Delta_2(l)(a) <: r(a)) \wedge (\Delta_1(l)(a) <: r(a))\end{array}}{\Delta_1, \Delta_2 \vdash_J \Delta(l : r)}$$

Figure 6.1: Joining the most recent heap.

$$\frac{\Sigma, \Delta \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta', 1}{\Sigma, \Delta \vdash_{\mathfrak{w}} @L.a := t \Rightarrow \Delta'}$$

$$\emptyset \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \emptyset, 0 \qquad\qquad \frac{\Delta \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta'', n \qquad \ell \in L}{\Delta(\ell : r) \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow (\ell : r[a \mapsto t])\Delta'', n + 1}$$

$$\frac{\Delta \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta, n \qquad \ell \notin L}{\Delta(\ell : r) \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta, n}$$

$$\frac{\Delta_1 \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta_1', n_1 \qquad \Delta_2 \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow \Delta_2', n_2}{(\Delta_1 \parallel \Delta_2) \vdash^{@}_{\mathfrak{w}} L.a := t \Rightarrow (\Delta_1' \parallel \Delta_2'), \max(n_1, n_2)}$$

Figure 6.2: Extension – writing properties with heap alternatives.

To handle union types over most recent objects sound, all auxiliary relations handle the heap alternatives specially, for example the environment $\Delta(\ell_1 : []) \parallel (\ell_2 : [])$ does only support the synchronous demotion of ℓ_1 and ℓ_2. It is not allowed to demote the ℓ_1 object without the ℓ_2 object, because there may be a union type of the form `obj`$(@\{\ell_1, \ell_2\})$. The following paragraphs will explain all the necessary adjustments to the auxiliary relations in detail.

The relation $\vdash_{\mathfrak{w}}$ takes the global heap environment Σ, the most recent environment Δ, a reference type $p = q\ell$, the property a and the type of the right hand side of the assignment t and relates them to a new most recent environment if the assignment is valid.

The original version of the relation is defined in Figure 4.25. There are two cases handled by the original relation. The first rule handles the case, where $p = \tilde{}L$.

A write access is only allowed if the type that should be stored inside the object property is a subtype of all the property types. Hence, the condition $t <: \Sigma(\ell)(a)$ is enforced for all $\ell \in L$. The second rule deals with the case of most recent objects. It supports strong updates, which means, it allows the write access and adjusts the type of the corresponding object type inside of Δ. In the base case, where L contains only one element and Δ is a mapping from abstract locations to heap types, the adjustment is straight forward.

Because of union types over precise objects and heap alternatives the relation is extended. An additional rule in Figure 6.2 delegates the work needed in such a situation to another relation $\vdash_{\mathfrak{w}}^{@}$, which supports strong updates for most recent objects. Because a strong update is only sound if for each heap alternative one object is changed, the relation computes the number of adjusted objects per alternative and returns is maximum. Since the heap alternatives ensure that only one of the alternatives is relevant during an execution, it is a valid operation to allow strong update on different objects if they exists in different alternatives. Hence, the rule dealing with the heap alternatives (the last one) computes the number of object changes using the formula $\max(n_1, n_2)$, while in the other rules the number is increased if necessary.

The relation $\vdash_{\mathfrak{w}}$ requires $n = 1$. This condition is essential for the soundness proof, because it allows to prove that the type change of a property write only effects one object, and that only one object description in Δ that has a corresponding object in the heap is changed. All the other changes are performed to heap alternatives that are dead in the sense that for their changed object descriptions there exists no object in the actual heap. Therefore, even with heap alternatives and union types over most recent objects, a one-to-one relation between abstractions and object can be established.

6.3 Prototypes

It is fairly straightforward to extend $\mathcal{JSR}$ with a JavaScript-style prototype mechanism. Figure 6.3 defines the calculus $\mathcal{JSR}'$ as an extension to syntax, operational semantics, and typing of $\mathcal{JSR}$. $\mathcal{JSR}'$ has an enhanced version of object creation, $\texttt{new}^{\ell}(v)$. Its reduction rule S-New′ initializes a reserved prototype property `_p` of the new object to v. This property must not be used in user code.

The read reduction rule S-RdI replaces the S-RdE rule in Figure 4.8. The read operation first examines the value v obtained by reading the property directly from the object itself. It returns v if $v \neq \texttt{udf}$. Otherwise, if a prototype is defined for the object, it delegates the lookup to the prototype. If the property is undefined or the prototype is not an object, the read operation returns `udf`.

The revised typing rule for `new` installs the prototype argument in the newly created object. The prototype argument is an arbitrary value. The rule T-Read′ ensures that if the property read follows a prototype chain, all accessed locations are added to A by using the relation $\Sigma, \Delta, A \vdash_a p, a$ instead of $A \vdash_a p$. The extended relation collects not just the one abstract location p, if p is a precise reference, but it also collects all most recent abstract locations used inside the prototype chain.

All other typing rules remain the same, but the auxiliary judgment to read a property needs to be revised. It mimics the operational semantics in descending the prototype chain of the object, returning the value when the property is found, and recursively reading the prototype if no value exists.

Writing of a property is not affected by prototypes, because the write operation only affects the top-level object and ignores the prototype chain [32].

Additional syntax

$$e ::= \cdots \mid \mathtt{new}^{\ell}(v)$$

Additional reductions

$$\begin{array}{llcl}
\text{S-N}{\small\text{EW}}' & H, H_0, \mathtt{new}^{\ell}(v) & \longrightarrow & H, H_0[(\ell, j) \mapsto \{\mathtt{\_p} \mapsto v\}], (@\ell, j) \\
 & & & \text{if } \mathrm{dom}(H_0) \cap \{\ell\} \times \mathbb{N} = \emptyset \\
 & & & \text{and } (\ell, j) \notin \mathrm{dom}(H) \\
\text{S-R}{\small\text{D}}\text{I} & H, H_0, (p, i).a & \longrightarrow & H, H_0, \mathit{read}(H, H_0, (p, i), a)
\end{array}$$

$$\mathit{read}(H, H_0, (p, i), a) = \begin{cases} v & \text{if } v \neq \mathtt{udf} \\ (q, i).a & \text{if } v = \mathtt{udf} \wedge \mathit{pt} = (q, i) \\ \mathtt{udf} & \text{otherwise} \end{cases}$$

$$\begin{array}{lrl}
\text{where} & \mathit{pt} &= (H, H_0)(p, i)\$\mathtt{\_p} \\
 & v &= (H, H_0)(p, i)\$a
\end{array}$$

Changes in the static semantics

T-New'

$$\dfrac{\Delta, \Gamma \vdash_c \ell \qquad \ell \in \mathrm{dom}(\Sigma) \qquad \Sigma, \Delta, \Gamma \vdash_w v : t}{\Sigma, \Delta, \Gamma \vdash_s \mathtt{new}^{\ell}(v) : \mathtt{obj}(@\ell) \Rightarrow \{\ell\}, \Delta(\ell \mapsto \{\mathtt{\_p} \mapsto t\}), \Gamma}$$

T-Read'

$$\dfrac{\Sigma, \Gamma \vdash_w w : \mathtt{obj}(p) \qquad \Sigma, \Delta \vdash_{\mathfrak{r}} p.a : t \qquad \Sigma, \Delta, A \vdash_a p, a}{\Sigma, \Delta, \Gamma \vdash_s w.a : t \Rightarrow \emptyset, A, \Delta}$$

$$\dfrac{\begin{array}{c} t <: t' \\ (\forall \ell \in L)\ \Sigma, \Delta \vdash_{\mathfrak{r}} {}^{\sim}\ell.a : t \end{array}}{\Sigma, \Delta \vdash_{\mathfrak{r}} {}^{\sim}L.a : t'} \qquad \dfrac{(\Sigma, \Delta)(p)(a) = t \neq \mathtt{udf}}{\Sigma, \Delta \vdash_{\mathfrak{r}} p.a : t}$$

$$\dfrac{\begin{array}{c} (\Sigma, \Delta)(p)(a) = \mathtt{udf} \\ (\Sigma, \Delta)(p)(\mathtt{\_p}) = \mathtt{obj}(q) \qquad \Sigma, \Delta \vdash_{\mathfrak{r}} q.a : t \end{array}}{\Sigma, \Delta \vdash_{\mathfrak{r}} p.a : t} \qquad \dfrac{\begin{array}{c} (\Sigma, \Delta)(p)(a) = \mathtt{udf} \\ (\Sigma, \Delta)(p)(\mathtt{\_p}) \neq \mathtt{obj}(p') \end{array}}{\Sigma, \Delta \vdash_{\mathfrak{r}} p.a : \mathtt{udf}}$$

$$\dfrac{A \vdash_a p \qquad (\Sigma, \Delta)(p)(\mathtt{\_p}) = \mathtt{obj}(q) \qquad \Sigma, \Delta, A \vdash_a q, a}{\Sigma, \Delta, A \vdash_a p, a}$$

$$\dfrac{\begin{array}{c} A \vdash_a p \\ (\Sigma, \Delta)(p)(\mathtt{\_p}) = \mathtt{obj}(q) \qquad \Sigma, \Delta, A \vdash_a q, a \end{array}}{\Sigma, \Delta, A \vdash_a p, a} \qquad \dfrac{\begin{array}{c} A \vdash_a p \\ (\Sigma, \Delta)(p)(\mathtt{\_p}) = \mathtt{udf} \end{array}}{\Sigma, \Delta, A \vdash_a p, a}$$

Figure 6.3: Extension to support prototypes.

7 Related Work

Jones and Muchnick [68] first use allocation points for abstracting heap structures and the per-program-point approximation of the heap. They utilize the information for program optimization, for example to find a more efficient storage allocation scheme for a LISP-like programming language.

Chase and coworkers [18] invented the notation of strong updates. Their analysis relies on complex rules involving a per-program-point “storage shape graph”, and no correctness argument is given. This dissertation reduces the storage shape information to the (per-program-point) singleton environments, and it provides a soundness proof.

Our type system is related to must-alias analysis [3] and uniqueness typing [16]. Our summary type for old objects $\mathtt{obj}(\tilde{}L)$ expresses may-alias information. Our type for most recent objects expresses that all variables of this type refer to the same object. Uniqueness on the other hand guarantees that some variable holds the only reference to a heap object. Object names from the system of Altucher and Landi [3] also refer to most recent allocated objects. Hence, we can repsond positive to a question raised in previous work [64]: Their initial approach can be extended to a higher-order language.

Balakrishnan and Reps [9] present an analysis called “Recency-Abstraction for Heap-Allocated Storage”. They aim to optimize dynamic dispatch in C++ binaries to static function calls where possible. They formalize and implement their system based on abstract interpretation. Contrary, our work is based on a type system and we present a constraint-based inference algorithm.

In their work “Alias Types” Smith, Walker and Morrisett [111] design alias types as an extension of linar types for typing low-level languages. In contrast to our recency aware calculus, their type system is designed for consumption by a machine, not by a human. Their calculus models object initialization with type changing assignments to heap records. Alias types separate pointer types from the actual store contents. A pointer has a singleton type $\mathtt{ptr}(l)$, where l stands for a store location.

The work “Alias Types for Recursive Data Structures” [117] covers also the treatment of recursive data structures, for example the type system is capable of encoding cyclic and doubly-linked lists and trees. Their system relies on existential quantification to specify recursive data structures. Explicit pack and unpack operations are needed for existentials and for recursive types. In contrast, our precise pointer type $\mathtt{obj}(@\ell)$ is a singleton type standing for a location *at a particular program point.* Our $\mathtt{obj}(\tilde{}L)$ type has an existential-type flavor and it can model recursive data structures because we support types of infinite height due to the co-inductive interpretation of the type syntax. Our demote operation corresponds to packing. Furthermore, the alias types system is a prescriptive, explicitly typed calculus with decidable type checking, whereas our calculus is descriptive, implicitly typed, and has type inference.

Cqual [41] is a tool for specifying and inferring flow-sensitive type qualifiers in C programs, which is related to typestate inference. Cqual at first performs a flow-insensitive type, alias, and effect analysis. Then, it infers linearities, which it uses to perform strong updates on assigned type qualifiers. Cqual differs in a number of technical aspects from our work, one point being that our store abstraction can always handle one reference per abstract store location exactly (linearly in Cqual terminology) as well as many summary reference at the same time whereas Cqual classifies a store location as either linear or non-linear. Cqual's use of polymorphism is similar to our notion of store splitting, which is explained in Chapter 6.

Fähndrich and Xia's delayed types [35] also provide a means of treating object initialization. An object with a delayed type does not have to fulfill its invariants, yet. The example in their paper is type checking not-null types. Our calculus could be put to use for a similar analysis; at present, with delayed types the programmer provides an explicit boundary when the invariants must hold, whereas our calculus tracks exact information about an object as long as possible and reverts to less precise summary information when unavoidable.

Richards and coworkers [107] state that after an initialization phase the shape of objects changes in a lot of JavaScript programs. They define the initialization phase of objects created by a constructor function as the time that is spend in the constructor. Their definition of the object initialization phase differs from the initialization phase of a recency aware type system. In such a system the initialization phase is the time until the next constructor call with the same abstract location. Hence, the type system can keep the object much longer in its initialization phase than the work of Richards et. al. assumes. A consequence is that their data [107] does not imply the non-existence of an initialization phase in JavaScript programs. We expect that the recency attribute holds sufficiently long to cover the initialization phase, but further investigation is required to confirm this expectation.

Liang and coworkers [77] measure how good an abstraction based on recency is compared to other alternative, as for example k-CFA. Their main result is that recency offers the best tradeoff between precision and size. Of course care must be taken to blindly transport their results to our target language, because they mainly analyze Java programs, while our type system has the intention to fit JavaScripts needs.

Kehrt and Aldrich [70] explore an imperative variant of Abadi and Cardelli's object calculus with delegation, linear object types, and linear methods. As long as objects have a linear type, their method suite and delegatee can be changed as typical in an initialization phase. Later on, the programmer can drop linearity of an object at the price of making it immutable. Recency can achieve similar objectives without requiring the object to be linear and without making it immutable. The object loses its special status only when the next object is allocated at the same abstract location.

Anderson and coworkers [4] define a flow-sensitive type system for JavaScript. They present a type inference algorithm for their type system. It extends record types by a precision indicator on each record component. The latter distinguish whether a record component is definitively initialized or whether it may be uninitialized. Hence, the indicator separates types that contain a undefined component from those that do not contain an undefined component. Compared to our work, they do not model type change and they do not consider prototypes. Our implementation can type check all examples in that work. We have not been able to get meaningful results from their implementation, which precludes further comparison.

A similar idea is the basis for Qi and Myer's masked types [102]. Their typestate-based system tracks the initialization of objects in a Java core language. A mask type $C\backslash f$ describes an instance of class C that contains a field f which might not be initialized. The type system of masked types ensures that a partially initialized object is not accepted if a fully initialized object is expected. For the initialization of cyclic data structures the work presents conditionally masked types. For such cyclic data structures the system rely on annotations. Our system handles cyclic data structures without annotations.

In his work "Towards a Type System for Analyzing JavaScript Programs" [114] Thiemann proposed a type system for JavaScript that focuses on the detection of "undesirable conversions". We present an example for such a conversion in Section 2.3. The work is complementary to this thesis, because the type system that models type conversions is not flow-sensitive.

Jensen and coworkers [66] have built a static analyzer based on abstract interpretation for most parts of the JavaScript language. This system is based on recency, context sensitivity, and some other techniques to obtain precise results. That work is complementary to ours because it is a practical implementation, which is not supported by a formal proof. Moreover, it suffers from the restriction of all abstract interpretation-based systems that it only affirms that a particular program does not misbehave on a given set of inputs. Thus, unlike our present system, it cannot compute a function type that describes the set of admissible inputs.

To no surprise all static works do not support dynamic features. For example the `eval` function is not supported in a satisfactory way. The same problems arises with the with-statement.

Part II

JSConTest – A Dynamic Tool Based on Type Contracts and Access Permission Contracts

8 A Tour of JSConTest

Increasing the quality of software is a challenging task, especially if it concerns software written in JavaScript. JavaScript is a language that seems easily understandable in the beginning, but which has a lot of sharp edges that make it a challenge to reason about.

In the first part of this work the static approach tackles some problems that arise during day to day work with the language, and presents a static type system, that is able to solve some of them. The system provides some excellent guarantees, for example it ensures that no null pointer exception happens during the execution of a program that passes the type checker.

The static approach has on the other hand some drawbacks, too. One problem is that the type signatures of functions are typically not easy to understand, especially for people who are not used to static type systems. Even if the inference of these signatures can minimize this problem dramatically, the type system has to confront the programmer with complicated error messages. It is a long way to go until the static system from the first part will be accepted by the community.

Even more importantly, if a project realized in JavaScript does not need a guarantee of the strength of a powerful type system, the situation gets worse. For example for a large number of online games created in JavaScript it is more important to apply an agile development process than investing a lot of time and money to write games in a fashion the recency aware type system requires. In such a situation the investment in a static type system will not pay off.

This part of the dissertation therefore concentrates on utilities for JavaScript projects that are less dependent on exact guarantees for their programs. In these projects, there is also a need for a better tool support for software development. An approach targeting these projects has to make different assumptions than the approach presented in the first part, and it therefore has another focus. In this setting, it is important that the system is simple to use for JavaScript developers, easy to understand, and easy to integrate with the existing program infrastructure.

This part of the dissertation presents JSConTest. It is a tool that implements a contract system for JavaScript with the following features:

- It is based on familiar concepts.
- The system is applicable step by step to already existing projects.
- The system supports the specification of complex properties.
- The system is independent from the browser. It supports all target platforms of the programming language JavaScript.

The main idea of JSConTest is that simple type signatures, similar to the ones used in programming languages provide important information about the functions and methods they are describing. Often the type signature – together with the function name – is sufficient information to understand what the purpose of the function is. JavaScript lacks native facilities to specify type signatures like the one JSConTest offers, and we believe that using these signatures makes the task of finding bugs in a JavaScript program much easier.

Instead of proving the correctness of the type signature statically, JSConTest tries to find counterexamples for these type signatures by automatic random testing and run-time monitoring. This approach gives up the guarantied promisses of the static approach, but it grants some useful properties in return.

No Restriction in Style The most important benefit of JSConTest is that the testing approach does not reject programs because they are written in a style that does not fit the static type system. So a user of JSConTest never has to tackle the limitations of a static type system. He can write his program the way he wants, not the way the type system is telling him.

Direct Feedback Loop If a contract is violated either by a manual test, by a randomly generated test case, or because run-time monitoring detects a contract violation during a program execution in the wild[1], the programmer has an execution at his hand to start debugging from.

Gradual Applicability Like other contract-based systems, JSConTest can be applied gradually. Starting from a few operations with contracts, it is easy to add contracts step by step thus gradually specifying larger parts of a program.

8.1 Type Signatures as Contracts

We introduce JSConTest with some JavaScript examples. The function isShipmentValid is a predicate which checks whether the object passed as first parameter is an object representing a valid order in an online shop.

```
/*c obj → bool */
function isShipmentValid(o) {
  if (x.name && x.address && x.totalprice) {
    return (x.totalprice > 20 || x.shipment > 0);
  }
  return false;
}
```

The function returns a boolean value indicating that the object is valid (true), or invalid (false). To this end, it checks the existence of the name and the address

[1] or during the execution of a manual test suite

and if the total price of the order is already computed. If that is the case, the next check to perform is, if the total price of the order is larger than 20, or, if that is not the case, if the shipment charges are larger than 0.

In JSConTest, the programmer can specify the interface of a function or method by attaching a special comment to the function. For example, the comment /*c obj → bool */ defines that the interface of the function isShipmentValid takes an object as a parameter and returns a boolean value. These contracts are similar to type signatures, well known in functional programming languages like ML [82], Haskell [98], OCaml [91], etc.

Of course, the programmer is not limited to such simple type signatures, as mentioned above. An important aspect of JavaScript is that functions are first-class values. JavaScript offers function expressions which are useful to create anonymous functions that should perform a task later on, for example if it is called as an event handler for a keystroke. Consider the anonymous function (saved in the variable f) for a contract that is attached to a function expression.

```
var f = function(x,y) /*c (int,int) → bool */ {
  return (x != y && 2 * x == x + 10);
}
var g = function (y) /*c (int) → (int → bool) */{
  return function(x) /*c int → bool */ {
    return x === y;
  };
};
```

The contract of the function is a simple type signature which states that the function is a predicate over two integer values. The type signature of the function g is a little bit more complicated. The function takes an integer value and returns a function of type int → bool. Of cause, JSConTest also supports function contracts in argument position. An example using this facility is the sort function, which takes an order and an array of integers.

```
/*c ((int, int) → bool, [int]) → [int] */
function sort(leq, xl) {
  // some sorting function using leq to compare two array entries
}
```

JSConTest offers a suitable contract for each JavaScript type.[2] JavaScript's object system is flexible and expressive. Hence, the contract system of JSConTest requires a flexible and expressive way to phrase, what kind of objects a function takes as parameter. For this purpose JSConTest supports composite contracts. A composite contract of the form $\{\ p_1 : c_1,\ ...,\ p_n : c_n\ (,\ldots)^?\}$ describes an object with at least properties p_1 to p_n. The contracts c_1 to c_n describe the types of the properties and the optional extension $(,\ldots)^?$ states that the random generator for the object may randomly add properties to the object. Since the check method for

[2] Some examples: string, number.

Program 8.1 JavaScript – Format a bill.

```
/*c { items : [ { name : string, price : number, count : int } ],
     name : string,
     address : string
    } → string */
function formatBill( o ) {
  var s = "", i = 0, sum = 0, t = 0;
  if ( o.name && o.address && o.items) {
    s += o.name + "</br>";
    s += o.address + "</br>";
    for (i = 0; i < o.items.length; ++i) {
      t = o.items[i].price * o.items[i].count;
      s += o.items[i].count + " x " + o.items[i].name;
      s += + "(each: " o.items[i].price") = " + t + "</br>";
      sum += t;
    }
    s += "sum: " + sum + "</br>”;
  }
}
```

the object only ensures that the properties p_1 to p_n exist with a correct contract, adding additional properties does not affect the checker of the object contract in any way. To emphasize that the object must not have any additional properties, JSConTest offers the notation $\{|\ p_i : c_i\ |\}$. The check procedure[3] for such a contract ensures that the object does not have a property p with $p \notin \cup_i\{p_i\}$. Another composite contract is $[c]$, which matches arrays, where the entries of the array matches c. Hence, arrays are considered to be homogeneous in JSConTest, which is no restriction due to the existence of a contract that matches all values named top.

The function formatBill (Program 8.1) collects information about a bill and creates a string representation. In JavaScript it is often the case that instead of passing multiple parameters to a function, an object with properties is passed to simulate named parameters. The function formatBill takes 'three named parameters' (items, name and address), the list of items with their prices, names and counts as an array, the name of the customer and the shipment address. It returns a string value.

Instead of continuing to introduce JSConTest's contract language with examples, the full syntax definition is presented in Figure 8.1. For primitive data types, JSConTest supports also singleton contracts, which accept only a single value. Singleton contracts for floats, integers, strings and booleans are specified by writing

[3] The check function of a contract is a predicate decides for an arbitrary value if it is an instance of the contract. For more details consider Section 8.1.1

JavaScript primitives

$x \in$ identifier, $f \in$ float, $i \in$ integer, $s \in$ string, $b \in$ bool,
$r \in$ regular expression, $prop \in$ property

Primitive contracts

p	$::=$	`undf` $\mid$ `top`	undefined, any value
	$\mid$	`bool` $\mid$ b	boolean values
	$\mid$	`string` $\mid$ s	string values
	$\mid$	`int` $\mid$ i $\mid$ `[`i`;`i`]`	integers, integer intervals
	$\mid$	`number` $\mid$ f $\mid$ `[`f`;`f`]`	floats, float intervals
	$\mid$	`obj` $\mid$ `fun`	object, function
	$\mid$	`js:`x	custom contract, JS scope

Composite contracts

c	$::=$	p	
	$\mid$	c`@numbers` $\mid$ c`@strings` $\mid$ c`@labels`	guided random testing
	$\mid$	$(d, \ldots, d)\ (\rightarrow \mid \Rightarrow)\ d\ (ap)^?$	functions, $ap \rightarrow$ *Figure* 8.2
	$\mid$	$c.(c, \ldots, c) \rightarrow c\ (ap)^?$	methods, $ap \rightarrow$ *Figure* 8.2
	$\mid$	$\{p_1 : c_1, p_2 : c_2, \ldots, p_n : c_n(, \ldots)^?\}$	objects
	$\mid$	$\{\mid p_1 : c_1, p_2 : c_2, \ldots, p_n : c_n \mid\}$	objects
	$\mid$	`[`c`]`	arrays

Annotations

a	$::=$	`~noAsserts` $\mid$ `~noTests` $\mid$ `#Tests:`i

Dependent contracts

d	$::=$	c $\mid$ c`($`i`, ..., $`i`)` $\mid$ `id($`i`)`

Top-level contracts (embedded in JavaScript comments)

t	$::=$	`/*c` $c\ a^*$ (`|` $c\ a^*)^*$ `*/`

Figure 8.1: Syntax of contracts.

the corresponding literal value. There is also support for intervals of numbers and integers with the syntax [f; f] or [i; i]. For instance, [0; 1.1] is the float interval between 0 and 1.1, inclusively. Another primitive contract is a contract of the form js:x for some JavaScript identifier x. This notation supports custom contracts written by the user in JavaScript. This can be done easily by writing an object, that implements two functions, check and generate.[4]

JSConTest supports not just one contract per function. It is possible to write a list of contracts to a function by separating the individual contracts by an |. If a function is attached by a list of contracts, it has to fulfill all of them.

[4] Consider Section 11.2.2 for more details.

8.1.1 Requirements on Contracts

Consider for example a function f with a contract int → int. To check this contract, JSConTest generates a random integer value. After that, it calls f with this value and checks if the return value of the call matches the return contract, which amounts to checking if the return value is an integer.

The automatic random testing approach to (in)validate the contracts requieres the ability to generate values for all contracts that may be used in parameter position of a function contract. For all contracts that are used in result position of a function contract the system relies on a checking operation for a contract and a value. A consequence of the functional style of JavaScript is that function contracts may be used in parameter and result position of function contracts. This imposes the need to automatically generate functions that fulfill a given function contract and to check if a function is a valid instance of a function contract.

In general, since all contracts may be used in both positions, JSConTest requires each contract to provide a random generator and an instance check. The reason for this requirement can be summarized in the following table:

A → B check	requires	random generator for A
	requires	instance check for B
A → B generate	requires	random generator for B
	requires	instance check for A

8.2 Guided Random Testing

Often it is useful to modify the random generator of a contract depending on the situation, in which it is used. JSConTest offers *guided random testing* to achieve a modification of the random generator depending on the source code of a function. It is available for integers (@numbers), objects (@labels) and strings (@strings).

Attaching @number to the contract int modifies the random generator of the integer contract such that it depends on the constants of the function body the contract is attached to. Once again consider the function f (page 123) as an example. Let us assume a programmer has written the function f as follows.

```
1 /*c (int,int) → bool */
2 function f(x,y) {
3   if (x != y) {
4     if (2 * x == x + 10) {
5       return "true"; // contract violation
6     }
7   }
8   return false;
9 }
```

In line 5, the return statement does not return the boolean value true, but the string value "true", which means the contract of the function is not valid. If

JSConTest now tests the contract for the function, it will most likely not reject the contract, even if it is invalid, because the probability to reach line 5, which will spot the error, is extremely small ($\approx 2^{-32}$). The reason is the use of an uniformly distributed random generator for x and y, and that the condition in line 4 requires x to be 10.[5] JSConTest offers an annotation for integer or number contracts of the form @numbers, which will increase the probability to find a counterexample to $p \approx \frac{1}{16}$.

```
/*c (int@numbers,int@numbers,int@numbers) → bool */
function fut_1(x,y,z) {
  if ((x*3+5 == y*5+4) && (x*2−1 == z*9 − 1))
    return "true";
  return false;
};
```

The difference to feedback directed testing is, that the values generated by the random generator does not depend on further runs of the generator [94–96]. Hence, there is no need to add additional state to the random generator.

Instead of using the uniformly distributed random generator for integers, JSConTest will do a simple static analysis of the body of the function f. In this body JSConTest finds two numbers $(2, 10)$. Based on these numbers, JSConTest generates integer values either by using the random generator or by generating an expression tree (both cases with a probability of 0.5). The nodes in the expression trees are picked randomly and correspond to the basic arithmetic operations (+,-,*,/). The leafs are picked from the set of collected numbers $\{0, 1, 2, 10\}$ or from randomly generated integers (each case with a probability of 0.5). Whether a leaf or a node is picked by the algorithm is decided randomly.[6]

This rather simple approach seems to be tailored to this example. But it turns out to work in many other situations, too. Here is an example.

```
/*c (int@numbers,int@numbers,int@numbers) → bool */
function fut_1(x,y,z) {
  if ((x*3+5 == y*5+4) && (x*2−1 == z*9 − 1))
    return "true";
  return false;
};
```

The contract violation is guarded by a diophantine equation, which is difficult to solve.[7] The approach to generate expression trees randomly finds a solution to the equation in seconds.

[5] Ignoring the condition in line 3 for this approximation is possible due to the fact, that the probability of fulfilling the condition is $1 - 2^{-32} \approx 1$.

[6] The initial probability to generate a node or a leaf is 0.5. The probability to generate a leaf is raised by each level of the tree by the random generator to ensure termination. For the depth i the probability to generate a node is $p(node, i) = 0.5 * 0.9^i$.

[7] In general the task of solving a diophantine equation is not decidable.

For example, the tree $1 * 10$ is generated with probability

$$P(\text{"}1 * 10\text{"}) = \underbrace{\frac{1}{4}}_{node} * \underbrace{\frac{1}{4}}_{*} * \underbrace{\frac{1}{4}}_{leaf(l)} * \underbrace{\frac{1}{8}}_{1} * \underbrace{\frac{1}{4}}_{leaf(r)} * \underbrace{\frac{1}{8}}_{10} = \frac{1}{16384} \quad ,$$

where the factor $\frac{1}{4}$ is the probability to generate a tree with a node ($\frac{1}{2}$ for the tree, and another $\frac{1}{2}$ for choosing a node). The same holds for picking a leaf ($\frac{1}{2}$ to pick a tree, and $\frac{1}{2}$ for choosing a leaf). If a leaf is generated, two decisions need to be made. First, a value from the set of collected numbers ($\frac{1}{2}$) has to be picked, and second, the correct one has to be chosen ($\frac{1}{n}$, where n is the number of items in the set of collected numbers). For the example the set of collected numbers is $\{0, 1, 2, 10\}$. 2 and 10 are members of the set, because they are present in the program source. 0 and 1 are always added to the set of numbers, because they are important input values for all programs.

There is also a version of guided random testing for strings and objects. Attaching @strings to a contract that generates string values – typically to string – starts a static analysis searching for string constants in the function body. The strings are used as an input value for string parameters.

The annotation @labels modifies the random generator for objects. JSConTest implements it by collecting all property names from the function body. Since objects in JavaScript are so special, this feature is discussed in more detail further with another example. Finding the error in the following example using a usual random generator for objects that does not have knowledge about the function is hopeless.

```
/*c obj@labels → bool */
function h(x) {
  if (x && x.p && x.quest) {
    return "true"; // violation
  }
  return false;
}
```

But the annotation raises the probability to generate an object with properties p and quest, because these two properties are contained in the function body of h. JSConTest typically finds the defect in less than 10 test cases for this example.

8.3 Monitoring

Every time a function calls another function, and does not satisfy the requirement the callee imposes, the call itself is the reason for a contract violation. Program 8.2 contains a function g, which calls the function f. The function f expects an integer parameter. The expression x * "3O" converts the string value "3O" into the float value NaN. Hence, NaN is passed to the function f, which is a value that does

Program 8.2 JavaScript – Catching an Error in a String Literal.

```
/*c int → int */
function f(x) {
  return 2 * x;
}

/*c (int,int) → bool */
function g(x,y) {
  return (f(x * "3O") == 60); // error
}
```

not fulfill the contract int. Contract monitoring therefore detects the error in the string constant.

Contract monitoring is a feature of JSConTest which is useful in different situations. It is possible to monitor all contracts during the execution of a manual test suite, which make sense due to the fact that a manual test suite typically contains a set of tests providing a high coverage. During random testing of the contracts JSConTest activates monitoring to detect software defects that violates the contract of called functions.

8.4 Dependent Contracts

Function contracts in JSConTest may contain various kind of dependencies. The parameters of a function may depend on each other.

```
/*c (int,int) → bool */
function comp(x,y) {
  if ((x == y) && (x < 10) && (x > 1)) return "true";
  return false;
}
```

The problem with this example is, that only when the same integer value is picked for x and y, the defect of the function creates an error. Talking about input values, where two parameters of the same type do have the same value is an important task, and hence it makes sense to automatically use function contracts that do test these cases with a high probability. It is enough to attach /*c (int,id($1)) ⇒ bool */ to comp to guide the random generator in this direction.

If it is necessary to express more complex dependencies, JSConTest offers a notation to express that a parameter may depend on another one by using $c(\$i_1, \ldots, \$i_n)$, where the is are the numbers of the parameters on which c depends. For example, the contract /*c (int, id($1)) → true */ expresses, that if the comparison operator gets two integer values, which are identical, it should return

$$
\begin{array}{rcll}
ap & ::= & \texttt{with}\ L\ \texttt{except}\ L & \text{access permission contracts} \\
L & ::= & \texttt{[}P^{+}_{,}\texttt{]} \mid \texttt{js:}x & \text{access path language} \\
P & ::= & x\,\texttt{.}\,(Pr)^{*}_{.} \mid r & \text{access path expression} \\
Pr & ::= & prop \mid r \mid \texttt{?} \mid \texttt{*} \mid Pr* & \text{property, property set}
\end{array}
$$

Figure 8.2: Access permission contracts syntax. For an X, X^*_s stands for all finite lists with separator s and node element X of arbitrary length. X^+_s encodes all finite lists of minimum length 1.

the boolean value true. In general the parameters of a function may depend on each other as long as the dependencies between the parameters are acyclic.

8.5 Access Permission Contracts

So far, all features of JSConTest concentrate on a value-oriented functional specification of the behavior of functions. However, in JavaScript a lot of functions not only have a value oriented functional effect, but they also perform side effects. JSConTest supports access permission contracts to specify, what kind of side effects a function performs. For example consider the function redirect.

```
/*c (string) → undf with [window.location] */
function redirect(url) {
  window.location = url;
}
```

The function takes a string parameter and saves it to the property location of the window object. This operation causes a redirect of the page. Granting all functions, especially third party libraries, the right to redirect the page is not welcome. Access permission contracts are a way to inform the contract system of JSConTest that a function may only manipulate or read a certain part of the heap. An access permission contract starts with a variable name, followed by a list of property names, which specify what part is readable and writeable. Figure 8.2 defines the syntax of access contracts.

Often a situation arises, where the programmer would like to express that a function may change any property of an object, except one. As an example of such a situation consider a binary tree. A node in the tree is implemented by an object, that contains three properties left, right and value, where left and right contains objects representing the subtrees and value contains some value associated with the tree node. A method that computes balance information for all nodes of the tree may store the information inside the object structure of the tree in a property balance for later reuse. The balance of the node is defined as the difference between the height of the left subtree and the height of the right subtree. The method only

Program 8.3 JavaScript – Balance of a Binary Tree.

```
/*c js:tree.() → int with [this./left|right/*.balance] */
function computeBalance() {
  var lb = 0, // hight of the left subtree
    rb = 0;  // hight of the right subtree

  if (this.left) {
    lb = computeBalance.call(this.left);
  }
  if (this.right) {
    rb = computeBalance.call(this.right);
  }
  this.balance = lb − rb;
  return max(lb, rb) + 1;
}
```

writes the balance property, while it reads the properties left and right. A viable access contract for a method might be with [this./left|right/*.balance]. The regular expression /left|right/ expresses the two alternative property names for accessing one of the child nodes. Since the method has to traverse the tree to arbitrary depth, the star after the regular expression grants access to a sequence of left and right properties with an arbitrary length. Hence for example the path this.left.right.balance is granted by the access contract. An access permission contract provides write access to all paths that completely match the access permission contract. But it also grants read access to all paths that are prefixes to a path described by the contract. Hence, in the example, reading the paths this, this.left and this.left.right is granted by the access permission contract.

Now, consider the more complicated example Program 8.4, in which a function should clean up the tree by deleting all temporary intermediate values. Such a method may be useful, if the tree, without the intermediate values, should be transferred to the server via an AJAX request. This function needs write access to all properties, except the properties left and right. Consider the access permission contract with [this./left|right/*.?] except [this./left|right/*.value?.←]. Its first part provides arbitrary write access to all properties in the whole tree, while the second part restricts the permitted path such that the method is not allowed to write to the properties left, right and value. The question-mark behind the property name value makes it optional.[8] Hence, the method may delete all of the intermediate values stored in the tree, but it cannot change the structure of the tree or change the value of the nodes. The modifier ← indicates that only the write permission

[8] Hence, the * stands for zero or more times and ? stands for at most one occurrence. Do not mix it with the question-mark, that is used to encode an arbitrary property name. For example, the path x.bla.?? matches x.bla.p and x.bla.

Program 8.4 JavaScript – Clean a Binary Tree.

```
/*c js:tree.() → undf with [this./left|right/*.?]
                except [this./left|right/*.value?.←] */
function cleanUp() {
  for (var property in this) {
    if (this.hasOwnProperty(property) {
      if (property !== 'left' && property !== 'right') {
        if (property !== 'value') {
          delete this[property];
        }
      } else {
        cleanUp.call(this[property]);
      }
    }
  }
}
```

should be restricted. For more details about the way that except works, refer to the formal definition of access permission contracts (Chapter 9).

Even without having a formal introduction for the access permission contracts at hand right now, two properties of access permission contracts can be defined.

There is no point to include a path $\pi\pi'$ in the set of readable paths, if the path π is not readable. If access to π is restricted, the program will never be able to access any values through the path $\pi\pi'$. Hence, a property for access permission contracts, which we call *prefix closedness*, is, that for all readable path, all prefixes of all readable paths are readable. The same argument holds for write access permission contracts. If an access permission contract grants write access to the path $\pi\pi'$, it must also grant read access to π. In the formal specification of the access permission contracts, these two properties are defined formally in the next chapter.

9 Access Permission Contracts

The tour of JSConTest introduces access permission contracts with simple examples. The meaning of the access permission contracts from the examples is simple. The first impression after discussion some example might be, that the creation of an access permission contract system is simple. But it turns out that the design space it huge, and that there exist many pitfalls. A formalization spots these and is also a basement for a robust implementation. The interesting cases for a formalization are not the common cases, but the exceptional ones. If functions are called with aliases, or if the heap contains cycles, it is not always intuitive to decide, what the access permission contract should grant. We first state six design principles and discuss each of them in a separate section. Of course, the design principles for the access contracts should not collide with the properties JSConTest offers. Especially that JSConTest is partially applicable in software projects is important in this discussion.

But before stating the design principles, we clarify our hypothesis about what the mental model of a typical JavaScript programmer might be.

A JavaScript programmer always has to think about his program with respect to a heap containing all the objects the program works with. The structure of the heap that the programmer has in mind can be expressed by a graph of objects. Each object is a vertex in the graph. A vertex contains a table of property names. For each property, either a primitive value or a reference to another object is stored. References to other objects are edges to these objects. Figure 9.1 presents a graphical representation of an example heap. It contains four objects, X, Y, Z and A. The object A has no properties, the object X has two properties, z and p. The property z contains a reference to the object Z, which is graphically represented by an edge to the object. In this graph, the five places that may be affected by an assignment are marked @i for $i \in \{1, \ldots, 6\}$.[1]

But in JavaScript side effects occur in two flavors. The first side effect is performed by an assignment of the form o.p = e. It writes the property p of the object, which is bound to the variable o at the time the expression is executed. But there is also the assignment of a variable (x = e), which is at first sight something completely different, because it changes the binding of the variable instead of the value of a property.

Hence, to discuss side effects our graphical representation needs not only the heap, but also the variable assignment. Figure 9.2 extends the graphical representation with the variable assignments. In this graph, the scope table is also an

[1] We name the places using the pattern @i to distinguish them from access paths easily. An alternative name for the place @1 is Z.x or, if the address of the object Z is ℓ_z, maybe ℓ_z.x.

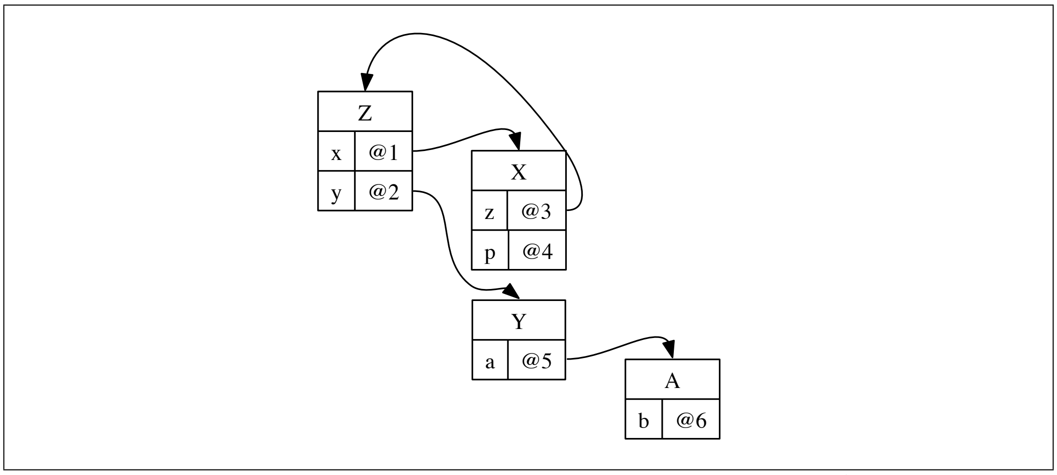


Figure 9.1: Heap graph with objects X, Y, Z and A.

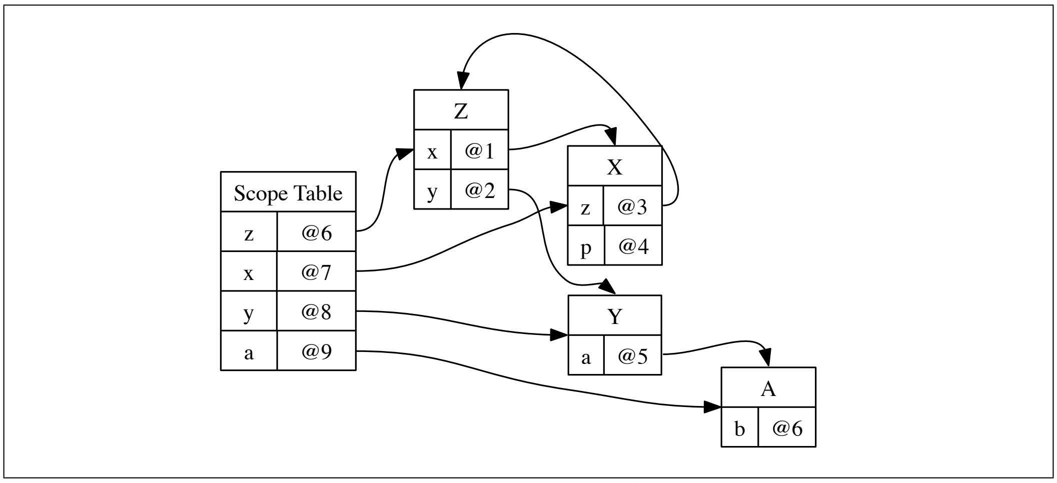


Figure 9.2: Heap graph with objects X, Y, Z, A, and with variables x, y, z, a.

object, with property names equal to the names of the variables. This graphical representation makes clear that there is not really a difference between a property assignment and a variable assignment. Both are just an instance of the same operation – a property assignment. This corresponds to the language specification [59], in which a variable assignment is just a property assignment of a scope object.

An access path starts with a variable name, followed by a possibly empty list of property names. It is easy to understand that using this notation every place that might be affected by a side effect is describable. For example the access path z.x is equal to the place @1, and the access path x.z.y and z.y is equal to @2 in the heap presented in Figure 9.2.

9.1 Design Principles

We continue by stating the basic design principles of access contracts, which we believe results in an intuitive, decidable and fast to implement system that fits well in JSConTest.

Access Granted by Default Principle When no access permission contract controls a read or write access, the access is granted. This principle provides a fallback, when no access permission contract is in force.

Path-Based Principle The permission to access a property of an object depends on the path taken to reach the object. Hence, an access permission contract grants a program the permission to traverse the heap.

Pre-State Snapshot Principle An access permission contract only extends to objects and paths in the heap at the time the contract is installed.

Last Writer Wins Principle The last write operation to a property determines the access rights for the descendants of the property.

Dynamic Extend Principle The lifetime of an access permission contract has dynamic extend analogous to the life time of a stack frame.

9.1.1 Access Granted by Default Principle

> When no access permission contract controls a read or write access, the access is granted.

The access granted by default principle defines precisely what happens, when the source code of a program is partially annotated with access contracts. Consider the function f:

```
// f behaves as f1
function f(x) {
  return x.a;
}
/*c top → top with [x.*] */
function f1(x) {
  return x.a;
}
/*c top → top with [] */
function f2(x) {
  return x.a;
}
```

There are two possibilities for the access permission contract system. Either, the function f behaves like f1. Or, it behaves like f2. In general, by default the access permission contract system grants access to all properties of all parameters and free variables for functions without an attached access permission contract. An access permission contract system following this approach implements the access granted by default principle.

If f behaves like f2, the access permission contract system rejects access to all properties of parameters and free variables for functions without an attached access permission contract.

9.1.2 Path-Based Principle

> The permission to access a property of an object depends on the path taken to reach the object. Hence, an access permission contract grants a program the permission to traverse the heap.

The following code illustrates the behavior of a path-based access contract system.

```
/*c obj → top with [x.a.o, a.b] */
function f(x) {
  return x.b.o;
}
var o = {};
var x = { a : {}, b {} };
x.a.o = o;
x.b.o = o;
f(x);
```

At the point where the function f is called the heap contains four objects, the one bound to x, the one bound to o, the one stored in the property a of x, and the one stored in the property b of x. The property o of a and of b points to the object o. Therefore, the heap contains two paths from the object x to the object o. The function f has an access permission contract that grants the right to traverse the heap via the object a, but in the function body the path via the object b is used. An access permission contract system based on the path-based principle reports a violation for the example.

An alternative design is to interpret the access permission contract location-based. In such a system it is not important which path is used to reach an object.

9.1.3 Pre-state Snapshot Principle

> An access permission contract only extends to objects and paths in the heap at the time the contract is installed.

The pre-state snapshot design principle fixes the heap that is used to interpret the traversal rights. The only reasonable choices are either the heap at the time when the access permission contract is installed, or the heap at the time when the read or write access is evaluated.

Let us assume a programmer attaches an access permission contract with [x.a, y.a, y.a.secret] to a function. He expects, that this contract protects a property secret of x.a, because it is not included in the access permission contract. With this intention in mind, consider the following example:

```
1  /*c ({}, {}) → any with [x.a, y.a, y.a.secret] */
2  function b(x, y) {
3    var tmp = y.a;
4    y.a = x.a;
5    y.a.secret = 42;        // allowed?
6    y.a = tmp;
7  }
```

If the access permission contract system grants a read or write operation with respect to the heap at the time when the read or write operatrion is evaluated, the write access to y.a.secret in Line 5 is granted due to the access permission contract y.a.secret. This behavior is undesired because from the point of view of the caller of b the function changes the property of the object x.a.secret.

The unintuitive behavior also occurs the other way round:

```
8  /*c ({}, {}) → any with [x.a, y.a, y.a.secret] */
9  function b(x, y) {
10   var tmp = x.a;
11   x.a = y.a;
12   x.a.secret = 42;        // allowed?
13   x.a = tmp;
14 }
```

Here, in Line 12, the write access to x.a.secret writes to the property secret of the object stored at function invocation time in x.a. The access permission contract system rejects the access if it used the heap at the time, when the read or write operation is evaluated.

The pre-state snapshot principle ensures that tricking access contracts by introducing new aliases is not possible once the contract is installed, because all access paths are evaluated with respect to the heap at the time at which the access permission contract was installed.

9.1.4 Last Writer Wins Principle

> The last write operation to a property determines the access rights for the descendants of the property.

A consequence of the pre-state snapshot principle is that the access contract system has to keep track of newly introduces aliases. Aliases which are created before an access permission contract are not considered. Here is an example.

```
1  /*c ({}) → any with [x.a,x.b.a] */
```

```
2  function l(x) {
3    x.a = x.b;
4    x.a.a = 42;
5  }
6  function l1() {
7    var x = { a: {}, b: {}};
8    l(x);
9  }
```

In this code fragment Line 3 is clearly permitted as x.a may be assigned to and x.b may be read. The following read access to x.a in Line 4 returns the reference to the object that was accessible through x.b, when the permission was installed. As this object was first reached via x.b, the access permission contract for x.b counts so that the assignment to x.a.a is sanctioned by the path x.b.a. Thus, function l1() runs without violation!

If we modify the example to create the alias *before* installing the permission, then things look different.

```
10 /*c obj → any with [x.a,x.b.a] */
11 function m(x) {
12   var y = x.a;
13   y.a = 42;      // violation
14 }
15 function m1() {
16   var x = { a: {}, b: {}};
17   x.a = x.b;
18   m(x);
19 }
```

In this case, running m1() yields a violation. While the first read access to x.a in Line 13 is sanctioned by x.a, the write access to property a of this object is not. Indeed, this behavior is consistent with invoking m on an object without any aliasing, which reports a violation under any semantics.

9.1.5 Dynamic Extent Principle

An access permission contract has dynamic extent.

If a programmer attaches an access permission contract to a function, his intention is to express, what side effects the function performs. The access permission contract system regards side effects as property reads and property writes. Hence, we have to define which property reads and writes are related to a function. As a first attempt we say, that every property read or property write, that happens *during* the function execution is related to the function.

We define now what we esteem as *during* the function execution. For this discussion it is important that the system under design is a dynamic system. Therefore, *during* does *not* denote the simple syntax property *inside the function body.* The access permission contract system observes only property reads and writes that the virtual machine executes. There are three different possibilities to define what we esteem to be *during* the function execution:

- minimal extent
- dynamic extent
- contagious extent

To illustrate minimal extent consider the following example:

```
1  function d(x) {
2    if (true) {
3      return x.a;        // reads property a of x
4    } else {
5      return x.b;        // never executed, not considered
6    }
7  }
```

Obviously, the read of property a inside of d happens during the execution of d. For the minimal extent principle, we define *during* function execution as the set of all property reads that happen between the call of the function and its return, for which there is a suitable expression inside the function body. Consider the following example:

```
8  /*c obj → any with [x.a] */
9  function d1(x) {
10   return x.a;          // violation if called from d?
11 }
12 /*c obj → any with [] */
13 function d(x) {
14   return d1(x);
15 }
```

An access permission contract system with minimal extent does not consider the property read in Line 10 as during the execution of d because the expression that performs the property read is not part of the body of d. Hence, an access contract system based on the minimal extent principle checks at most the properties that a static system checks, if it analyses the function body without taken care of function calls.

Next, we present the definition for the dynamic extent principle. It extends the set of property reads and writes from the minimal extent principle with all property reads and writes that happen during the execution of called functions. Hence, because the property read in Line 10 is during the execution of d1 (even in the minimal extent principle), it also happens during the execution of d, because d calls d1.

Based on the dynamic extent principle, a closure returned from a function is not affected by the access permission contracts in charge during the time, when the closure is created. But it is restricted with the access contracts that are in effect at the time, when the closure is called. For example, consider the permission of f in the following code fragment:

```
/*c obj → (() → any) with [x.b] */
function f(x) {
  return function() { return x.a + x.b; };
}
var o = { a: "secret", b: "public" };
var r = f(o);
r();
```

The example does not yield a contract access violation because the access to x.a happens outside of the dynamic extent of the call f(o). The read access which happens in the last line does not lead to a contract violation, because there is no access contract in force. Therefore, the access is allowed due to the access granted by default principle.

A slightly modified version provokes an access permission contract violation.

```
function g(x) {
  return function() { return x.a + x.b; };
}
/*c obj → any with [x.b] */
function g1(o) {
  var r = g(o);
  r();          // violation
}
```

As before, creating the closure does not lead to a contract violation, but during the execution of r, g1's access contract with [a.b] is in force. Therefore, the access x.a yields a contract violation.

An access permission contract system that is based on the contagious extent principle extends the set of property reads and writes by the property accesses that happens during the execution of closure created inside of the function.

9.1.6 Reference Attachment Principle

> Permissions are attached to individual references, not to heap locations.

To understand the reference attachment principle, think about a function, which is called with aliased parameters. In such a situation the access permission contract system can either attach the access permission contract to objects or to references. Both decisions lead to a reasonable access contract system. We prefer the option to attach the access permission contracts to the references and not to the objects. To understand the difference consider the following example.

```
/*c (obj, obj) → any with [x.b,x.a] */
function h(x, y) {
  y.a = 1;
  y.b = 2;            // violation
}
function h1() {
  var o = { a: −1, b: −2 };
  h(o, o);
}
```

In this example the function h is called by the expression h(o,o). Hence, the function's two parameters both refer to the same object. If access permission contracts are attached to objects, the access y.b is not distinguishable from the access x.b inside of the function h.

9.2 Core Language

In this section we present a formalization of a core calculus. It is a lambda calculus with imperative objects and a special expression to register access permission contracts. It is based on the design principles from Section 9.1.

Definition 9.2.1 (Access Permission Contract)**.** *Let Prop be a set of properties. An access permission contract $(L_r, L_w) \in 2^{Prop^*} \times 2^{Prop^*}$ is a pair of decidable languages of property lists. An access permission contract has to fulfill the following two properties:*[2]

Prefix Closedness *A pair $(L_r, L_w) \in 2^{Prop^*} \times 2^{Prop^*}$ is prefix closed, if for all paths $p \in L_r$, with $p = p_1, \ldots, p_n, p_i \in Prop$ it holds, that $p_1, \ldots, p_m \in L_r$ for all $m \in \{0, \ldots, n\}$.*

Prefix Accessibility *A pair $(L_r, L_w) \in 2^{Prop^*} \times 2^{Prop^*}$ is prefix closed, if for all paths $p \in L_w$, with $p = p_1, \ldots, p_n, p_i \in Prop$ it holds, that $p_1, \ldots, p_m \in L_r$ for all $m \in \{0, \ldots, n-1\}$.*

Figure 9.3 specifies the syntax of $\mathcal{JSE}$, a call-by-value lambda calculus with objects, similar to other core calculus used to model JavaScript [50, 56]. Property read, property write and object creation work analogously to the constructs from $\mathcal{JSC}$ (Section 4.1). The construct `permit` $x : L_r, L_w$ `in` e supports the specification of access permission contracts. The access permission contract L_r, L_w restricts access to the heap during the evaluation of e. If a violation during the evaluation of e occurs, the evaluation stopps.

[2] The only hard requirement of the formal system is the decidability of the word problem of the two languages of the access permission contract.

$$\begin{array}{lcll}
\text{variable} & x & \in & \mathit{Var} \\
\text{property name} & p & \in & \mathit{Prop} \\
\text{access path} & \pi & \in & \mathit{Path} = \mathit{Prop}^* \\
\text{path language} & L & \in & \mathit{PLang} = 2^{\mathit{Path}} \\
\text{expression} & e & ::= & x \mid \lambda x.e \mid e(e) \mid \texttt{new} \mid e.p \mid e.p := e \\
 & & \mid & \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e
\end{array}$$

Figure 9.3: $\mathcal{JSE}$ – syntax.

A difference between $\mathcal{JSE}$ and $\mathcal{JSC}$ is that the semantics of $\mathcal{JSE}$ is specified by a big-step evaluation judgment of the form,

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; v \qquad ,$$

while $\mathcal{JSC}$ uses a small step operational semantics. The judgment is defined inductively in Figure 9.4. It specifies, how the expression e transforms the initial heap H to the final heap H' under a given variable environment ρ and indexed collections $\mathcal{R}$ and $\mathcal{W}$ of read and write permissions. It also defines the return value (v) of the execution. The evaluation uses time stamps $u \in$ Stamp for multiple purposes. The rule for property read uses the time stamp to implement the last writer wins principle. Therefore, the semantics increases the time stamp for each property write. Time stamps are also used to identify the access contracts. Hence, additionally to the property read rule, the permit expression increases the time stamp. All other rules just pass through the time stamp unchanged.

References in the calculus are special. This is a direct consequence of the reference attachment principle – the calculus stores all the information about the already traversed access path inside each reference. Hence, a reference contains a pointer and also a map of indexed access paths. The index map identifies which access path corresponds to which access permission contract. The calculus uses time stamps for this purpose.

Figure 9.5 defines which expressions are values, what indexed permissions are and some other sets and mappings formally. A value is either a reference to an object or a closure – a pair of a variable environment and an expression. The heap is a finite map from locations to object. Objects are finite maps from property names to pairs of time stamps and values. The time stamp component of the pair saves the time at which the last write access to the property happened. Indexed permissions are maps from time stamps to languages over property paths.

The evaluation rules for variables (VAR), lambda abstraction (LAM) and function application (APP) are standard. They pass through the time stamp. The rule APP propagates the indexed permission maps $\mathcal{R}$ and $\mathcal{W}$ unchanged to their subevaluations.

The evaluation rule NEW creates a new object in the heap and returns a reference to the newly created object. The object has no properties. Therefore, the map added to the heap under the new reference ℓ is empty. Due to the reference

VAR

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; x \hookrightarrow H; u; \rho(x)$$

LAM

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \lambda x.e \hookrightarrow H; u; (\rho \downarrow_{\mathrm{fv}(\lambda x.e)}, \lambda x.e)$$

APP

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \hookrightarrow H''; u''; v_1 \qquad \rho'[x \mapsto v_1], \mathcal{R}, \mathcal{W} \vdash H''; u''; e \hookrightarrow H'''; u'''; v\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \hookrightarrow H'''; u'''; v}$$

NEW

$$\frac{\ell \notin \mathrm{dom}(H)}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{new} \hookrightarrow H[\ell \mapsto \emptyset]; u; (\ell, \emptyset)}$$

GET

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \vdash_{\mathrm{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow H'; u'; \mathcal{M}.p \circledvee H'(\ell)(p)}$$

PUT

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \\ \mathcal{W} \vdash_{\mathrm{chk}} \mathcal{M}.p \qquad H''' = H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \hookrightarrow H'''; u'' + 1; v}$$

PERMIT

$$\frac{\begin{array}{c}\rho', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H; u + 1; e \hookrightarrow H'; u'; v \\ \rho' = \rho[x \mapsto \rho(x) \lhd [u \mapsto \varepsilon]]\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e \hookrightarrow H'; u'; v}$$

Figure 9.4: $\mathcal{JSE}$ – Semantics.

attachment principle the access path to this object is not stored inside the object, but inside the reference, which is the result of the new expression. The map of indexed access paths of the reference is empty, because the reference is not accessible with respect to all heaps for which access permission contracts are already registered (due to the pre-state snapshot principle). For that reason, the new object is completely unrestricted until a permit operation installs new restrictions for that reference. The rule NEW does not change the time stamp, because no object in the heap is modified.

The rule GET defines the read operation of object properties. It relies on some auxiliary operations defined in Figure 9.7. After computing the location ℓ and the collection $\mathcal{M}$ of already traversed access paths of the reference, the premise $\mathcal{R} \vdash_{\mathrm{chk}} \mathcal{M}.p$ checks the read permission for these paths extended with property p. This check is specified by the rule CHECK PERMISSION (Figure 9.6), which requires that for each currently active index u, the access path for u is contained in the set of permitted access paths for u.

The rule CHECK PERMISSION implements the access by default principle by requir-

$$\begin{array}{rrcl}
\ell \in & \mathrm{Loc} & & \text{countable set of locations} \\
u \in & \mathrm{Stamp} & & \text{time stamps} \\
H \in & \mathrm{Heaps} & = & \mathrm{Loc} \xrightarrow{\mathit{fin}} \mathit{Obj} \\
o \in & \mathit{Obj} & = & \mathit{Prop} \xrightarrow{\mathit{fin}} \mathrm{Stamp} \times \mathit{Val} \\
\mathcal{P}, \mathcal{R}, \mathcal{W} \in & & & \mathrm{Stamp} \xrightarrow{\mathit{fin}} \mathit{PLang} \\
\mathcal{M}, \mathcal{N} \in & \mathit{PMap} & = & \mathrm{Stamp} \xrightarrow{\mathit{fin}} \mathit{Path} \\
(\ell, m) \in & \mathit{Ref} & = & \mathrm{Loc} \times \mathit{PMap} \\
v \in & \mathit{Val} & = & \mathit{Ref} + \mathit{Env} \times \mathit{Expr} \\
\rho \in & \mathit{Env} & = & \mathit{Var} \xrightarrow{\mathit{fin}} \mathit{Val}
\end{array}$$

Figure 9.5: $\mathcal{JSE}$ – sets.

CHECK PERMISSION

$$\frac{\forall u \in \mathrm{dom}(\mathcal{P}) \cap \mathrm{dom}(\mathcal{M}) : \mathcal{M}(u) \in \mathcal{P}(u)}{\mathcal{P} \vdash_{\mathrm{chk}} \mathcal{M}}$$

Figure 9.6: $\mathcal{JSE}$ – checking permissions.

ing $\mathcal{M}(u) \in \mathcal{P}(u)$ for the time stamps, which are in the domain of both mappings. If the rule requires the inclusion for all time stamps in the domain of $\mathcal{P}$, the formal system will implement a reject by default principle.

A get operation also returns a value from the heap. If the return value is a reference, this reference needs an appropriate indexed property path map. If the value is no reference, this map is not needed. The function $\olessthan$ decides if such a property map is needed. If the map is needed, $\olessthan$ uses another auxiliary function $\olessthan_u$ to compute the appropriate map. The two functions are defined in Figure 9.7.

A possible new access path to the reference is $\mathcal{M}.p$, which is the first parameter of the $\olessthan$ operation. The second parameter is the return value of the property lookup. It contains a time stamp u, which states when the property was written and a value v. The function $\olessthan$ checks, if v is a reference. If that is not the case, there is no need to compute a map of indexed access paths. Hence, the function returns the value v. If v is a reference, the map of access paths returned by the read operation is computed by the operator $\olessthan_u$. The parameter u of the operator is the time stamp from the heap. The purpose of $\olessthan_u$ is to implement the last writer wins principle.

In the application $\mathcal{M} \olessthan_u \mathcal{N}$, the first argument $\mathcal{M}$ contains the newly discovered access paths (that is the $\mathcal{M}.p$ of the GET rule). The second argument $\mathcal{N}$ contains the access paths from the heap. The subscript u is the time stamp used to model the last write wins principle. The definition in Figure 9.7 distinguishes three cases depending on when the property was last written and where the written value came from. The examples in Section 9.3 illustrate these three cases. Let u' be the

$$\mathcal{M}' \olessthan (u,v) \quad := \quad \begin{cases} (\ell, \mathcal{M}' \olessthan_u \mathcal{N}) & \text{if } v = (\ell, \mathcal{N}) \\ v & \text{if } v \notin \mathit{Ref} \end{cases}$$

$$(\mathcal{M} \olessthan_u \mathcal{N})(u') \quad := \quad \begin{cases} \mathcal{N}(u') & \text{if } u' \in \mathrm{dom}(\mathcal{N}) \\ \mathcal{M}(u') & \text{if } u' \in \mathrm{dom}(\mathcal{M}) \backslash \mathrm{dom}(\mathcal{N}) \wedge u < u' \\ \text{undefined} & \text{if } u' \in \mathrm{dom}(\mathcal{M}) \backslash \mathrm{dom}(\mathcal{N}) \wedge u \geq u' \\ \text{undefined} & \text{if } u' \notin \mathrm{dom}(\mathcal{M}) \cup \mathrm{dom}(\mathcal{N}) \end{cases}$$

$$(\mathcal{M}.p)(u) \quad := \quad \begin{cases} \mathcal{M}(u).p & \text{if } u \in \mathrm{dom}(\mathcal{M}) \\ \text{undefined} & \text{if } u \notin \mathrm{dom}(\mathcal{M}) \end{cases}$$

Figure 9.7: $\mathcal{JSE}$ – auxiliary definitions. We split the third case of the override operation $\olessthan_u$ into two cases to make the definition easier to read.

serial number of an execution of a permit expression.

1. The object's property value already has an access path for index u' (in $\mathcal{N}$). In that case the property has been overwritten since the introduction of u' and the existing access path is kept as it reflects an access path at the time when the permission u' was created. Preferring elements from $\mathcal{N}$ over elements from $\mathcal{M}$ models the last writer wins design principle.

2. The object's property value has no access path for index u' in $\mathcal{N}$. It has been written before the permission with index u' was installed as can be seen from $u < u'$. Therefore, the reference was accessible in the heap at the time the permission was installed by the path $\mathcal{M}(u')$. Hence, we attach the path $\mathcal{M}(u')$ to the value.

3. There is no access path for index u' in $\mathcal{N}$ and the property has been written after the permission of index u' was installed (viz. $u \geq u'$)[3]. This property was not linked to the data structure, when u' was created. The access by default principle defines that in this situation the access is granted. Therefore, no entry is attached to the value. Together with rule CHECK PERMISSION, which only checks the inclusion for all $u \in \mathrm{dom}(\mathcal{P}) \cap \mathrm{dom}(\mathcal{M})$, no violation is reported with respect to the access permission contract registered under unique id u'.

   For the case that both index maps $\mathcal{M}$ and $\mathcal{N}$ do not contain an entry for time stamp u', obviously no entry is attached to the value.

The rule PUT specifies the operation that writes and (if necessary) defines a property. It first computes the location ℓ and the collection $\mathcal{M}$ of access paths of the object and then checks the write permission to the object with the premise

[3] The case, that $u = u'$ never arises, hence $u > u'$ is equivalent to $u \geq u'$.

Program 9.1 Permit expression with an alias.

```
let x = new in
let y = x in
permit x : {a},{a} in
  y.b  // y.b is granted, x.b not
```

(a) Valid Access.

```
let x = new in
permit x : {a},{a} in
let y = x in
  y.b  // violation
```

(b) Invalid Access.

$\mathcal{W} \vdash_M \mathcal{M}.p$. It overwrites the object's property with the new value. The value is stored together with the time stamp u'' in the object. Hence, the time stamp u'' is consumed and the rule passes $u'' + 1$ to the outer context to avoid collisions with later permit or put operations.

The rule PERMIT specifies the access permission contract operation. The rule installs an access permission contract under a unique time stamp u. Hence, u is consumed and $u+1$ is the smallest time stamp that may be used by other rules. As a consequence, the rule PERMIT executes the body of the permit operation under the fresh time stamp $u + 1$. It also modifies the variable binding for x. The modified binding records that under the time stamp u the object bound to x (if any) is reachable by an empty path from x in the actual heap. Hence, the rule attaches $[u \mapsto \varepsilon]$ to the indexed path map of the reference $\rho(x)$. It is realized by the operation $\rho(x) \lhd [u \mapsto \varepsilon]$. Additionally, the rule extends the indexed collections of read and write permissions with L_r and L_w.

An access permission contract is dynamically scoped, because the access permission contracts are propagated with the flow of execution and the rule CHECK PERMISSION only considers the entry points in the domain of the current access permission contract $\mathcal{P}$. In particular, access contracts are not captured by closures created while they are in force: Closure creation (rule LAM) ignores the access contracts and function application (rule APP) continues to use the current permissions with the body of the invoked function. Hence, after evaluation of the body of an access permission contract is complete, paths associated with its entry point u could be garbage collected both from the value and from the heap.

9.3 Examples of the Formal System

9.3.1 Handling Aliases

Let us start with a simple example exploring the basic rules of the formal calculus. The program fragments in Program 9.1 serve as an example to clarify how the system handles aliases. The execution of Fragment (a) creates a heap containing one object. The scope ρ, in which the body of the second let expression is evaluated, contains two variables x and y. Both variables contain references to the object. Since the access path map is stored inside the references, $\rho(x)$ and $\rho(y)$ each do have their own access path map. Fragment (a) installs the access permission

Program 9.2 Nested permissions.

```
1 let x = new in // object x
2 let y = new in // object y
3   x.a = new;  // object o
4   permit y : {a}, {a} in
5   permit x : {a}, {a} in
6     x.a = y;
7     x.a.a = 42
```

contract after the creation of the alias y. The permit expression registers an access permission contract under the time stamp 0 in $\mathcal{R}$ and $\mathcal{W}$. The access path map of the reference x is extended due to the permit operation, while the access path map of y is not. Therefore, the read access to property b is prohibited if it is performed with the reference x, but granted if y is used.

In Fragment (b) of Program 9.1, line 2 and 3 are swapped such that the access permission contract is established before the alias y of x is created. In this situation, the access path map of the reference to the object, stored in the variable x, is first extended. Afterwards, the alias is created and the reference is copied. Therefore, the alias y to the object also contains the entry in its access path map for the time stamp 0, which ends in a rejection of the access, even through the variable y.

9.3.2 Exploring the Override Function

The code in Program 9.2 is supposed to exercise case 3 of the definition of $\olessthan_u$. For conciseness, we extend the language with a let expression and sequential execution in the usual way. After establishing the two permissions, the environment ρ is $[x \mapsto (\ell_x, [u_3 \mapsto \varepsilon]), y \mapsto (\ell_y, [u_2 \mapsto \varepsilon])]$ where the u_i are sorted according to their indexes. After the assignment x.a = y (with serial number u_4) the object in location ℓ_x is $\{a : (u_4, (\ell_y, [u_2 \mapsto \varepsilon]))\}$. In line 7, x.a evaluates to

$$[u_3 \mapsto a] \olessthan (u_4, (\ell_y, [u_2 \mapsto \varepsilon])) = (\ell_y, [u_3 \mapsto a] \olessthan_{u_4} [u_2 \mapsto \varepsilon]) = (\ell_y, [u_2 \mapsto \varepsilon])$$

Case 3 of $\olessthan_u$ applies because $u_4 \geq u_3$. In consequence, u_3 vanishes from the domain of the map because the object that was reachable via x.a before line 6 has become garbage. With this reasoning, the update of x.a.a is permitted, because it is equivalent to y.a.

The code fragments in Program 9.3 serve to illustrate the two remaining cases of the $\olessthan_u$ operator. The code fragments differ only in the placement of the permit expression. Fragment (a) installs the permission *before* the assignment x.a = x.b, whereas Fragment (b) installs the permission afterwards. In both cases, let the permit expression be associated with the time stamp u' and let x.b contain a location ℓ_b paired with an empty map (according to rule NEW).

Program 9.3 Exercising the definition of $\olessthan_u$.

```
let x = new in
  x.a = new;
  x.b = new;
  permit x :
    {a,b,b.a},{a,b.a} in
    x.a = x.b;
    x.a.a = 42
```

(a) Valid access.

```
let x = new in
  x.a = new;
  x.b = new;
  x.a = x.b;
  permit x :
    {a,b,b.a},{a,b.a} in
    x.a.a = 42
```

(b) Invalid access.

The expression x.b on the left hand side returns the location ℓ_b paired with the map $[u' \mapsto b]$ (according to case 2 of $\olessthan_u$: $u < u'$, because it was generated by the preceding assignment x.b = **new**). This value is written to x.a. The following access to x.a returns $(\ell_b, [u' \mapsto b])$ according to case 1 of $\olessthan_u$, which governs that the paths stored in the object take precedence. For the final write access, the extended access map $[u' \mapsto b.a]$ is checked against the set of write permissions and succeeds.

On the right hand side x.a = x.b is executed before the permit expression. Hence, x.a contains $(\ell_b, \emptyset)$ and the GET rule makes it return $(\ell_b, [u' \mapsto a])$ according to case 1 of $\olessthan_u$. For the write operation, the extended access map $[u' \mapsto a.a]$ is checked against the set of write permissions and fails.

For a more complex example, which explores all cases of the override operation in one property read operation, see the next section.

9.3.2.1 One Example to Rule Them All

Program 9.4 is an example that depends on all three cases of the override operation. The example shows that there is no possibility to simplify the override operation. Figure 9.8 contains the heap snapshots of Program 9.4. The snapshot of the function f contains a list of nodes. Each node points to its sucessor, or to the value undef, which is also presented as an edge from the property to the undef node. The access permission contract of f states, that all these edges where readable, and that the next property of the x2 object is writeable.

The graphs keep the scope tables from the outer calls to simplify finding the correspondence between the edge in the graph and an earlier snapshot, for example the dashed edges in the graphs of g and h are the one, which already exists in the snapshot of f.

Before calling g, f creates the alias x2 and a new object xn. The alias makes it necessary for the system to track that x2 is reachable with respect to the snapshot of f under the path next.next from the parameter x0. Hence, the reference stored in the variable x2 contains the following path map (we use n instead of next to shorten the maps a little bit):

$$[0 \mapsto \texttt{n.n}]$$

Program 9.4 Override operation, an example.

```
/*c ... with [x2.next.bla] */ // time stamp: 4
function h(x2) {
  x2.next.bla = 5;   // time stamp: 5
}

/*c ... with [ x2.next, xn.bla ] */
// time stamp 1 for x2, time stamp 2 for xn
function g(x2,xn) {
  x2.next = xn; // time stamp: 3
  h(x2);
}

/*c ... with [x0.next.next.next] */ // timestap: 0
function f(x0) {
  var x2 = x.next.next;
  var xn = { };
  g(x2,xn);
}

var x0 = { next : { next : { next : {} } } };
f(x0);
```

Since g has two parameters, we have to convert the access permission contract into two parts, each with its own time stamp. The first parameter x2 gets time stamp 1, the second parameter xn gets time stamp 2. Both refer to the same heap. So the two time stamps are used to distinguish the two variables from each other. During the execution of g, the new object xn is stored under the property next of the x2 object. Hence, the object's property will contain a reference to the new object, and a time stamp 3. The reference contains the indexed map $[2 \mapsto \varepsilon]$.

The call of h extends the indexed map of x2 by the entry for time stamp 4:

$$[0 \mapsto \mathtt{n.n}, 1 \mapsto \varepsilon, 4 \mapsto \epsilon] \tag{9.1}$$

The override operation is used to merge the following path maps during the get operation of x2.next in h:

$$\begin{aligned}[0 \mapsto \mathtt{n.n}, 1 \mapsto \varepsilon, 4 \mapsto \varepsilon].\mathtt{n} \ \olessthan_3 [2 \mapsto \varepsilon] &= [0 \mapsto \mathtt{n.n.n}, 1 \mapsto \mathtt{n}, 4 \mapsto \mathtt{n}] \ \olessthan_3 [2 \mapsto \varepsilon] \\ &= [2 \mapsto \varepsilon, 4 \mapsto n]\end{aligned}$$

An additional extension of this map with the bla property results in: $[2 \mapsto \mathtt{bla}, 4 \mapsto \mathtt{n.bla}]$, which is supported by the access permission contracts of g (the one registered under timestamp 2) and h. For the access permission contract of f and for the access permission contract of g (the one registered under timestamp 1) the last assignment is not checked, because the object xn was not reachable in the snapshots from x0 or x2.

This example demonstrates that in a single read operation all four cases of the override operation are used to compute the correct resulting map. Hence, it is not possible to simplify the path handling in the framework without giving up the prestate snapshot principle.

9.4 Formal Properties

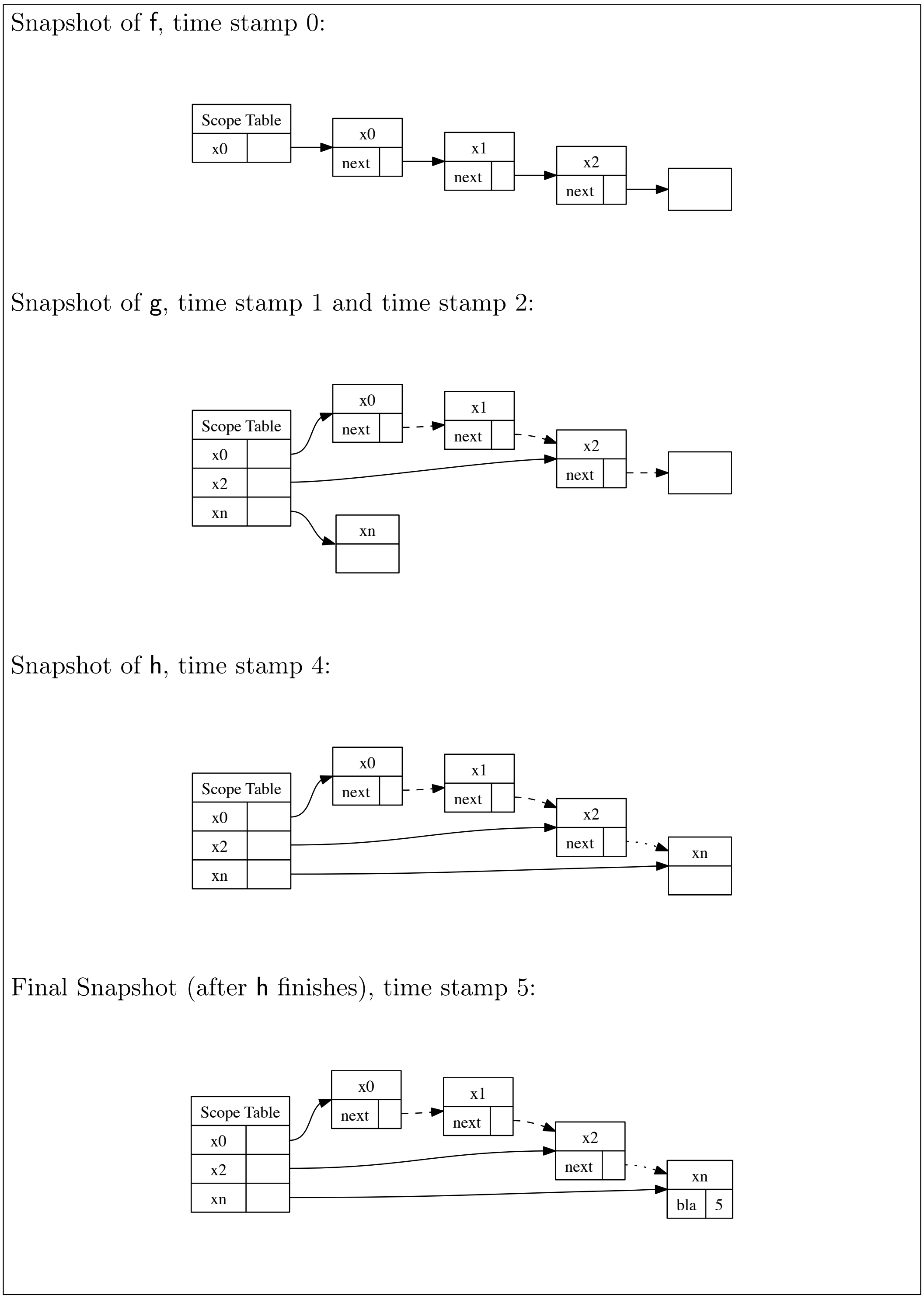
9.4.1 Time Stamps

A simple property that is used in the description of the override operation states that in case 3 of the override definition ($u \geq u'$) it never happens that $u = u'$.

Lemma 9.4.1 (Time Stamps for Writes and Permits)**.** *For $\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H', u', v$ it holds that for each override operation $\mathcal{N} \olessthan_u \mathcal{M}$ used during the evaluation $u \notin dom(\mathcal{M})$.*

The proof establishes the invariant that the set of time stamps in the heap identifying a write operation and the time stamps used to identify registered access permission contracts are disjoint. We call this property time stamp consistency of the heap.

Definition 9.4.2. *A heap H is time stamp consistent, if all its values are H time stamp consistent. A value v is H time stamp consistent, if*



Snapshot of f, time stamp 0:

Snapshot of g, time stamp 1 and time stamp 2:

Snapshot of h, time stamp 4:

Final Snapshot (after h finishes), time stamp 5:

Figure 9.8: $\mathcal{JSE}$ – heaps of override example.

- *$v = (\rho, \lambda x.e)$ and ρ is H time stamp consistent or*
- *$v = (\ell_v, \mathcal{M}_v)$, for all $u \in dom(\mathcal{M}_v)$, for all $\ell \in dom(H)$ and for all $p \in dom(H(\ell))$ it holds that $H(\ell)(p) = (u', v')$ implies $u \neq u'$.*

An environment ρ is H time stamp consistent, if for all $x \in dom(\rho)$, $\rho(x)$ is H time stamp consistent.

Proof of Lemma 9.4.1. by induction over $\hookrightarrow$.
We establish the invariant that the heap and the variable environment stays time stamp consistent and that the time stamp returned by the evaluation is not used if the time stamp parameter of the evaluation is fresh.

- *Case* Put, Permit: The invariant ensures that the time stamp u is not used. Therefore, the use either for the permit or for the write operation cannot invalidate the fact that the sets of time stamps for write and permits are disjoint. Since the recursive call is done with an increased, hence fresh, time stamp, induction is applicable and yields the desired result.
- *Case* others: All other cases are trivial or just by induction.

□

9.4.2 Reachability

The semantics of access permission contracts has some subtle implications. For instance, if x is bound to some object ℓ, then the access contract `permit` x : $\emptyset, \emptyset$ `in` e does *not* establish the protection of all properties of the object ℓ during the execution of e. Program 9.1 is a simple example for a violation of this assumption.

So what do access permission contracts actually enforce? The problem with Program 9.1 is that x and y are aliases of one another. A meaningful characterization of the guarantees of an access contract must consider aliasing. To formulate a precise statement, we extend the evaluation judgment to trace all read and write accesses in sets $T^r, T^w \subseteq \text{Loc} \times \mathit{Prop}$:

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; v\ [T^r, T^w]$$

Figure 9.9 shows the modified rules for property read and write; the remaining rules union the trace sets from the subcomputations as in the put' rule.

Furthermore, *reach* refers to all heap locations reachable from a given object location with respect to a heap H. The type of the mapping is $\mathit{reach} : \mathit{Heap} \times \mathit{Val} \xrightarrow{\mathit{fin}} 2^{\mathit{Loc}}$. It returns the set of locations that are reachable from an input value v in a heap using the auxiliary function $\Downarrow$ (see Figure 9.10). The *acc* function yields a set of location and property names for all accessible properties along a path $\pi \in \Pi$ under a heap H.

GET’

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T^r, T^w] \\ \mathcal{R} \vdash_{\text{chk}} \mathcal{M}.p \qquad v' = \mathcal{M}.p \olessthan H'(\ell)(p)\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow' H'; u'; v'\ [T^r \cup \{(\ell, p)\}, T^w]}$$

PUT’

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T_1^r, T_1^w] \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow' H''; u''; v\ [T_2^r, T_2^w] \\ \mathcal{W} \vdash_{\text{chk}} \mathcal{M}.p \qquad H''' = H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \qquad \hookrightarrow' H'''; u'' + 1; v\ [T_1^r \cup T_2^r, T_1^w \cup T_2^w \cup \{(\ell, p)\}]}$$

Figure 9.9: $\mathcal{JSE}$ – tracing property read and write.

Theorem 9.4.3 (Completeness)**.** *Suppose that*

$$\rho, \mathcal{R}, \mathcal{W} \vdash H_0; u; \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e \hookrightarrow' H_1; u'; v\ [T^r, T^w] \tag{9.2}$$

and that

$$\mathit{reach}(H_0, \rho(\mathit{fv}(e) \setminus \{x\})) \cap X = \emptyset \tag{9.3}$$

where $X = \mathit{reach}(H_0, \rho(x))$.

Then $T^r \cap (X \times \mathit{Prop}) \subseteq \mathit{acc}(H_0, \rho(x), L_r)$ and $T^w \cap (X \times \mathit{Prop}) \subseteq \mathit{acc}(H_0, \rho(x), L_w)$.

Condition (9.3) states that everything reachable from the variable x, is not reachable from any other free variable of e. The conclusion of the theorem is that for every access pair $(\ell, p) \in T_r$ where ℓ happens to be reachable from $\rho(x)$ this access must be sanctioned by the language L_r of read permissions. The latter is formalized via the *acc* function: It splits every access path in L_r in a prefix π and last property p, computes the dereferenced locations from $\rho(x)$ along path π and pairs the results (at most one) with p.

To prove this theorem, we establish an invariant that we formulate for the judgment without the traces because they are not needed to prove it. The assumption $\rho, \mathcal{R}, \mathcal{W} \vdash H_0; u_x; \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e \hookrightarrow H_1; u_1; v$ in the theorem can only hold (by inversion) if its premise also holds:

$$\rho', \mathcal{R}[u_x \mapsto L_r], \mathcal{W}[u_x \mapsto L_w] \vdash H_0; u_x + 1; e \hookrightarrow H_1; u_1; v \tag{9.4}$$

where $\rho' = \rho[x \mapsto \rho(x) \olessthan [u_x \mapsto \varepsilon]]$. Let’s further assume that $\rho(x) = (\ell_x, m_x) \in \mathit{Ref}$ – otherwise, the theorem is trivially true because $v \notin \mathit{Ref} \Rightarrow \Downarrow(H_0, v, \pi) = \emptyset$ for all π, so that $X = \emptyset$.

Definition 9.4.4. *A value v is* primarily reachable *(short: p.r.) from ℓ_x with index u_x in H_0, if*

$$
\begin{array}{lcl}
\mathit{reach}(H, \{v_1, \dots, v_n\}) & = & \bigcup_i \mathit{reach}(H, v_i) \\
\mathit{reach}(H, v) & = & \Downarrow(H, v, \mathit{Path}) \\
\Downarrow(H, v, \Pi) & = & \bigcup\{\Downarrow(H, v, \pi) \mid \pi \in \Pi\} \\
\Downarrow(H, (u, v), \pi) & = & \Downarrow(H, v, \pi) \\
\Downarrow(H, v, \pi) & = & \begin{cases} \Downarrow'(H, \ell, \pi) & v = (\ell, m) \\ \emptyset & v \notin \mathit{Ref} \end{cases} \\
\Downarrow'(H, \ell, \varepsilon) & = & \{\ell\} \\
\Downarrow'(H, \ell, p.\pi) & = & \begin{cases} \Downarrow(H, H(\ell)(p), \pi) & p \in \mathrm{dom}(H(\ell)) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{acc}(H, v, \Pi) & = & \bigcup\{\mathit{acc}(H, v, \pi) \mid \pi \in \Pi\} \\
\mathit{acc}(H, (\ell, \mathcal{M}), \pi.p) & = & \{(\ell', p) \mid \ell' \in \Downarrow'(H, \ell, \pi)\} \\
\mathit{acc}(H, v, \pi) & = & \emptyset \qquad \text{if } v \notin \mathit{Ref}
\end{array}
$$

Figure 9.10: $\mathcal{JSE}$ – heap traversal.

- *$v = (\ell, \mathcal{M})$ with $u_x \in dom(\mathcal{M})$ implies that*

$$(\ell, p) \in acc(H_0, \ell_x, \mathcal{M}(u_x)) \tag{9.5}$$

- *$v = (\rho, \lambda y.e)$ with ρ is p.r.*

An environment ρ is p.r. if $(\forall y \in dom(\rho))$ $\rho(y)$ is p.r. A heap H is p.r. if $\forall \ell \in dom(H)$ and $\forall p \in dom(H(\ell))$ $H(\ell)(p)$ are p.r. A pair (u, v) is p.r. if v is p.r. (All with respect to the same fixed ℓ_x, u_x, and H_0.)

Lemma 9.4.5. *For each judgment of $\rho', \mathcal{R}', \mathcal{W}' \vdash H'; u'; e' \hookrightarrow H''; u''; v''$ occurring in the derivation of (9.4) it holds that: if ρ' and H' are p.r. from ℓ_x with index u_x in H_0, then so are H'' and v''.*

Proof. By induction on the derivation. Each case refers to the variables used in the respective rule in Figure 9.4.

Case VAR: Obviously true.

Case LAM: Obviously true.

Case APP: By the assumption on ρ and H, induction on e_0 yields H' and ρ' p.r. As now ρ and H' are p.r., induction yields that H'' and v_1 are p.r. As $\rho'[x \mapsto v_1]$ and H'' are p.r., induction yields H''' and v are p.r., which proves the result.

Case NEW: The heap $H[\ell \mapsto \emptyset]$ and the value $(\ell, \emptyset)$ are both p.r.

Case GET: By induction H' and $(\ell, \mathcal{M})$ are p.r. That means, if $u_x \in \mathrm{dom}(\mathcal{M})$ then $\ell \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x))$. It remains to show that $\mathcal{M}.p \olessthan H'(\ell)(p)$ is p.r. The only significant case occurs if $H'(\ell)(p) = (u, (\ell', \mathcal{N}))$, in which case the returned value is $\mathcal{M}.p \olessthan (u, (\ell', \mathcal{N})) = (\ell', \mathcal{M}.p \olessthan_u \mathcal{N})$.

If $u_x \in \mathrm{dom}(\mathcal{N})$, then the heap location has changed its content since the access permission contract is associated with u_x and it has been overwritten with a value reachable in H_0 from ℓ_x on the path $\mathcal{N}(u_x)$. This path annotation has to stay in force to ensure that the result $(\mathcal{M}.p \olessthan_u \mathcal{N})(u_x) = \mathcal{N}(u_x)$ is p.r. It holds by induction that $\mathcal{N}(u_x)$ is p.r.

If $u_x \notin \mathrm{dom}(\mathcal{N})$, then the contents of the heap location has not yet been reached from ℓ_x. There are two cases, which can be distinguished by comparing u and u_x. If $u < u_x$, then the heap location has not changed compared with H_0. Thus the result can be marked as visited. This is expressed by $(\mathcal{M}.p \olessthan_u \mathcal{N})(u_x) = (\mathcal{M}.p)(u_x) = \mathcal{M}(u_x).p$. By the property read that happens in this rule, it is clear that $\ell' \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x).p)$.

If, however, $u > u_x$, then the heap location has changed compared to H_0, but the new value is not reachable from ℓ_x in H_0. For that reason, the value must not receive a u_x annotation. This is expressed by $(\mathcal{M}.p \olessthan_u \mathcal{N})(u_x) = \text{undefined}$.

The case $u = u_x$ never occurs, because u_x is a time stamp used to register an access permission contract. It can't be used by a write operation.

Case PUT: By induction H' and $(\ell, \mathcal{M})$ are p.r. Hence, H'' and v are also p.r. by induction. The final heap is p.r., because the rule overwrites a value with a p.r. value.

Case PERMIT: Immediate by induction. □

Proof of Theorem 9.4.3. The proof is by induction on the evaluation judgment with traces. We observe that the top-level judgment "seeds" the lemma in a non-trivial way. The environment $\rho[x \mapsto \rho(x) \olessthan [u_x \mapsto \varepsilon]]$ is p.r. with respect to u_x, ℓ_x, and H_0 (from (9.4)) because $\ell_x \in \Downarrow'(H_0, \ell_x, \varepsilon)$ and no other environment entry refers to u_x. Similarly, the heap H_0 is p.r. because it does not contain any reference to u_x. Thus, the lemma tells us that H_1 and v are also p.r.

We can conclude from the lemma that all subderivations only return p.r. values. *Case distinction* of $\hookrightarrow$.

- *Case* GET': Since the rule GET' is the only rule that extends the set T^r, we have to show that

  $$(\ell, p) \in acc(H_0, \rho(x), L_r)$$

  for a p.r. $(\ell, \mathcal{M})$ (from ℓ_x with respect to u_x) or that $\ell \notin X$.

  If $u_x \in \mathrm{dom}(\mathcal{M})$ then $\ell \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x))$. Inversion of GET' also yields $\mathcal{R} \vdash_{\text{chk}} \mathcal{M}.p$. Hence $\mathcal{M}.p(u_x) \in L_r$ holds, and we can conclude that $(\ell, p) \in acc(H_0, \rho(x), L_r)$.

  If $u_x \notin \mathrm{dom}(\mathcal{M})$, it holds that $\ell \notin X$. This holds due to the definition of primary reachable values and the precondition of the theorem.

- *Case* PUT': The rule PUT' extends the set T^w. We have to show that

  $$(\ell, p) \in acc(H_0, \rho(x), L_w)$$

for a p.r. $(\ell, \mathcal{M})$ (from ℓ_x with respect to u_x) or that $\ell \notin X$.

If $u_x \in \text{dom}(\mathcal{M})$, then $\ell \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x))$. Inversion of PUT' yields $\mathcal{W} \vdash_{\text{chk}} \mathcal{M}.p$. Hence $\mathcal{M}.p(u_x) \in L_w$ and we can conclude that $(\ell, p) \in acc(H_0, \rho(x), L_w)$.

If $u_x \notin \text{dom}(\mathcal{M})$, it holds by the definition of primary reachable and the precondition of the theorem that $\ell \notin X$.

End case distinction of $\hookrightarrow$.

□

9.4.3 Stability of Violation

Stability of violation is a property linked to the reference attachment principle (Section 9.1.6). It states that a violation of an access permission contract is preserved (in a precisely defined sense) when restarting the same computation on a heap with more aliasing.

Let us first clarify what we mean by saying "more aliasing." For two heaps H_1 and H_2, H_2 has more aliasing than H_1 if H_2 identifies locations that are distinct in H_1 and merges the contents of the objects in these locations. That is, if o' and o'' are distinct objects in H_1 which are merged to object o in H_2, then o has all properties from o' and o''. Properties present in o' and o'' must have suitably related values that map into the same value in H_2. We call H_1 a refinement of H_2, because it makes more distinctions between objects.

Definition 9.4.6. *A heap H_1 is a γ-*refinement *of heap H_2, written as $H_1 \succcurlyeq_\gamma H_2$, if $\gamma : dom(H_1) \to dom(H_2)$ is a surjective mapping between heap locations and $\forall \ell_1 \in dom(H_1)$, $o_1 = H_1(\ell_1)$, $o_2 = H_2(\gamma(\ell_1))$:*

RH1 *$dom(o_1) \subseteq dom(o_2)$ (objects in the refined heap have fewer properties) and*

RH2 $(\forall p \in dom(o_1))\ o_1(p) = (u_1, v_1) \wedge o_2(p) = (u_2, v_2) \wedge u_1 = u_2 \Rightarrow v_1 \succcurlyeq_\gamma v_2$

A value is a γ-refinement of another value, written as $v_1 \succcurlyeq_\gamma v_2$ iff

RV1 *$v_1 = (\ell_1, \mathcal{M}_1)$ and $v_2 = (\ell_2, \mathcal{M}_2)$ and $\ell_2 = \gamma(\ell_1)$ and $\mathcal{M}_1 = \mathcal{M}_2$, or*

RV2 *$v_1 = (\rho_1, e_1)$ and $v_2 = (\rho_2, e_2)$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and $e_1 = e_2$.*

An environment is a γ-refinement of another, $\rho_1 \succcurlyeq_\gamma \rho_2$ iff

RE1 *$dom(\rho_1) = dom(\rho_2)$ and*

RE2 $(\forall x \in dom(\rho_1))\ \rho_1(x) \succcurlyeq_\gamma \rho_2(x)$.

The implication in **RH2** might be surprising. This choice allows the coarser heap H_2 to contain a value which does not refine to all corresponding values in heap H_1: for each object in H_2, there may be any number of γ-preimages of this object in H_1. **RH2** states that such an object does not need to be consistent with all its preimages. This case can be detected by the condition $u_1 < u_2$: the shared version of the object has been updated after one of its unshared preimages. The remaining case $u_1 > u_2$ can never arise.

Such inconsistencies are allowed in a heap refinement, because they only influence the semantics of a program if there is a subsequent read operation that observes the inconsistency. In this case, the criterion $u_1 < u_2$ can be used to detect the inconsistency and react accordingly.

Having established the notion of heap refinement, it remains to run the same program on two heaps and compare the outcomes. Finally, it is not sufficient to consider only successful, terminating evaluations. Evaluations that end in a contract violation or an interrupted evaluation must be taken into account, too. Figure 9.11 specifies the rules of three judments of the form $\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \Uparrow^i$ where $i \in \{R, W, O\}$. Each judgment formalizes an interrupted evaluation. The superscript R indicates a missing read permission (rule GET-CRASH2). Superscript W indicates a missing write permission (rule PUT-CRASH3). Superscript O indicates non-deterministically giving up on a read operation (rule GET-CRASH3). The remaining rules are straightforward variants of the evaluation rules in Figure 9.4 that propagate the error conditions in the standard way.

The error rules are only intended to capture permission errors. Derivations with type errors or non-termination do not exist, as it is common with big-step semantics.

These judgments enable us to capture the notion of an inconsistent read operation as discussed after Definition 9.4.6. The following definition formalizes the intuitions of this discussion.

Definition 9.4.7. *Let $H_1 \succcurlyeq_\gamma H_2$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and*

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1' \tag{9.6}$$

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \Uparrow^O \tag{9.7}$$

Derivation (9.7) ends in an inconsistent read operation with respect to (9.6) *if there are corresponding subderivations of (9.6) ending in an instance of* GET

$$\textsc{get}\quad \frac{\begin{array}{c}\rho_1'', \mathcal{R}'', \mathcal{W}'' \vdash H_1''; u''; e'' \hookrightarrow H_1'''; u'''; (\ell_1, \mathcal{M}_1) \\ \mathcal{R}'' \vdash_{chk} \mathcal{M}_1.p \qquad v_1''' = \mathcal{M}_1.p \olessthan H_1'''(\ell_1)(p)\end{array}}{\rho_1'', \mathcal{R}'', \mathcal{W}'' \vdash H_1''; u''; e''.p \hookrightarrow H_1'''; u'''; v_1'''}$$

and (9.7) ending in an instance of GET-CRASH3

$$\textsc{get-crash3}\quad \frac{\rho_2'', \mathcal{R}'', \mathcal{W}'' \vdash H_2''; u''; e'' \hookrightarrow H_2'''; u'''; (\ell_2, \mathcal{M}_2) \qquad \mathcal{R}'' \vdash_{chk} \mathcal{M}_2.p}{\rho_2'', \mathcal{R}'', \mathcal{W}'' \vdash H_2''; u''; e''.p \Uparrow^O}$$

such that there exists γ'' extending γ and γ''' extending γ'' where $\rho_1'' \succcurlyeq_{\gamma''} \rho_2''$, $H_1'' \succcurlyeq_{\gamma''} H_2''$, $H_1''' \succcurlyeq_{\gamma'''} H_2'''$, but $H_1'''(\ell_1)(p) = (u_1, v_1)$ and $H_2'''(\ell_2)(p) = (u_2, v_2)$ with $u_1 < u_2$.

APP-CRASH1
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}$$

APP-CRASH2
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}$$

APP-CRASH3
$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \hookrightarrow H''; u''; v_1 \qquad \rho'[x \mapsto v_1], \mathcal{R}, \mathcal{W} \vdash H''; u''; e \Uparrow^i\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}$$

GET-CRASH1
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^i}$$

GET-CRASH2
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \nvdash_{\text{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^R}$$

GET-CRASH3
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \vdash_{\text{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^O}$$

PUT-CRASH1
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^i}$$

PUT-CRASH2
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^i}$$

PUT-CRASH3
$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \qquad \mathcal{W} \nvdash_{\text{chk}} \mathcal{M}.p\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^W}$$

PERMIT-CRASH
$$\frac{\rho[x \mapsto \rho(x) \lhd [u \mapsto \varepsilon]], \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H; u+1; e \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e \Uparrow^i}$$

Figure 9.11: $\mathcal{JSE}$ – crashing and partial computations.

The definition mentions the notion of γ' extending γ. This "extending" relation between maps is the reflexive, transitive closure of the "directly extends" relation: γ' directly extends γ if $\gamma' = \gamma[\ell_1 \mapsto \ell_2]$ where $\ell_1 \notin \mathrm{dom}(\gamma)$ and $\ell_2 \notin \mathrm{ran}(\gamma)$, where $\mathrm{ran}(\gamma)$ denotes the range of γ.

We are now ready to state two theorems about related executions on refined heaps. The second theorem is our desired stability result, but the first one is needed to prove it. The following theorem states that a program running successfully on some heap runs on a coarser heap with more aliasing either successfully or crashes due to a inconsistent read after write operation.

Theorem 9.4.8 (Simulation)**.** *If*

$$H_1 \succcurlyeq_\gamma H_2 \tag{9.8}$$

$$\rho_1 \succcurlyeq_\gamma \rho_2 \tag{9.9}$$

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1' \tag{9.10}$$

then either

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \hookrightarrow H_2'; u_2'; v_2' \tag{9.11}$$

such that there exists γ' extending γ where $H_1' \succcurlyeq_{\gamma'} H_2'$ and $u_1' = u_2'$ and $v_1' \succcurlyeq_{\gamma'} v_2'$ or

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \Uparrow^O \tag{9.12}$$

such that the derivation of (9.12) ends in an inconsistent read operation with respect to (9.10).

Proof. We prove the theorem by induction on the derivation of (9.10).
Case distinction over $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1'$.

- *Case* Var, $e \equiv x$: From (9.10) we conclude $x \in \mathrm{dom}(\rho_1)$ and $H_1' = H_1$ and $v_1' = \rho_1(x)$. By (9.9) it holds that $x \in \mathrm{dom}(\rho_2)$. Therefore, Var is applicable to the expression under the environment ρ_2. Since the heap does not change due to the rule application, $H_2 = H_2'$ holds. Therefore, $H_1' \succcurlyeq_\gamma H_2'$ is trivial by (9.8). It also holds that $v_1' \succcurlyeq_\gamma v_2'$ due to (9.9) and $\rho_1(x) = v_1'$ and $\rho_2(x) = v_2'$. It trivially holds that the time stamps are equal.

- *Case* Lam, $e \equiv \lambda x.e'$: Since the rule Lam does not have any preconditions considering the heap or the variable environment, (9.11) holds trivially for the heap H_2. It is also clear that the time stamps are equal and that $H_1' \succcurlyeq_\gamma H_2'$ holds. It is left to show that

$$v_1' = (\rho_1 \downarrow_{\mathrm{fv}(\lambda x.e')}, \lambda x.e') \succcurlyeq_\gamma (\rho_2 \downarrow_{\mathrm{fv}(\lambda x.e')}, \lambda x.e') = v_2' \tag{9.13}$$

  (9.13) holds since it is obvious that $\rho_1 \succcurlyeq_\gamma \rho_2$ implies for all X

$$\rho_1 \downarrow X \succcurlyeq_\gamma \rho_2 \downarrow X \tag{9.14}$$

  (which can easily be proven by induction on the elements of X).

- *Case* App, $e \equiv e_0(e_1)$: We can conclude by inversion of (9.10) that e_0 simplifies to a closure under heap H_1 and variable environment ρ_1:

  $$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_0 \hookrightarrow H_1'; u'; (\rho_1', \lambda x.e') \tag{9.15}$$

  The induction hypothesis lets us now conclude that either e_0 crashes, or that the first conclusion of the theorem holds. If e_0 crashes, this leads to a crash of the expression $e_0(e_1)$ under H_2 and ρ_2, too. Hence, in this case the theorem is proven.

  Let us continue under the assumption, that e_0 is executable under the heap H_2 and the variable environment ρ_2. The induction hypothesis implies that there exists γ', which is an extension of γ, such that

  $$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_0 \hookrightarrow H_2', u', (\rho_2', \lambda x.e') \tag{9.16}$$

  with $H_1' \succcurlyeq_{\gamma'} H_2'$ and $(\rho_1', \lambda x.e') \succcurlyeq_{\gamma'} (\rho_2', \lambda x.e')$.

  Next, we continue by using the induction hypothesis for e_1 with γ'.

  The third step is to apply induction on the body of the function e'. The function is executed under a variable environment, in which x is bound to v_1, the result of execution e_1. Obviously all preconditions of the theorem are fulfilled, and the error cases can be handled analogous to the error case for expression e_0.

  It is important for the proof to work correctly that if γ' is an extension of γ and γ'' is an extension of γ' then γ'' is also an extension of γ.

- *Case* New, $e \equiv \texttt{new}$: It holds $H_1' = H_1[\ell_1 \mapsto \emptyset]$ and $v_1' = (\ell_1, \emptyset)$ for an arbitrary $\ell_1 \notin \text{dom}(H_1)$. Obviously New is applicable under H_2 and ρ_2 ($H_2' = H_2[\ell_2 \mapsto \emptyset]$ and $v_2' = (\ell_2, \emptyset)$ for an arbitrary $\ell_2 \notin \text{dom}(H_2)$). The existens of such an ℓ_2 is trivial due to the fact that the domain of the heap H_2 is finite.

  The extension $\gamma' = \gamma[\ell_1 \mapsto \ell_2]$ relates the values v_1' and v_2' and the heaps H_1' and H_2' as asked by the theorem.

- *Case* Get, $e \equiv e'.p$: Inversion of (9.10) yields

  $$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e' \hookrightarrow H_1'; u'; (\ell_1, \mathcal{M}_1) \tag{9.17}$$
  $$\mathcal{R} \vdash_{\text{chk}} \mathcal{M}_1.p \tag{9.18}$$
  $$v_1' = \mathcal{M}_1.p \olessthan H_1'(\ell_1)(p) \tag{9.19}$$

  Induction on (9.17) yields

  $$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e' \hookrightarrow H_2'; u'; (\ell_2, \mathcal{M}_2) \tag{9.20}$$

for an γ' extending γ such that

$$H'_1 \succcurlyeq_{\gamma'} H'_2 \tag{9.21}$$

$$(\ell_1, \mathcal{M}_1) \succcurlyeq_{\gamma'} (\ell_2, \mathcal{M}_2) \tag{9.22}$$

(we can ignore the case in which e' crashes since in that case e crashes under H_2 and ρ_2, too). A consequence of (9.22) is that $\gamma'(\ell_1) = \ell_2$. It also implies $\mathcal{M}_1 = \mathcal{M}_2$. We conclude from this with (9.18) that $\mathcal{R} \vdash_{\text{chk}} \mathcal{M}_2.p$. Therefore GET is applicable on e under H_2 and ρ_2:

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H'_2; u', \mathcal{M}_2.p \olessthan H'_2(\ell_2)(p) \tag{9.23}$$

It is left to prove that γ' is a valid extension of γ, such that

$$v'_1 \succcurlyeq_{\gamma'} v'_2 \tag{9.24}$$

where $v'_2 = \mathcal{M}_2.p \olessthan H'_2(\ell_2)(p)$ or that the execution under the heap H_2 and the variable environment ρ_2 crashes. Depending on the time stamps stored inside of the heap H'_1 and H'_2 for ℓ_1, ℓ_2 and property p we can prove either the former or the latter. Let $(u_1, v_1) = H'_1(\ell_1)(p)$ and $(u_2, v_2) = H'_2(\ell_2)(p)$.

Case distinction over the equality of u_1 and u_2.

- *Case* $u_1 = u_2$: If the time stamps are equal, we can conclude from (9.21), that $v_1 \succcurlyeq_{\gamma'} v_2$. (9.24) holds due to $\mathcal{M}_1 = \mathcal{M}_2$ (consider the definition of $\olessthan$).
- *Case* $u'_1 \neq u'_2$: In this case, we can apply GET-CRASH3 to conclude (9.12). Its derivation ends in an inconsistent read operation with respect to (9.10).

End case distinction over the equality of u_1 and u_2.

- *Case* PUT, $e \equiv e'_1.p := e'_2$: Inversion of (9.10) yields

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e'_1 \hookrightarrow H''_1; u'; (\ell_1, \mathcal{M}_1) \tag{9.25}$$

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H''_1; u'; e'_2 \hookrightarrow H'''_1; u''; v'_1 \tag{9.26}$$

$$\mathcal{W} \vdash_{\text{chk}} \mathcal{M}_1.p \tag{9.27}$$

$$H'_1 = H'''_1[\ell_1, p \mapsto (u'', v'_1)] \tag{9.28}$$

$$u'_1 = u'' + 1 \tag{9.29}$$

We proceed with induction over (9.25). We can skip the cases with a crash, because they are analogous. Therefore, we can assume

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e'_1 \hookrightarrow H''_2; u', (\ell_2, \mathcal{M}_2) \tag{9.30}$$

There also exists an extension γ' of γ such that $H''_1 \succcurlyeq_{\gamma'} H''_2$ and $(\ell_1, \mathcal{M}_1) \succcurlyeq_{\gamma'} (\ell_2, \mathcal{M}_2)$. We conclude from the latter $\ell_2 = \gamma'(\ell_1)$ and that $\mathcal{M}_1 = \mathcal{M}_2$.

We continue with induction on (9.26) and find an extension γ'' of γ'. By the fact that $\mathcal{M}_1 = \mathcal{M}_2$ and analogous to the situation in the rule LAM, we can conclude that the rule PUT is applicable. The relations between the resulting heaps and the return values also hold (due to the fact that the value are in relation to each other and the time stamps are equal).

- *Case* PERMIT, $e \equiv \texttt{permit}\ x : L_r, L_w\ \texttt{in}\ e'$: Inversion of (9.10) yields

$$\rho_1', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H_1; u+1; e' \hookrightarrow H_1'; u_1'; v_1' \tag{9.31}$$

$$\rho_1' = \rho_1[x \mapsto \rho_1(x) \lhd [u \mapsto \varepsilon]] \tag{9.32}$$

By induction on (9.31), the evaluation under the heap H_2 and the variable environment $\rho_2' = \rho_2[x \mapsto \rho_2(x) \lhd [u \mapsto \varepsilon]]$ either crashes (continue with PERMIT-CRASH) or it holds that

$$\rho_2', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H_2; u+1; e' \hookrightarrow H_2'; u_2'; v_2' \tag{9.33}$$

where $u_1' = u_2'$, $H_1' \succcurlyeq_{\gamma'} H_2'$ and $v_1' \succcurlyeq_{\gamma'} v_2'$ for some γ' extending γ. Therefore, the rule PERMIT is applicable for the heap H_2 and the environment ρ_2 and yields the desired values.

End case distinction over $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1'$.

□

The second theorem states that crashes due to violated read or write permissions are preserved when more aliasing is added. The main complication here is that an inconsistent read operation in the version with additional aliasing may lead to arbitrary behavior of the program including non-termination. For that reason, the theorem only constructs a related execution up to the first inconsistent read.

Theorem 9.4.9 (Error Preservation). *If $H_1 \succcurlyeq_\gamma H_2$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and*

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \Uparrow^i \tag{9.34}$$

(for $i \in \{R, W\}$) then

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \Uparrow^j \tag{9.35}$$

such that either $i = j$ or $j = O$ and the derivation of (9.35) ends in an inconsistent read operation with respect to (9.34).

Proof. We prove the theorem by induction over (9.34). Because of Theorem 9.4.8 we only consider crashing subcomputations. Therefore, our case distinction only handles the rules for crashing evaluations in Figure 9.11.
Case distinction over $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \Uparrow^i$.

- *Case* APP-CRASH1, GET-CRASH1, PUT-CRASH1, PERMIT-CRASH: These cases just pass through the result. We can directly apply the induction hypothesis and conclude the desired result.

- *Case* App-Crash2, $e \equiv e_0(e_1)$: Inversion of App-Crash2 yields

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_0 \hookrightarrow H_1'; u'; (\rho_1', \lambda x.e') \tag{9.36}$$

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1'; u'; e_1 \Uparrow^i \tag{9.37}$$

  We apply Theorem 9.4.8 to (9.36). Either the expression evaluates normally under the heap H_2 and the environment ρ_2, or it crashes with an inconsistent read operation. In the second case we can apply App-Crash1 to prove that $j = O$. In the first case, we can apply induction on (9.37). The induction yields two possibilities, either $j = i$ or $j = O$. We can continue in both cases by applying App-Crash2 to prove either the first or the second claim of our theorem.

- *Case* App-Crash3, $e \equiv e_0(e_1)$: This case is analogous to App-Crash2. The only difference is that we apply Theorem 9.4.8 twice; first, to the evaluation of e_0, and second, to the evaluation of e_1. The step that puts the two together is handled by induction.

- *Case* Get-Crash2, $e \equiv e'.p$: Inversion yields:

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e' \hookrightarrow H_1'; u'; (\ell_1, \mathcal{M}_1) \tag{9.38}$$

$$\mathcal{R} \not\vdash_{\text{chk}} \mathcal{M}_1.p \tag{9.39}$$

  By Theorem 9.4.8, either e' crashes under H_2 and ρ_2, or it evaluates normally. In the first case, we apply Get-Crash1. In the second case, the result of evaluating e' under H_2 and ρ_2 returns a reference $(\ell_2, \mathcal{M}_2)$, such that $\mathcal{M}_1 = \mathcal{M}_2$. Therefore, $\mathcal{R} \not\vdash_{\text{chk}} \mathcal{M}_2.p$ holds. Hence, we can apply Get-Crash2 to conclude the desired results.

- *Case* Get-Crash3, $e \equiv e'.p$: In this cases $i = O$, which is not allowed by the precondition. Therefore, in this case, there is nothing to prove.

- *Case* Put-Crash2, $e \equiv e_0.p = e_1$: Analogous to App-Crash2.

- *Case* Put-Crash3, $e \equiv e_0.p = e_1$: Similar to Get-Crash2.

  One of the cases is that since the two references resulting from executing e_0 in the two different heaps contain the same maps $\mathcal{M}_1$ and $\mathcal{M}_2$. This implies allows that the third precondition of Push-Crash3 is applicable to the heap H_2 and ρ_2.

End case distinction over $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \Uparrow^i$.

It is interesting to observe that the $\Uparrow^O$ outcome only arises due to subcomputations that did not crash in H_1. So, they are only generated by invocations of Theorem 9.4.8, not by cases handled directly in this proof.

□

10 Inferring Access Permission Contracts

So far we have not discussed how access permission contracts are created. The task of the JSConTest was just to dynamically monitor the provided access permission contracts during random testing, during the execution of a manual test suite or during the execution of the program in the wild. The contracts were provided by the programmer with a notation based on comments (e.g. Chapter 8 or Chapter 12), or, in the formal system (Chapter 9), by a language contract.

If the access permission contracts are already used during software development, the annotation burden is not an issue. The notation of access contracts is fairly compact. Because the system is partially applicable, the programmer may decide to use it only for functions, for which the side effects are important. In such cases, independent from the fact, if access permission contracts were used, there is the need to document the side effects of the functions (otherwise, they could not be important). Therefore, using access permission contracts for the description of side effects is no more effort than using natural langue for this purpose. Furthermore, since access permission contracts come with the automatic dynamic checking facility, the programmer will save a lot of work ensuring that the documentation and the code is synchronized. Therefore, we believe that in this setting, the use of access permission contract will pay off.

But when a tester uses JSConTest to specify the behavior of a software system, the tester typically does not have a lot of intuition about the function, which he has to annotate. Therefore, he tries to extract the access permission contract from a language description, or, if such an description is not available, directly from the source code. Of course, this is a tedious and time consuming task, and it will typically end up in building the access permission contract on a large number of guesses about the dynamic behavior of the code. Then, the tester will run the test suite. If access permission contract violations are reported, the tester will adjust the access permission contract, or the code (and fix a bug), und start with a new iteration, until enough test runs pass without an access permission contract violation.

But it is not enough to find an arbitrary access permission contract that does not yield an access violation in a large number of test runs. Because the access permission contract is a specification of the desired behavior of the system, its form has to be sufficiently general, and, at the same time, it has to be sufficiently specific to give a tight description of the behavior. Only if both conditions are fulfilled, the access permission contract is valuable for documentation purposes and future

$$\begin{array}{lcll}
p & \in & \mathit{Prop} & \text{property names} \\
\pi & ::= & \varepsilon \mid p.\pi & \text{access paths} \\
P & \subseteq & \mathit{Prop} & \text{set of property names} \\
b & ::= & \varepsilon \mid P.b \mid P{*}\,.b & \text{path contract} \\
a & ::= & \emptyset \mid b \mid a + a \mid a - a & \text{access permission contract} \\
\gamma & ::= & \mathbf{R} \mid \mathbf{W} \mid \mathbf{N}_r \mid \mathbf{N}_w & \text{access classifiers} \\
\kappa & ::= & \gamma(\pi) & \text{classified access path} \\
\multicolumn{4}{l}{? = \mathit{Prop}, \qquad @ = \leftarrow = \emptyset \subseteq \mathit{Prop}, \qquad *.b = ?{*}.b}
\end{array}$$

Figure 10.1: Syntax of access paths and access permission contracts.

error detection purposes.

Therefore, the task of an inference algorithm for access permission contracts is to find an access permission contract for a function that is simple to read and write, easy to understand, valid and tight. It is not enough to just find a valid one, ignoring the other requirements. To simplify matters the inference algorithm does not infer arbitrary access permission contracts, but it only computes access permission contracts which are prefix closed (Definition 9.2.1) and prefix accessible (Definition 9.2.1).

Another important aspect of the inference is that the inferred access permission contracts can be attached to the functions as the manually provided access permission contracts. Therefore, even if the formal system does not restrict the read and write set of an access permission contract in any way, our goal is not to find a pair of valid sets, but to find an access contract, as they occur in the implementation.

In Section 10.1 we formally present the access permission contracts as they arise in the implementation. After that we present an inference algorithm for access permission contracts (Section 10.2). Some special cases handled by the implementation, but not by the formal description of the algorithm are presented in Section 10.3. Section 10.4 contains a proof that the algorithm is sound.

10.1 Access Path Formalization

The goal of this section is to develop a formal presentation of access contracts that is more specific than the definition of access contracts used in Chapter 9. But, at the same time, the formalization of access permission contract should not contain to many low level details. For example Figure 8.2 contains a concrete syntax for access permission contracts with a lot of parsing details. For a formal presentation these details should be avoided. Therefore, this section introduces a formal definition of access permission contracts. Of course it is important to connect the three different abstract levels of access permission contracts. Therefore, the section finishes by connecting the three different forms of access permission contracts.

Let us start by introducing access paths as lists of property names (Figure 10.1). An access permission contract a is either the empty set $\emptyset$, a path permission b,

$$\mathbf{W}(\varepsilon) \prec \varepsilon \qquad \mathbf{R}(\varepsilon) \prec b \qquad \mathbf{N}_r(\pi) \prec \varepsilon \qquad \mathbf{N}_w(\varepsilon) \prec \emptyset.\varepsilon \qquad \mathbf{N}_w(\varepsilon) \prec \varepsilon$$

$$\frac{\gamma(\pi) \prec b \qquad p \in P}{\gamma(p.\pi) \prec P.b} \qquad \frac{\gamma(\pi) \prec b}{\gamma(\pi) \prec P{*}.b} \qquad \frac{\gamma(\pi) \prec P{*}.b \qquad p \in P}{\gamma(p.\pi) \prec P{*}.b}$$

$$\frac{\kappa \prec a_1}{\kappa \prec a_1 + a_2} \qquad \frac{\kappa \prec a_2}{\kappa \prec a_1 + a_2} \qquad \frac{\mathbf{R}(\pi) \prec a_1 \qquad \mathbf{N}_r(\pi) \not\prec a_2}{\mathbf{R}(\pi) \prec a_1 - a_2}$$

$$\frac{\mathbf{W}(\pi) \prec a_1 \qquad \mathbf{N}_w(\pi) \not\prec a_2}{\mathbf{W}(\pi) \prec a_1 - a_2} \qquad \frac{(\forall \kappa \in K)\ \kappa \prec a}{K \prec a}$$

Figure 10.2: Matching paths with access permission contracts.

the union of two access permission contracts $a + a$, or the difference between two access contracts $a - a$. A path permission b is typically a list of property name sets. The only special thing to mention is, that $P{*}.b$ is used to express that the properties $p \in P$ may be repeated arbitrary often.

To define the semantics of access contracts, it is necessary to distinguish if a path was used in a read operation, or in a write operation. Therefore, we classify access paths as read paths and write paths by writing $\mathbf{R}(\pi)$, $\mathbf{W}(\pi)$. For an access permission contract a

$$\kappa \prec a \tag{10.1}$$

defines that the classified access path κ matches the access permission contract a. The definition of the relation $\prec$ also makes use of the two classifiers $\mathbf{N}_r$ and $\mathbf{N}_w$ to define the semantics of negative access permission contracts. Figure 10.2 defines the semantics of access permission contracts with inference rules for the judgment $\kappa \prec a$. Essentially, a single path step P in a permission is matched by a corresponding property $p \in P$ in the path. An iterated path step $P{*}$ is matched by a sequence of properties from P in the path. The five axioms on top of Figure 10.2 implement the different treatments of the four kinds of paths. A write path must be matched exactly by the permission, a read path may match any prefix of the permission, and a negative read path only requires that a path prefix is matched by the permission. The latter choice is required for the implementation of the difference operator $a_1 - a_2$, where the second premise asks for $\mathbf{N}_r(\pi) \not\prec a_2$, that is, there should be no derivation of $\mathbf{N}_r(\pi) \prec a_2$. To remove the write permission from an access permission contract, without influencing the read permission, $\mathbf{N}_w(\pi)$ does also match the empty set. This definition of the matching relation enforces that the read language is prefix closed as well as that the access permission contract is prefix accessible (consider Definition 9.2.1 and Definition 9.2.1).

Lemma 10.1.1. *For all access permission contract a, the read language $L_r = \{\pi \mid \mathbf{R}(\pi) \prec a\}$ is prefix closed.*

Proof. By induction. □

Lemma 10.1.2. *For all access permission contract a, the pair (L_r, L_w) with $L_r = \{\pi \mid \mathbf{R}(\pi) \prec a\}$ and $L_w = \{\pi \mid \mathbf{W}(\pi) \prec a\}$ is prefix accessible.*

Proof. By induction. □

10.1.1 Connecting Access Permission Contract Specifications

Turning to the concrete syntax, an access permission contract for a variable x has the following general form:

$$\mathsf{with}\ [\mathsf{x}.w_1,\ldots,\mathsf{x}.w_n]\ \mathsf{except}\ [\mathsf{x}.e_1,\ldots,\mathsf{x}.e_m] \tag{10.2}$$

Translated to the formal syntax defined in Figure 8.2, this permission reads as follows:

$$a = (w_1 + \ldots + w_n) - e_1 - \ldots - e_m\ .$$

The access permission contract (10.2) corresponds to the language $\mathsf{L}_r = \{\pi \mid \mathbf{R}(\pi) \prec a\}$ of permitted read paths and the language $\mathsf{L}_w = \{\pi \mid \mathbf{W}(\pi) \prec a\}$ of permitted write paths for the variable x. Hence, adding a contract with an effect annotation to a function as in (10.2) is equivalent to surrounding the function body e with the permit expression permit x : L_r, L_w **in** e. This notation allows to remove only write permissions of π from the set L_w, without affecting L_r, by

$$\mathsf{with}\ [\ldots]\ \mathsf{except}\ [\mathsf{x}.\pi.\emptyset]\quad . \tag{10.3}$$

To make the two situations more distinguishable (in the positive specification extend a path with the empty set means, we are talking about read access, while in the negative part it means, we are talking about write access), you may use $@_r$ in the positive case and $\leftarrow$ in the negative case.

As in the formal syntax, paths of arbitrary length can be specified using the $*$ operator. For example, an access permission contract for x, x.next, x.next.next, ... for the elements of a list is written as x.next$*$. The wildcard property ? stands for the set of all property names. If the operator $*$ is used without a preceding property name, then it stands for ?$*$, specifying a sequence of arbitrary property names.

A property set can be specified in several ways. An identifier (as in x.test) or a string literal (x."foo.bar") specify singleton sets, with the string notation allowing special characters (like .) in property names. A regular expression (x./left|right/) specifies the set of properties that match the expression.

10.2 Algorithm

The inference algorithm takes a set of access paths (either a read access, or a write access) and computes a reasonable access permission contract for them. This task is a special instance of the more general problem, learning a (regular) language from a set of positive examples. Consider the following lemma:

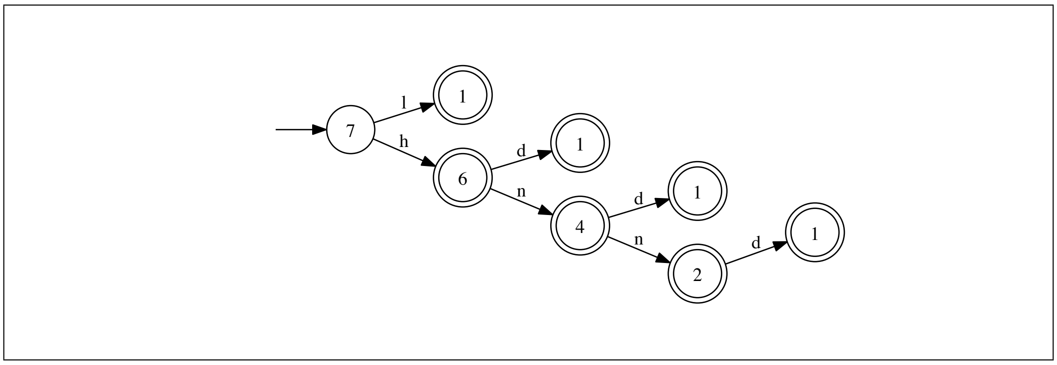


Figure 10.3: Example trie.

Lemma 10.2.1. *Let $K = \{\gamma_i(\pi_i) \mid i \in I\}$ be an finite set of access path for $\gamma_i \in \{\mathbf{R}, \mathbf{W}\}$.*

- *For $b_i = \pi_i$ it holds: $K \prec \sum_i b_i$.*
- *$K \prec *$.*

Proof. Trivial by induction over $\prec$. □

The Lemma proves that all finite sets of access paths do have two access permission contracts subsuming all paths. The access permission contract $\sum_i b_i$ is worthless, because it is too special. The access permission contract $*$ is worthless, too, because it is too general.

Because the task of finding an access permission contract subsuming all access path has these two worthless solutions, the inference algorithm has to do better. It has to find an (valid) access permission contract, which actually helps during software development or software testing. Therefore, our algorithm is based on a heuristic, which tries to compute a reasonable result for a large amount of interesting examples. The result should be similar to an access permission contract a programmer would create for the software.

10.2.1 Building the Trie

For our purposes, a trie [42] is a rooted, directed graph where each node is labeled with an integer and each edge is labeled with a property name. The trie $T(\Pi)$ represents a set of access paths Π as follows. The root node r is labeled with the number of paths $|\Pi|$. For each property p, let $p\backslash\Pi = \{\pi \mid p.\pi \in \Pi\}$ be the set of tails of paths that start with p. If $p\backslash\Pi$ is non-empty, then the trie for Π includes $T(p\backslash\Pi)$, where there is an edge from r to the root node of $T(p\backslash\Pi)$.

For example, the path set $\Pi_{list} = \{l, h, h.d, h.n, h.n.d, h.n.n, h.n.n.d\}$ is represented by the trie in Figure 10.3. The trie can also be considered a finite automaton recognizing the set Π with final states indicated by the double circles in the figure.

10.2.2 Extracting Access Permission Contracts

The goal of the extraction algorithm is to create access permission contracts of one of the forms π or $\pi.P{*}.\pi'$ where $P \subseteq \mathit{Prop}$ and π' may be empty. The initial component π is determined by computing a set of "interesting" prefixes from a set of paths Π, where π is a prefix of Π if there exists some $\pi' \in \Pi$ such that π is a prefix of π'.

The question remains what is an interesting prefix. An alternative formulation of this question is, what properties should be chosen to put into P, and what properties should be part of π and π'. Let us call P loop properties, because the set P contains those properties which are deferred inside of a loop or recursive call, because only P contains properties, which may arise in an unknown number inside a valid matching path of an access permission contract of the form $\pi.P{*}.\pi'$. Putting those properties into P which are used multiple times in some path and never in others, results in a good heuristic for identifying loop properties. The following definition reflects this intuition.

Definition 10.2.2 (Interesting Path)**.** *A path $\pi.p$ is* interesting *with respect to the path set Π, if π is interesting, and if for all properties q, the set of properties of $\pi.q\backslash\Pi$ does not contain q. All prefixes shorter than a given length are always considered interesting.*

The intuition behind this inductive definition is to filter all properties that occur in a path $\pi \in \Pi$ twice or more. If that is the case, the heuristic assumes the reason for the multiple dereferencing is a loop, which may follow the property arbitrary often. These properties should not be part of the prefix, but should be part of the set P, if the algorithm infers an access permission contract of the form $\pi.P{*}.\pi'$. The informal definition can be formally restated as

$$\frac{|\pi| < l}{\pi \in \text{Prefixes}_l(\Pi)} \qquad \frac{\pi \in \text{Prefixes}_l(\Pi) \qquad \exists\pi' : \pi.q.\pi' \in \Pi \\ \forall p \exists \pi' : \pi.p.\pi' \in \Pi \Rightarrow p \notin \text{Prop}(\pi.p\backslash\Pi)}{\pi.q \in \text{Prefixes}_l(\Pi)}$$

We define the shortcut $\text{Prefixes}(\Pi) := \text{Prefixes}_1(\Pi)$.

The implementation is done efficiently by a recursive function using a reformulation of the predicate in the inference rule.[1] Let us now consider an example to get an intuition, what is an interesting prefix. A good access permission contract for the set of paths Π_{list} is $l, h.n{*}.d$. Therefore, the set of interesting prefixes should

[1] The predicate is defined for a path and a path set

$$\begin{aligned}
&\forall p : \exists\pi' : \pi.p.\pi' \in \Pi \Rightarrow p \notin \text{Prop}(\pi.p\backslash\Pi)\\
\Leftrightarrow\ &\forall p : \exists\pi' : \pi.p.\pi' \in \Pi \Rightarrow \text{Prop}(\pi\backslash\Pi) \supset \text{Prop}(\pi.p\backslash\Pi)\\
\Leftrightarrow\ &\forall p : \exists\pi' : \pi.p.\pi' \in \Pi \Rightarrow np > llp\\
\Leftrightarrow\ &\forall p : \exists\pi' : \pi.p.\pi' \in \Pi \Rightarrow \neg(np \le llp)
\end{aligned}$$

with $np = |\text{Prop}(\pi\backslash\Pi)|$ and $llp = |\text{Prop}(\pi.p\backslash\Pi)|$.

contain l and h, and no path containing a single n. The definition of interesting prefixes yields for this example:

$$\text{Prefixes}(\Pi_{\mathit{list}}) = \{\varepsilon, l, h\} \quad . \tag{10.4}$$

A weakness of the definition of interesting prefixes is that the resulting set of interesting paths contains a lot of information twice, e.g. if a path $\pi.p$ is interesting, the definition ensures, that π is also an interesting prefix. It is important to keep both paths, if π has been added to the trie due to a write access, but it is not important to keep it, if π was only read, because $\pi.p$ will subsume this case. Therefore, at this point, we distinguish the treatment of read paths from the treatment of write path. As read paths are closed under taking the prefix, we may compute the prefix reduct by removing all paths that are proper prefixes of other paths.

$$\text{Reduct}(\Pi) = \{\pi \in \Pi \mid (\forall \pi')\ |\pi'| > 0 \Rightarrow \pi.\pi' \notin \Pi\}$$

For write paths, a more conservative reduction must be applied. Only those proper prefixes can be removed that are not members of the underlying original set. Let Π be a set of prefixes of Π_0.

$$\text{ReductW}(\Pi, \Pi_0) = \text{Reduct}(\Pi) \cup (\Pi \cap \Pi_0)$$

The reduct operation removes the empty prefix from the set of interesting prefixes in the case of read paths, and in the case of write paths. Due to $\text{Reduct}(\text{Prefixes}(\Pi_{\mathit{list}})) = \{l, h\}$, the set of suffixes are:

$$\begin{array}{rcl} l\backslash\Pi_{\mathit{list}} & = & \{\varepsilon\} \\ h\backslash\Pi_{\mathit{list}} & = & \{\varepsilon, d, n, n.d, n.n, n.n.d\} \end{array}$$

For each of these sets, we now consider the set of interesting suffixes, where "interesting" is defined in the same way as for prefixes. Technically, we just reverse all path suffixes and apply the interesting-prefixes algorithm. That is,

$$\text{Suffixes}(\Sigma) = \overleftarrow{\text{Prefixes}_2(\overleftarrow{\Sigma})}$$

where $\overleftarrow{\Sigma} = \{\overleftarrow{\pi} \mid \pi \in \Sigma\}$ and $\overleftarrow{\pi}$ are the reverse of a path π. Please note, that we choose $l = 2$ for this case, because we like to ensure, that all suffixes with a length of 1 are added to the set of interesting suffixes. The intuition behind this is, that an access permission contract $\pi.P*$ is in nearly all situations too general. It is typically better to add multiple $\pi.P*.p$ access permission contracts for the properties p, for which the accesses are realized by the program.

Going back to the example, Figure 10.4 shows the trie containing the reversed suffixes of $h\backslash\Pi_{\mathit{list}}$. From this trie, it is easy to see that the interesting suffixes of $h\backslash\Pi_{\mathit{list}}$ are $\{\varepsilon, d, n\}$, whereas there is only one respective suffix of $l\backslash\Pi_{\mathit{list}}$, namely ε.

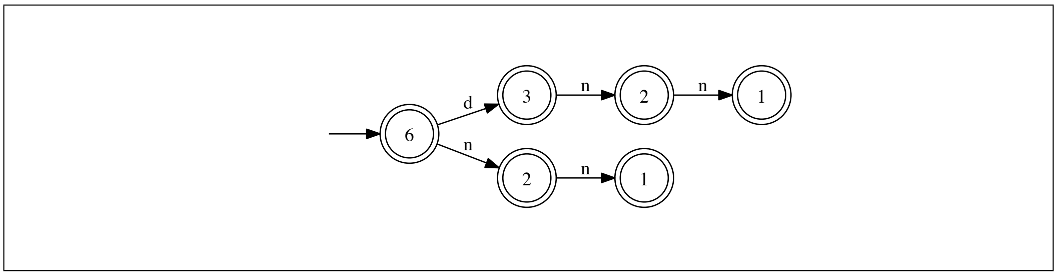


Figure 10.4: Reversed suffix trie

The final step of the algorithm considers for each pair of interesting prefix and interesting suffix the remaining part in the middle. The *right quotients* of the suffix language yield exactly this remaining part. The right quotient Π/π of a language with respect to a path π is defined dually to the left quotient by

$$\Pi/\pi = \{\pi' \mid \pi'.\pi \in \Pi\}$$

To abstract the resulting middle language, we restrict the algorithm to two choices. Either ε, if the middle language is $\{\varepsilon\}$, or $P*$ in all other cases.

In the example, we need to consider four cases, with the computation shown left and the resulting access permission contracts shown in the right column:

$$\begin{array}{lcll}
(l\backslash\Pi_{list})/\varepsilon & = & \{\varepsilon\} & \mapsto & l \\
(h\backslash\Pi_{list})/\varepsilon & = & h\backslash\Pi_{list} & \mapsto & h.\{n,d\}* \\
(h\backslash\Pi_{list})/d & = & \{\varepsilon, n, n.n\} & \mapsto & h.n*.d \\
(h\backslash\Pi_{list})/n & = & \{\varepsilon, n\} & \mapsto & h.n*.n
\end{array}$$

This result is not entirely satisfactory because $h.\{n,d\}*$ clearly subsumes $h.n*.d$ and $h.n*.n$, but the latter two permissions are more informative and thus preferable. Unfortunately, even together, they do not cover the access path h, which is only covered by $h.*$.

The source of the problem is that the set $\{\varepsilon, d, n\}$ is suffix-closed. For prefixes we apply the prefix reduction, because the semantics of access paths is prefix-closed. However, we cannot just apply suffix reduction as the example shows: If the suffix (in this case ε) is actually an element of the underlying set $h\backslash\Pi_{list}$, then dropping the suffix would be incorrect.

The solution is to treat the suffixes which would be removed by suffix reduction but which are elements of the underlying set specially and to drop the rest. The special treatment is simple: we just declare their middle language to be $\{\varepsilon\}$. With this treatment (specified in function BUILDPERMISSIONS in Program 10.1), the case $(h\backslash\Pi_{list})$ with suffix ε yields the access permission h. The function has to be called for each interesting prefix with the corresponding suffix language (function PERMISSIONSFROMPATHSET).

The final result of this phase applied to the running example is the set of access permission contracts $\{l, h, h.n*.d, h.n*.n\}$.

Program 10.1 Building access permission contracts.

```
function BuildPermissions($\pi, \Sigma$)
                                        ▷ $\pi$ is a prefix, $\Sigma$ corresponding suffix language
    $R \leftarrow \emptyset$                      ▷ result set of access permission contracts
    $\Sigma_0 \leftarrow$ Suffixes($\Sigma$)              ▷ set of interesting suffixes of $\Sigma$
    for all $\sigma \in \Sigma_0$ do
        if $\sigma$ is proper suffix of an element of $\Sigma_0$ then
            if $\sigma \in \Sigma$ then
                $R = R + \pi.\sigma$
        else
            if $\Sigma/\sigma = \{\varepsilon\}$ then              ▷ middle language is empty
                $R = R + \pi.\sigma$
            else
                $R = R + \pi.P{*}.\sigma$              ▷ $P$ is set of properties in $\Sigma/\sigma$
    return $R$

function PermissionsFromPathSet($\Pi_0, \Pi$)
                                        ▷ $\Pi_0$ set of prefixes of $\Pi$, sampled set of paths
    $R \leftarrow \emptyset$                      ▷ result set of access permission contracts
    for all $\pi \in \Pi_0$ do
        $R = R +$ BuildPermissions($\pi, \pi\backslash\Pi$)
    return $R$
```

Program 10.2 Simplification.

```
function SIMPLIFY($R, W$)          ▷ sets of path permissions (Reading, Writing)
    while $(\exists b, b')\ b \in R \wedge (b' \in R \wedge b \neq b' \vee b' \in W) \wedge \vdash b \subseteq b'$ do
        $R \leftarrow R - b$
    return (R,W)
```

Program 10.3 Overall algorithm.

```
function MAIN($\Pi^r, \Pi^w$)                    ▷ $\Pi^r$ read paths, $\Pi^w$ write paths
    $\Pi^r_0 \leftarrow$ Prefixes($\Pi^r$)                    ▷ interesting prefixes of $\Pi^r$
    $\Pi^w_0 \leftarrow$ Prefixes($\Pi^w$)                   ▷ interesting prefixes of $\Pi^w$
    $R \leftarrow$ PERMISSIONSFROMPATHSET(Reduct($\Pi^r_0$), $\Pi^r$)
    $W \leftarrow$ PERMISSIONSFROMPATHSET(ReductW($\Pi^w_0$), $\Pi^w$)
    $(R, W) \leftarrow$SIMPLIFY($R, W$)
    return $R.@ + W$
```

10.2.3 Simplifying Access Permission Contracts

The result of the previous phase is not as concise as it could be. It may still generate redundant access permission contracts. Consider the result of the example $\{l, h, h.*.d, h.*.n\}$. As this set only contains read permissions, which are closed under prefix, it follows that permission h is subsumed by $h.*.d$ and $h.*.n$, so that the result is equivalent to (the simpler set) $\{l, h.*.d, h.*.n\}$.

To perform this simplification, we first define a subsumption relation $\subseteq$ on path permissions.

$$\vdash \varepsilon \subseteq b \qquad \frac{\vdash b \subseteq P'*.b' \qquad P \subseteq P'}{\vdash P.b \subseteq P'*.b'} \qquad \frac{\vdash P.b \subseteq b'}{\vdash P.b \subseteq P'*.b'}$$

$$\frac{\vdash b \subseteq b' \qquad P \subseteq P'}{\vdash P*.b \subseteq P'*.b'}$$

This relation is sound in the sense that it reflects the semantic subset relation on sets of accepted access paths.

Lemma 10.2.3. *If* $\mathbf{R}(\pi) \prec b$ *and* $\vdash b \subseteq b'$, *then* $\mathbf{R}(\pi) \prec b'$.

Given this relation, simplification just removes all read path permissions that are subsumed by other (read or write) path permissions as specified in Program 10.2. In the example, clearly $\vdash h \subseteq h.n*.d$, so that h can be removed from the read path permissions.

10.2.4 Putting it Together

Program 10.3 summarizes the overall algorithm as it has been presented so far. The parameters that determine the length and degree for the computation of interesting prefixes and suffixes have default values that yield good results in our experiments. In addition, our implementation makes them accessible through the user interface for experimentation, on a global as well as on a per-function basis.

10.3 Special Cases

There are two special cases of property accesses that lead to extremely high branching degrees.[2] The first case is that an object is used as an array. The symptom of this case is the presence of accesses to numeric properties. Our implementation assumes that arrays contain homogeneous data and collapses all numeric property names to a single *pseudo property name* $\sharp$. This collapsing already happens when the trie is constructed from the access paths.

Similarly, an object might be used as a hash table. This use leads to the same high branching degrees as array accesses, but cannot be reliably detected at trie construction time. Instead, the implementation makes a pre-pass over the trie that detects nodes with a high number of successors, merges these subtries, and relabels the remaining edge to the merged successor trie with a wildcard pseudo property name ?.

As the rest of the algorithm does not depend on the actual form of the property names, the introduction of these pseudo property names is inconsequential.

10.4 Soundness

To establish the soundness of the algorithm, we need to prove that each element of the original path set is matched by the extracted access permission contract. The first phase, building the trie, is trivially sound. The third simplification phase is sound by Lemma 10.2.3. It remains to consider the second phase. We only examine the case for read paths with write paths handled similarly.

Suppose $\pi \in \Pi$, the initial set of access paths. As $\Pi_0 = \text{Reduct}(\text{Prefixes}_{l,d}(\Pi))$ is prefix free, there are two possibilities. Either, there is exactly one element $\pi_0 \in \Pi_0$ such that π_0 is a prefix of π, or there is at least one element $\pi' \in \Pi_0$ such that π is a prefix of π'.

In the second case, π' will be prefix of an access path $\pi'.b$ with $\pi \prec \pi'.b$.

In the first case, it remains to show that π_0 is extended to an access path that matches $\pi = \pi_0.\pi_1$. Let Σ_0 be the set of interesting suffixes of $\Sigma = \pi_0 \backslash \Pi$. By construction, $\pi_1 \in \Sigma$. We need to show that there is an element $\sigma \in \Sigma_0$ where either $\pi_1 = \sigma$ or $\pi_1 \prec *.\sigma$.

For a contradiction, suppose that neither is the case and let σ be the maximal suffix of π_1 in Σ_0 (such σ must exist). If σ is a proper suffix of an element of Σ_0 and $\sigma \in \Sigma$, then $\sigma = \pi_1$, a contradiction. If $\Sigma/\sigma = \{\varepsilon\}$, then $\sigma = \pi_1$, a contradiction. If $\Sigma/\sigma \neq \{\varepsilon\}$, then $\pi_1 \prec *.\sigma$, a contradiction.

[2] A branching degree is considered high, if it is larger than the configurable parameter HIGH_DEGREE, which defaults to 20.

Hence, all cases are matched.

11 Implementation

The contract compiler is implemented in roughly 15000 lines of OCaml (including about 2000 lines devoted to parsing JavaScript). It parses the JavaScript file, creates an abstract syntax tree of the source code, and parses the contracts from the comments of the JavaScript code. Next, the compiler analyzes the dependencies between the contracts to ensure that they are acyclic. Finally, it generates code for the contracts as well as code to connect them to the test framework. Depending on the chosen options and annotations, it additionally transforms the functions under contract by adding monitoring assertions or rewrites the code to support effect monitoring.

The test library consists of about 13500 lines of browser-independent JavaScript (including a test suite with 2500 lines). The core library of JSConTest contains about 2000 lines of code. This part contains code for contracts together with a part that manages the test cases and assertions. Another important part is an event handler module, which provides a user-configurable way of reacting to contract violations, assertions and effect violations. Additionally the JavaScript library of JSConTest provides a large number of helpful modules, for example a module that runs the test cases on a cluster of machines, such that it is easy to run all test cases on multiple operation systems and browsers in parallel, while collecting the results of all these tests at one place. This module is realized by building an interface to JSTestDriver [69], a project that supports executing JavaScript code on a cluster of machines. Another module to mention is the Delta Debugging module, which supports the minimization of counterexamples. The effect system is also realized in a module (JSConTest.effects). We provide multiple versions of the effect systems, for example a version where the reference attachment principle is replaced by a heap location attachment principle.

11.1 Compiler

The compiler's most important task is to transform contracts into JavaScript code and to monitor assertions and effects.

We illustrate the transformation of contracts into JavaScript code with an example. Program 11.2 shows the result of compiling the contract for function f from Program 11.1. To make the code easier to read, it defines abbreviations (line 1-14) for some functions from JSConTest with their type signatures and some comments. As an example consider the function setVar, which stores its second parameter in a private scope and returns it, as indicated by the id($2) contract. The compiled code is organized as a sequential application of functions to the

Program 11.1 JavaScript – Snippet from Program 8.2.

```
/*c int → int */
function f(x) {
  return 2 * x;
}
```

original function f. First the function checks its parameters against its contracts, and then the return value against its contracts, if the contracts on the parameters were successful. This work is done by function enableAsserts with type signature (fun, [string], string) → fun. The first parameter is the function to wrap, the second a list of contract identifiers (line 24), and the third is the unique function identifier (line 25). It returns a function that may fire events on assertion failures and behaves otherwise as the wrapped function. The returned function is then passed to overrideToStringOfFunction, which takes two objects and a boolean value. The first parameter is the object the toString method of which is overridden, the second is converted to the string returned by the method toString (of the first object). The boolean parameter is a flag that indicates if asserts are enabled. This makes it easier to debug the transformed source code in a browser, because the original source code of the transformed functions is available. The next step is to store the function together with its variable scope under its unique name for the testing framework. To that end the function setVar is used. After that, the following work is still left to do:

1. Create JavaScript code for contracts.
2. Store contracts under their names in the test framework.
3. Register the (function,contract) pair in the test suite, if that is not deactivated by an annotation.

The JavaScript code that creates the contract (line 33-35) builds the contract of function f from basic contracts, supported by JSConTest through the object JSConTest.contracts, e.g. Function. The contract Function takes four parameters. The first parameter is a list of contracts, defining what parameters the function except, the second is the return contract of the function, the third is an object defining what access permission contracts the function has, and the fourth parameter is a unique identifier of the function. This contract is stored inside the scope of the library using setVar. Then an object with property contract and count is created, which is used by the addContract function to store the contract together with the function in the test suite. The function addContract takes four parameters, which are the name of the contract, the function to test, a list of contracts, that should be added to the test library, and a list of contracts that should not be added.

Program 11.2 JavaScript – Compiled Code for Function f - no Effect Monitoring.

```
// (string, top) → id($2)
var sv = JSConTest.tests.setVar;

// (fun, [string], string) → fun
var eA = JSConTest.tests.enableAsserts;

// {Function: js:contract, Integer: js:contract, ...}
var C = JSConTest.contracts;

// (obj, obj, boolean) → obj
var otS = JSConTest.tests.overrideToStringOfFunction;

// (top, [{contract: js:contract, count: int}]) → id($1)
var ac = JSConTest.tests.addContracts;


var f = ac(sv("f_f0",
           // otS
           otS(// enableAsserts, that checks if x is int
               // and if an int is returned
               eA(function f (x) {
                    return (2. * x);
                  },
                  ["c_1"],
                  "f_f0"),
               // original source code of f
               function f (x) {
                 return (2. * x);
               },
               // toString shows hint, that asserts are enabled
               true)),
          // contracts
          [{contract : sv("c_1", C.Function([C.Integer],
                                 C.Integer,
                                 {pos : [], neg : []},
                                 "f_f0"))}]
);
```

Program 11.3 JavaScript – Compiled Code for Function f - Effect Monitoring.

```
// and if an int is returned
eA(JSConTest.effects.enableWrapper(function f (x) {
    return (2. * JSConTest.effects.unbox(x));
  }),
```

11.1.1 Effect Monitoring

To support effect monitoring, the compiler transforms the code of the function to monitor. Based on the formalization from Chapter 9 references have to store a map of access path to support effect checking against access permissions. Because the implementation lacks access to the underlining JavaScript engine, it cannot change the representation of a reference. Hence, we introduces boxes to store the indexed map of access path for references.

The transformation has to fulfill some properties in order to support calls between transformed code and untransformed code and vice versa. It is not an option to forbid one of the call situations, since it is very often the case that a function is registered as an event handler. In this case the browser directly invokes transformed code. Similarly it is not an option to forbid transformed code calling native browser functions (which are obviously not transformed). Because we try to avoid adding additional properties to functions (each additional property may be observed by untransformed code), some consequences from these requirements are:

- Parameters are generally unboxed before calling a function.
- Transformed code needs access to the boxes of their parameters, if these exists.

If the function from Program 8.2 is compiled with effect monitoring enabled, the result will be similar to Program 11.2, except the function body is generated as in Program 11.3. The function f can handle the situation that x is a wrapper object. The second change is that the function gets an additional wrapper which manages the box and unbox operations and that registers the corresponding access permission contracts, when the function is called.

For the effect monitoring the transformation rewrites property reads, property writes and function calls along with some other adjustments (see Figure 11.1).

11.1.1.1 Function calls

This section considers a code fragment with a recursive function (Program 11.4). It demonstrates the interplay of wrapping, unwrapping, and access permission contracts. Given a linked list, the function returns the value (v) of the last node after recursively following the references of the next pointers (n). For expository

$[\![e_1[e_2]]\!]$	$\rightsquigarrow$	pRead($[\![e_1]\!]$,$[\![e_2]\!]$)
$[\![e_1[e_2] = e_3]\!]$	$\rightsquigarrow$	pAssign($[\![e_1]\!]$,$[\![e_2]\!]$,$[\![e_3]\!]$)
$[\![$ f(e_1,...,e_n) $]\!]$	$\rightsquigarrow$	fCall(f,[$[\![e_1]\!]$,...,$[\![e_n]\!]$])
$[\![$ o.m(e_1,...,e_n) $]\!]$	$\rightsquigarrow$	mCall(o,"m",[$[\![e_1]\!]$,..., $[\![e_n]\!]$])
$[\![$ **new** O(e_1,...,e_n) $]\!]$	$\rightsquigarrow$	cCall(O,[$[\![e_1]\!]$,..., $[\![e_n]\!]$])

$[\![$ **for** (**var** i **in** o) { e } $]\!]$	$\rightsquigarrow$	**for** (**var** i **in** o) {
		  if (mCall(o,"hP",[i])) { $[\![$ e $]\!]$}
		}

Figure 11.1: Rewrite rules for code with enabled effect monitoring. We instrument the original source code from the left such that we can monitor property reads, property writes and creation of new objects. We also rewrite function calls to register contracts and access permission contracts.

Program 11.4 Example of a transformation with effect monitoring enabled. The transformation algorithm ensures that the name of the contract (c_1 in the example) is unique. The example presents only the transformation of the source code of the function f, not the code that we generate to create the contract.

```
/*c (any) → any
   with [x.n./n|v/.@] */
function f(x) {
  if (x.n) return f(x.n);
  if (x) return x.v;
  return x;
};
```

$\overset{partial}{\rightsquigarrow}$

```
eA(enableWrapper(function _f(x) {
     if (pRead(x, "n")) {
        return fCall(f, [pRead(x, "n")]);
     }
     if (x) return pRead(x, "v");
     return x;
  },
  ["c_1"]), ...
```

purposes, the function's implementation does not abide by its contract. Suppose that the code under test contains the following call to f:

```
var x = { v: 5, n: { v: 11, n: { v: 24, n: undefined } } };
f(x);
```

To enable the framework to locate the permissions that need to be respected by read and write operations in the function body, the access permission contract for f is stored under its time stamp in a global effect store when the function is called. At this point, the effect store contains { 0: x.n./n|v/.@ }. Then the object wrappers for the parameters are created. For parameter x, the wrapper is given by

```
{ ref: { v: 5, n: { v: 11, n: { v: 24, n: undefined } } }, pmap: { 0: x } }
```

The pmap property stores the information that in the (outermost) function call which is given the uid 0, the access path for the object is x. Now the execution of the actual function body commences with the expression pRead(x,"n"). Comparing the actual access path x.n with the permission x.n./n|v/.@ causes no violation, so the property access is granted, yielding a reference to {v: 11, n: { v: 24, n: **undefined** } }, which is returned in a wrapper as

{ ref: {v: 11, n: { v: 24, n: **undefined** } }, pmap: { 0: x.n } } .

Performing the recursive call with this object as parameter creates a fresh uid 1, and extends the wrapper for the parameter to

{ ref: {v: 11, n: { v: 24, n: **undefined** } }, pmap: { 0: x.n, 1: x } } .

Additionally, the global effect store is extended to { 0,1: x.n./n|v/.@ }. When reading the property n again, the read access is admitted, and the wrappers are again extended as in the previous case when calling the recursive function again. The parameter x is now wrapped as

{ ref: {v: 24, n: **undefined**}, pmap: { 0: x.n.n, 1: x.n, 2: x }} .

Finally, when f tries to read n, the access to the property is not granted because the access path for uid 0 only grants access to {x.n./n|v/.@ }, but the read operation tries to dereference {x.n.n.n }. The result ist an access violation.

11.2 Library

11.2.1 Test Cases and Assertions

As an alternative to using the syntax introduced in Figure 8.1, the library may also be used to construct contracts manually. For that reason (and also to simplify compilation), the library contains many predefined contracts, for instance Top, Null, Undefined, Boolean, True, False, String, Number, Function, Object, PObject, AInteger. Most of them are self-explanatory, but PObject and AInteger ask for further explanation. PObject is a contract that takes an array of property names as an argument. It implements object contracts that rely on information generated by the label analysis of the contract compiler. The behavior of AInteger is similar.

As a concession to the dual role of contracts, any object implementing a primitive contract provides two methods, one to check if a value adheres to the contract and another to randomly generate a value that is guaranteed to adhere to the contract.

11.2.2 Custom Contracts

The library contains a number of operations on contracts, for example, a union and an intersection operation as well as operations for writing contracts from scratch. There are many contracts for describing objects. All of them offer different ways to specify details of object shapes by restricting the set of allowed properties, or by restricting the corresponding values, etc. An example is an object contract restricting the allowed property names to characters and digits (EObject). It is also possible to specify recursive contracts for objects by using names as follows:

```
Let ("o", EObject[{name: "m",
                   contract: Function ([Name ("o")], Boolean)}])
```

Let binds a contract to a name and this contract can be referred to with the Name function. EObject constructs an object contract from an association list of property names to contracts. The resulting contract describes an object with an m method, which excepts an object of the same kind as its argument. Due to Let-contracts, contracts may contain cycles. To avoid nontermination of the checking algorithm, we assume that c.check(v) holds, if we encounter it a second time inside the recursive call. This design choice results in an inductive interpretation of the Let contracts.

Currently, this facility is not reflected in the surface contract language, because it is designed with simplicity in mind.

12 Evaluation

In this chapter, we evaluate the contract system, the access permission contract system and the access permission contract inference algorithm of JSConTest. We split the chapter into three sections. Section 12.1 presents case studies with the main focus on the contract system. Section 12.2 concentrates on the evaluation of access permission contracts. The third section evaluates how well the inference algorithm of access permission contracts works.

All case studies in the first section work analogously. We start from JavaScript source code and annotate the code with contracts. We validate the contract by manual inspection and extensive testing. After establishing contracts we apply mutation testing [5, 6] to the source code. In all case studies the mutator swaps identifiers, boolean values and operators (e.g. ===, ==, !=, !== or +, −, ∗, \.) The actual choice of the applied mutators depends on the source code, for example if the source code contains float constants they are mutated, too. The result of the mutator application is a set of mutated versions of the original program. The mutation testing approach then results in a percentage indicating how many of the mutated programs were rejected by JSConTest, and how many pass all contacts. The percentage is a good measure for the effectiveness of our approach. A high detection rate gives evidence that (guided) random testing and the monitoring approach are capable of detecting a high percentage of programming errors. A low detection rate lets us conclude that the dynamic approach of JSConTest is not well suited, because it lacks the facility to detect errors reliable.

Our procedure is similar in the second section, but we provide, in addition to contracts, access permission contracts for the programs. Hence, all case studies we perform for the evaluation of access permission contracts also give data about the efficiency of contracts.

In the third section our evaluation approach is different. To measure the effectiveness of access permission contract inference we cannot apply a purely automatic approach. The reason for that is already explained Chapter 10. There is no measure for the best access permission contract and an important aspect of the inference is, how simple and accurate the access permission contract is. The best evaluation we can provide is to let the access inference compute the access permission contracts for our case studies from the second section. We can then compare the manually created access permission contract and the inferred access permission contracts. If the inference algorithm is capable of computing the same or similar access permission contracts then the manually created access permission contracts, then the inference algorithm is able to save a lot of development time and is therefore worth using.

categories	(#)	(%)
accepted mutant	67	9.6
rejected mutant	628	90.4
total number of mutants	695	100.0

Table 12.1: Testing random mutations of the Huffman case study.

12.1 Contracts

In our case study to assess the efficiency contracts we do not use access permission contracts. We implement a Huffman decoder in JavaScript and specify its interface with the contract system. Writing this decoder is not difficult, but nevertheless JSConTest discovered errors during the development. Usually after just one or two test runs the test framework found counterexamples. Each time the counterexample simplified the debugging of the code significantly.

After some iterations of fixing small errors in the functions under test, we found an error in the specification. The contract js:ht is a custom contract for Huffman trees. A Huffman tree is either a node or a leaf. A node then contains two Huffman trees as children. Leaves and nodes both contain additional information, which can be ignored for now. The problem is that a valid Huffman tree has to have a depth greater than zero. If the tree consists of only one leaf, no bits are consumed by the decoder to generate an output string and the decoder enters an infinite loop. We discovered this bug when our framework generated a Huffman tree of the form HuffmanLeaf {s: '', w: ...}. This input results in a stack overflow, which is reported by our framework as a failing test. After we fixed this specification error the system works correctly and has correct contracts attached to each function.

The mutator swaps makeHuffmanLeaf and makeHuffmanNode, [left,right], [true,false], [0,1], and [===,==,!=,!==], [-,+]. The mutator randomly determines the number of modifications to apply and repeatedly applies a modification at a random place in the program.

Our test run generated 695 mutations of the original huffman.js, which were all submitted to the contract compiler and then the resulting test runs were analyzed.

There were 67 mutants (about 10% of all mutants) for which all contracts passed successfully. All other mutants were rejected outright by our test suite. The average run time of each file in the browser is 5 seconds.

It is not surprising that there are modified versions for which all tests pass, because some of the randomly chosen modifications (e.g., swapping left and right) create mutants, in which everything is correct from a typing perspective. For these 67 mutants, we verified manually that the modifications of these files are not observable at the type level, for example if a contract states Top → bool and inside the function body **return** true; is changed to **return** false;, the contract is fulfilled by the original and the modified version. Finding such bugs is either the job of a hand-written test suite or the task of a more detailed specification.

Program 12.1 JavaScript – Custom contract ht — check for a Huffman tree.

```
function generate() {
  function genRandomLeaf() {
    return makeHuffmanLeaf(TESTS.genStringL(1), TESTS.genNInt(0,1));
  };
  function genRandomNode(l,r) {
    return makeHuffmanNode([],TESTS.genNInt(0,1),l,r);
  };
  function cdes() { return "makeHuffmanNode"; };
  var gN = { getcdes: cdes, arity: 2, f: genRandomNode};
  return TESTS.genTree(isHuffmanTree,[genRandomLeaf],[gN],0.5,true);
};
var gen = TESTS.restrictTo(isHuffmanNode,generate);
var ht = new TESTS.newSContract(isHuffmanTree,gen,"HuffmanTree");

/*c Top → bool */
function isHuffmanLeaf(v) { /* function body here */ };

/*c Top → bool */
function isHuffmanNode(v) { /* function body here */ };

/*c Top → bool | js:ht → true ~noAsserts */
function isHuffmanTree(v) { /* function body here */ };
```

As the generated tests rejected 90% of the mutations and found 100% of the type errors, we argue that our testing framework detects a type error with very high probability in this case study.

12.2 Contracts and Access Permission Contracts

Our next case studies utilize contracts and access permission contracts. For each method from the source code a person without prior knowledge created the contract and access permission contract for the methods. The access contracts were created by inspecting the source code manually to identify the set of properties a method may access. Next, we applied mutation testing to the code to get statistical information about the effectiveness of the contracts and access permission contracts. We derived mutants and tested each of them against the contracts. The mutation operations were renaming of variables and properties, replacing a float constant by another, swapping basic values and replacing a binary operator with another. For each mutant, each method was tested with 1000 randomly generated test cases. To assess the effectiveness of access contracts, we executed the mutants with activated and deactivated access permission contracts.

If a mutant had violated a contract, it was categorized as rejected. If a mutant passed all 1000 test runs for all six methods, it was sorted into the fulfilled category. The following lines break down the mutants in six categories explaining why they were rejected:

contract failure The mutant was rejected due to a contract failure, for example the method returned an invalid value.

signaled error The mutant was rejected due to an error, for example because it access a property of **undefined**.

browser timeout The mutant was rejected, because it did not terminate after a timeout of 15 seconds. The average run-time for a test case was less than 10ms.

read violation If the code violated the access permission contract due to a read of a property, the mutant was added to this category.

write violation If the code violated the access permission contract due to a write of a property, the mutant was added to this category.

read/write violation If a mutant violated an access permission contract either by a read or a write of a property, it was added to this category. This category is a union of the read category and write category.

Testing a contract (and access permission contract) of a function was stopped, if violation of either one was detected. Therefore, a method could only add a mutant to one of the categories (except the read/write violation categorie, which is a union of the read and write category). But, since each mutant contained multiple methods, a mutant could be added to multiple categories. For example if a mutant violates its contract in one of its methods, it could violate its access permission contract of another method due to a read access and its access permission contract of a third method due to a write access. We count such a mutant under the category contract failure, under the category read violation, under the category write violation and under the contract read/write violation. Of course we never count a mutant multiple times in a category, even if a corresponding violation occurred in multiple methods.

12.2.1 Singly-Linked List

Our case study examines a small third-party library (200 LOC), which implements a singly-linked list (SLL) data structure [120]. The library contains one constructor for list nodes and six functions for lists. The functions are all stored in a prototype object for the singly-linked lists. Therefore, the list is passed on to all of them implicitly, and they are typically used as methods. Contracts for methods have the structure

$$\mathsf{class.}(\mathsf{p}_1,\ \ldots,\ \mathsf{p}_n) \to\ \mathsf{p}_r$$

For the singly-linked-list implementation the class contract is the custom contract js:ll. It performs an instance of check to decide, if a value is a singly-linked list. It also uses a hand-written random generator for singly-linked lists (16 LOC). The library contains six methods to operate on lists: add, item, remove, size, toArray, toString.

Table 12.2 presents the results from the test runs. JSConTest rejects already 82% of the mutants if only the contract system is activated, while the rejection percentage raises to 87%, when the access permission contract system is activated additionally, which is an increase of 6%. The number for contracts without access contracts is comparable to the 88% of rejected mutants measured in the Huffman decoder. Therefore, we take this case study as additional evidence that it was the right decision to use random testing to analyze contracts for JavaScript. It is also remarkable that access contract increases the amount of rejected mutants.

The amount of mutants rejected by a contract failure is significant. Therefore, we conclude that the specification given by our contracts is tight.Since these contracts are the source for our random generators we also set out to investigate the behavior of access permission contracts with less tight contracts.

To this end we created a third configuration for the singly-linked list case study. It employed more general contracts together with the same access permission contracts that had been used earlier.

Table 12.3 contains the results for this configuration. It turns out that the detection rate is also fairly good, even if the number of mutants rejected by a

[1] one mutant may be counted multiple times

categories	only contracts #	only contracts (%)	contracts and access permission contracts #	contracts and access permission contracts (%)
accepted mutants	1011	18.0	711	12.7
rejected mutants	4607	82.0	4907	87.3
total number of mutants	5618	100.0	5618	100.0

reasons to reject a mutant	only contracts #	only contracts (%)	contracts and access permission contracts #	contracts and access permission contracts (%)
contract failure	2020	43.8	1643	33.5
signaled error	2034	44.2	2136	43.5
browser timeout	553	12.0	243	5.0
read violation	-	0.0	1018	20.7
write violation	-	0.0	1606	32.7
read/write violation	-	0.0	1842	37.5

Table 12.2: Mutation testing of the SLL case study with tight contracts. The column “only contracts” counts mutants depending on test runs with deactivated access permission contract system. The column “contracts and access permission contracts” counts mutants depending on test runs with activated access permission contract system. The second part of the table presents the reason for the rejection of a mutant. The percentage is computed with respect to 4607 and 4907, the total number of rejected mutants per column. The entries does not sum up to 100%, because there might be multiple reasons for a rejection of a mutant.

contract failure drops dramatically. Access permission contracts are able to catch many of these mutants.

Of special interest are the cases where the contract system did not detect the mutation of the code as these cases indicate the effectiveness of the annotations. A manual inspection of these mutants revealed that in many cases the mutated code is semantically equivalent to the original version, for example `x.p` was changed to `x.q`, where both properties `p` and `q` were always undefined. In other cases, the contract was fulfilled by a mutant because the modification did not change any property access or return value from a type perspective, for instance **return** `true` was changed to **return** `false`. While these mutants may violate the intended semantics, we cannot have higher expectations, because we started from a partial specification, not from a full specification of a linked-list data structure.

We also manually inspected ten randomly selected mutants that timed out. All of these ten mutants timed out because of an infinite loop. Hence, we have reason to believe that our choice of timeout is sensible.

categories	contracts and access permission contracts	
	#	(%)
accepted mutants	1055	18.4
rejected mutants	4721	81.6
total number of mutants	5776	100.0

reasons to reject a mutant	contracts and access permission contracts	
	#	(%)
contract failure	1096	23.2
signaled error	2243	47.5
browser timeout	369	7.8
read violation	1004	21.3
write violation	1593	33.7
read/write violation	1823	38.6

Table 12.3: Mutation testing of the SLL case study with general contracts. All numbers are collected with activated contract system and activated access permission contract system. The first part of the table presents the number of accepted and rejected mutants. The second part of the table presents the reason for the rejection of a mutant. The percentage is computed with respect to 4721, the total number of rejected mutants. The entries does sum up to more than 100%, because there might be multiple reasons for a rejection of a mutant.

12.2.2 Google V8 - Richards and Deltablue Benchmark

Our largest case study is based on the JavaScript V8 benchmark from Google.[2] The first part is the Richards benchmark, which implements 29 functions in 650 LOC. It simulates a task dispatcher of an operating system. The benchmark was originally created by Martin Richards and was reimplemented by Google in JavaScript.[3] A person without prior knowledge about the code developed the contracts and access permission contracts for the functions. The main part of this work was to write custom generators for contracts to achieve a high statement coverage. For mutation testing we create about 2950 mutants. Each mutant executes each of the 29 functions 50 times. Table 12.4 presents the results of the test runs. The number of executions per function was picked small to keep the execution time low. Introducing access permission contracts increases the detection rate from 61.1% to 69.2%, which is an improvement by 13%.

The second part of the V8 benchmark that we examined is the Deltablue con-

[2] `http://v8.googlecode.com/svn/data/benchmarks/v6`

[3] `http://www.cl.cam.ac.uk/~mr10/Bench.html`

categories	only contracts #	only contracts (%)	contracts and access permission contracts #	contracts and access permission contracts (%)
accepted mutants	1148	38.9	911	30.8
rejected mutants	1807	61.1	2044	69.2
total number of mutants	2955	100.0	2955	100.0

reasons to reject a mutant	only contracts #	only contracts (%)	contracts and access permission contracts #	contracts and access permission contracts (%)
failed contracts	872	48.3	866	42.4
signaled error	1052	58.2	1037	50.7
browser timeout	28	1.5	30	1.5
read violation	0	0.0	202	9.9
write violation	0	0.0	149	7.4
read/write violation	0	0.0	349	17.1

Table 12.4: Testing random mutations of the Richards case study. We perform the richards benchmark once with deactivated access contracts (column “only contracts”) and once with activated access contracts (column “contracts and access permission contracts). The second part of the table presents the reason for the rejection of a mutant. The percentage is computed with respect to 1807 and 2044. The entries does not sum up to 100%, because there might be multiple reasons for a rejection of a mutant.

straint solver. The Google implementation is derived from a Smalltalk implementation by John Maloney and Mario Wolczko. It contains 59 functions and constructors in 670 LOC. In this implementation building up the constraint model is done by side-effects of the constructors. Therefore, the unit testing approach, as used in the other cases studies, is not easily applicable. The work needed to set up the environment, for example some parts of the code rely on initialized global variables. Instead of writing additional initialization code, we saved the work by using the main method of the benchmark itself, which sets up the necessary environment. Therefore, only the monitoring facility of the contracts was utilized to perform the mutation testing. The main challenge during the creation of contracts was to figure out the object hierarchy. This work was done by a person without prior knowledge of the benchmark. The access permission contracts were automatically inferred by JSConTest, and added to the functions automatically. A manual inspection of the inferred access permission contracts provides evidence, that the access permission contract inference works well. Table 12.5 shows the result of executing about 830 mutated versions of the benchmark with deactivated contract and access permission contract monitoring. Contract monitoring was deactivated,

categories	only asserts		access permission contracts	
	#	(%)	#	(%)
accepted mutants	176	21.3	102	12.8
rejected mutants	626	75.6	697	87.2
total number of mutants	802	100.0	799	100.0

reasons to reject a mutant	only asserts		access permission contracts	
	#	(%)	#	(%)
signaled error	505	80.7	469	67.3
browser timeout	26	4.2	29	4.2
application exception	121	19.3	73	10.5
read violation	0	0.0	135	19.4
write violation	0	0.0	20	2.9

Table 12.5: Testing random mutations of the Deltablue case study. We perform the benchmark once without any contract checking and once with access permission contracts activated. The second part of the table presents the reason for the rejection of a mutant. The percentage is computed with respect to 626 and 697. The entries does sum up to 100%, because there exists only one reason to reject a mutant.

because the efficiency of access permission contracts should be monitored. The benchmark contains some internal checks of invariants, which result in an application exception. They create a reason to reject a mutant that depends not on the contract or access permission contract system. Hence we report them in Table 12.5 in its own row. In the Deltablue benchmark adding access permissions increased the detection rate from 75.6% to 84.2%, which is an improvement of 11.34%.

12.3 Access Permission Contract Inference

The effectiveness of access permission contract inference is another interesting thing to investigate. JSConTest is capable of inferring access contracts of functions as follows. To obtain a first impression what properties are accessed by the different functions, it is sufficient to suffix an empty access permission contract to the contract

```
/*c js:ll.(top) → undefined with [] */
```

for the add function. This augmented contract states that the function with this contract is not allowed to read or write any property. Extending the remaining functions' contracts in the same way and applying the JSConTest compiler results in instrumented code that monitors all property accesses.

If the compiled code executes in a browser, then – besides its intended functionality – it collects a list with tens of thousands of property accesses which all violate the empty effect annotations. From this raw data, our effect inference computes concise access contracts. For example, the computed effect for add is

```
[this._head,this._head.*.next,this._length]
```

which means that add only accesses objects via its this pointer, it reads and writes the _head and _length properties, and it reads and writes a next property that is reachable via _head followed by an unspecified sequence of properties as indicated by $*$. Technically, all three path permissions are write permissions that implicitly permit reading all prefixes of any path leading to a permitted write. Our inference algorithm also determines that, in this case, $*$ stands for a sequence of next accesses, but this extra information is outside the scope of the file path syntax.

The computed effect for remove is also interesting:

```
this._head.*.data.@,this._head.*.next,this._length
```

The function remove removes a given value from the list. To this end, it compares this value with all data properties reachable via _head and a sequence of (all next) properties, as indicates with the first access path. Its ending in @ indicates a read-only path. Furthermore, remove changes next pointers and modifies the _length property of this.

We also applied the access permission inference algorithm to various other data structures like doubly-linked lists and binary search trees. It turns out that the quality of the computed access permission contracts is comparable to manually created access contract permissions.

13 Related Work

13.1 Testing Based on Interface Specification

The QuickCheck library [21] utilizes the type signatures that are available in the programming language Haskell, a purely functional language, to randomly test properties that a programmer specifies. The programmer can write Haskell functions to express the properties a Haskell program should fulfill. In QuickCheck, the programmer can also define his own generators to increase coverage. JSConTest uses type contracts instead of Haskell type signatures to derive its test cases. Type contracts are extensible. The programmer can write JavaScript functions to specify arbitrary complex properties, as it is possible in QuickCheck. The programmer is also capable of writing his own random generators for type contracts in JSConTest. Therefore, he is able to increase coverage similar to QuickCheck. A feature that QuickCheck does not offer is to specify what side effects functions are allowed to perform. JSConTest offers access permission contracts to solve this problem, which is important in JavaScript. Both, QuickCheck and JSConTest support test case minimization. JSConTest supports delta debugging [121, 122] and hierarchical delta debugging [83].

DoubleCheck [31] is an adaptation of QuickCheck to the ACL2 language. It is implemented in the Racket[1] programming environment [36, 101]. If the ACL2 theorem prover cannot verify a property of a program, DoubleCheck generates counterexamples for this property. The developer's next step is to restate the property and try a next verification run. Because of the counterexamples the developer has a good information at hand why the property does not hold for the program. PLT-Redex also comes with a random testing facility that has detected errors in semantic specifications [72].

JCrasher [25] uses a random testing approach to find crashing program fragments. It randomly creates code snippets that contain a sequence of method calls from a set of Java classes. JCrasher ensures that a method that depends on a parameter is called only if the parameter is created by another method call earlier in the snippet. In JCrasher a failure is a program crash. This approach makes JCrasher a fully automatic testing tool that does not depend in any way on contracts, as JSConTest does. This benefit is also a weakness, because JCrasher cannot test the program for user-specific contracts. Additionally, JSConTest provides a limited amount of white-box testing by collecting data to perform guided random testing. Randoop [94] is a tool for directed random testing of Java classes.

[1] The Racket teaching languages also provide QuickCheck-style testing of contracts.

It generates test cases in a similar way as JCrasher, but additionally uses the test outcomes as feedback to avoid creating useless or outright erroneous tests. An approach based on Randoop for JavaScript is the tool Artemis [8]. It seems easy to combine Artemis and JSConTest, because JSConTest is based on a source-to-source transformation and Artemis is implemented as a modification to the Rhino JavaScript interpreter. As a benefit of the combination one gets the a high coverage of an automatic test suite generated by Artemis, especially a test suite that covers event handlers, and the expressiveness of contracts and access permission contracts at the same time.

RUTE-J [6] is a **R**andomized **U**nit **T**esting **E**ngine for **J**ava. It helps the programmer to add some portion of randomness into his unit test suite. RUTE-J can randomize a list of method calls as well as input data and it performs minimization of tests and the test suite, such that the suite fulfills a coverage criteria.

The ARTOO system [19] performs adaptive random testing (ART) for Eiffel. It is integrated into AutoTest that helps developers create, manage, and execute software tests. It adapts previous ideas from the ART approach to an object-oriented setting. Its underlying idea is that tests are more effective if they evenly cover the parameter space of the method under test. Its execution requires a distance metric on the input values. Even if a large amount of researchers propose that ART is a technique to enhance the random testing approach, recent empirical work [7] shows, that the amount of time used to compute the distance metric for the input is often so expensive, that ART does not perform well. Often random testing without any distance metric is faster and better in finding faults than ART. JSConTest uses only simple modifications to the random testing approach that do not increase the computation time for a new test case. The criticism against ART does not apply to JSConTest. The Evotec system [109] is a tool for Eiffel that uses genetic algorithms to create a good testing strategy. This approach is evaluated with respect to the amount of time used to compute the test input and to run the test cases. The approach is more efficient than the random+ strategy [20] and the precondition-satisfying strategy [118], both available in AutoTest.

The DART system [44] follows an idea named concolic testing: It uses information that it collects during a concrete and a symbolic execution of the unit under test. Based on the collected information it generates symbolic predicates that encode conditions for the path taken by the unit under test in its concrete execution. By falsifying these predicates a theorem prover systematically generates input data to test all possible branch alternatives. In general, it is not possible to falsify the predicates for complicated programs. But if the theorem prover is successful, the result is a set of tests with perfect branch coverage. In contrast, JSConTest collects its information about unit under test by a static analysis. This is possible because the information needed by the guided random testing approach of JSConTest is simple enough.

Based on the small scope hypothesis [63] – the set of test inputs within a some small scope finds a high proportion of bugs – the tool Smallcheck [108] for Haskell and the tool Korat [15] for Java systematically execute the unit under test with all inputs below a certain size threshold. The idea is compatible with JSConTest, but it is not implemented in JSConTest right now.

13.2 Unit Testing Tools for JavaScript

There are many tools available to simplify the automatic execution of unit test, for example JSUnit [74] or Jasmine [73]. JSUnit is a JavaScript port of JUnit [11] that is not maintained anymore and replaced by Jasmine, which is a behavior-driven tool for testing JavaScript code. JsTester [2] supports the execution of JavaScript code inside of Java, which is useful in projects that use Java and JavaScript. FireUnit [105] is an extension of FireBug [38], which is itself a Firefox Add-On that facilitates debugging of JavaScript inside of Firefox. Rhinounit [115] executes the test suite outside of a browser, which yields better performance, but often as a downside, runs the test in another environment than the production code. Other tools to support debugging and testing in JavaScript are JSCoverage [40] and JSMock [28]. All these frameworks have in common that they simplify the work of manual unit testing. In contrast, JSConTest requires interface annotations in form of contracts. In practice, the two approaches complement one another, and none can replace the other.

13.3 Monitoring Tools

AdJail [79] is a tool that allows to specify what part of the DOM a third party JavaScript code fragment is allowed to read or modify. The web page developer specifies the right a fragment gets by policy attributes in the DOM. For example write−access: subtree specifies that third party code is allowed to modify the subtree of the HTML element at which the attribute is set. Access permission contracts are not applicable in this scenario in their current implementation, because they concentrate on side effects on the heap, not on the DOM. But it is just a matter of work to support also the protection of HTML elements by access permission contracts.

AjaxScope [71] is a tool capable of monitoring the execution of JavaScript code remotely. It is based on a proxy server that performs on the fly parsing and instrumentation. That makes it possible to reduce overhead because the proxy server can be configured to just annotate specific aspects of the JavaScript code. It is also used to spread tests among users, such that each user does not notice the test and annotation overhead. It might be possible to combine the monitoring facilities of AjaxScope and JSConTest for example to remotely monitor property access or detect remotely type errors and create short summaries about these errors.

Jim, Swamy and Hicks [67] modify the browser to register a "security hook" function that decides whether a script is executed. They call the strategy BEEP (Browser-Enforced Embedded Policies). Example security hooks they implement are a DOM sandbox and a whitelist for JavaScript source code. JSConTest can benefit from such security hooks in order to ensure safe monitoring of code executed by eval. It is for example possible to write a predicate that checks if javascript source code is transformed by JSConTest. This predicate could then be installed as a security hook similar to the whitelist.

13.4 Access Permission Contracts

13.4.1 Static Approaches

The book Principles of Program Analysis [88] states that effect systems are used to describe how a function, beside its return value, interacts with its environment. Gifford and Lucassen [43] invented effect systems to describe and infer side effects. Their goal was to improve memory management and to detect code fragments that may be executed in parallel. Typically, effect systems are integrated in static type systems. In contrast, our system is a dynamic system, which uses monitoring to enforce the access policy.

As an example for a static effect system we pick the work of Greenhouse and Boyland [46]. Their system uses effect annotations to facilitate compiler optimizations. Effects are collected for regions, which are provided by the programmer by annotations, to statically ensure that two method invocations manipulate different parts of the heap. In contrast, our system does not target these optimizations. Our prime goal is to detect software defects. Our approach is a dynamic approach based on monitoring that does not provide a static guarantee about absence of side effects. Consequentially, our system provides a more flexible and precise specification language. Access permission contracts for example support the star operator to describe that a method changes a property value inside of the heap following a set of properties arbitrary often.

The static effect system of Smans and coworkers [110] relies on the Z3 SMT solver [26] to verify that a method only accesses the part of the heap specified by a predicate. The argument to the predicate is an access path. Therefore, the difference to our access permission contracts – besides the fact that JSConTest is a dynamic approach – is that an access permission contract supports the star operator to express much more complicated access conditions than the acc predicates. An interesting observation is, that the static effect system agrees with our interpretation of the path-based semantics, and that they also use a last writer wins principle to verify their correctness. In the static work the connection to the Hoare Calculus [58] rule for assignment is much easier to see than in our dynamic work.

Access permission contracts are related to work on ownership and aliasing control. Typically, an ownership system is a static analysis that imposes the ownership structure on the object graph. In most cases, the ownership graph is a tree [1, 30, 89, 123]. Recent work [17] presents an ownership type system that give up the constraint ownership as dominator and imposes a directed acyclic graph on the object graph. There also exists work on a dynamic ownership system [16]. The main difference between ownership systems and access permission contracts is that our system imposes path-based access restrictions.

Bierhoff and Aldrich's work "Modular Typestate Checking of Aliased Objects" [14] presents a static system to check protocols for safe object access in Java. They model object protocols as finite state machines. The system uses splitting and joining rules for references based on fractions. Only if the system ensures that a reference is unique, a type change for the underlining finite state machine is granted.

Finifter and co-workers [37] design a JavaScript heap analysis framework to detect information leaks. They state that a third of the Alexa US Top 100 web sites are exploitable by an ADsafe-verified advertisement. To prevent exploits, third-party code is restricted to a name space by prefixing properties with a unique identifier. This restriction is enforced by a static verifier. Our approach based on access permission contract supports a more flexible approach because the specification which part of the heap is accessible is more precise than just introducing a white list of allowed property names. It is an open question whether the flexibility provided by JSConTest is necessary.

13.4.2 Dynamic Approaches

Run-time monitoring is a well-known approach to provide safety and security guarantees. Erlingsson [34] provides an overview of such applications. JSConTest does not focus on security issues. Its goal is to increase the productivity of software development in JavaScript. But access permission contracts, especially in their location-based interpretation, might serve well in a security setting context. Location based access permission contracts offer the possibility to prohibit access to specific locations in the heap. But the current implementation of JSConTest is not capable of establishing the guarantees a location-based access permission contract needs due to various reasons. First, it does not observe property access done by dynamically executed code if the code is not transformed by the off-line compiler of JSConTest. Second, the interface to the library used to implement JSConTest ist open, and not protected. Hence, malicious third party code just can use the public interface of JSConTest's effect library to modify its behavior.

Many other works follow a transformation-based approach to establish safety and security features in scripting languages, in particular in JavaScript. The reason for the approach is that with a transformation-based approach no adjustment of the browser is necessary. That is huge benefit, because it supports the establishment of the security and safety features on actual browsers. BrowserShield [104] is a project that restricts critical operations according to a user policy by a code transformation. The Google Caja project [45] employs an online compilation process of JavaScript code to a safe subset named Cajita. Maffeis and co-workers [80] combine several isolation techniques for restricting heap accesses of third-party code. They disallow eval, Function, and constructor within untrusted code and also rewrite property accesses with wrappers to enable run-time checks. They also present a set of expression templates that defines precisely by which expressions security risks arise. For example, the expression "a" + e implicitly executes the function toString of the object that is returned by the expression e, even if the property name toString is not mentioned in the code. The work enumerates all property names that may be implicitly accessed by JavaScript expressions. The combination of the facts presented in the work and our access permission contract system (in its location-bases flavor) may be used to impose security relevant isolation properties for programs.

ConScript [81] allows fine-grained application specific security policies that are enforced at run time by a modified JavaScript execution engine. Compared to our approach, they have different goals and less overhead, but are tied to a particular, obsolete browser implementation.

Program specification frameworks like Spec# [10], JML [75], or Eiffel [33] permit the formulation of access permission contracts as FOL-formulas in Hoare-style pre- and postconditions. Because specialized syntax is missing, the annotation process is rather heavy-weight. Besides, these frameworks are geared towards full specifications, whereas we are only interested in partial specifications.

Part III

Conclusion

14 Conclusion

This dissertation has presented two different approaches to increasing the software quality of JavaScript programs.

The first part of the dissertation introduced the static type system $\mathcal{JSC}$. The type system is capable of supporting strong updates for recent objects. Since its types are defined co-inductive, it is capable of supporting cyclic object structures. Therefore, it supports a typical initialization pattern of objects in JavaScript. The system is formalized and proven sound. A prototype implementation is available at `http://proglang.informatik.uni-freiburg.de/JavaScript/`.

The second part of the dissertation presented JSConTest, the first testing tool for JavaScript based on type contracts and access permission contracts. JSConTest validates contracts by (guided) random testing and contract monitoring. Type contracts are used to generate input values for functions. JSConTest uses access permission contracts to specify side effects of functions. The second part contains a full formalization of path-based access contracts with a soundness theorem, a completeness theorem and an error preservation theorem. It sketches alternative designs for access permission contracts, which are useful in other settings, for example in a security context. It also presents an inference algorithm for access permission contracts. The algorithm ensures that due to the simplification and generalization process no access paths are missed. The effectiveness of JSConTest has been evaluated based on case studies. An implementation of JSConTest, which also contains the cases studies, is available at `http://proglang.informatik.uni-freiburg.de/jscontest/`.

14.1 Review

14.1.1 Recency Abstraction

Chapter 3 introduces the idea of recency abstraction in the context of abstract interpretation. It explains that the one-to-one relation between objects and abstractions enables the type system to support strong updates for most recent objects. Consequentially, the type system based on recency abstraction is capable of typing typical object initialization patterns in such a way that the type system can ensure the absence of null pointer errors during its flow-sensitive initialization phase.

Chapter 4 starts by formalizing a core calculus for JavaScript which is named $\mathcal{JSC}$. It continues by extending $\mathcal{JSC}$ with recency information. This extension results in the development of $\mathcal{JSR}$, a calculus that makes it simple to distinguish between most recent objects and old objects. $\mathcal{JSR}$ uses a static type system to establish a set of invariants for programs. The most important invariant ensures that the most recent heap always contains at most one object per abstract location. In order to prove soundness, Chapter 4 proves some auxiliary lemmata. It finally results in a progress theorem and a preservation theorem. Further, the decidability of type checking for $\mathcal{JSR}$ is proven.

In Chapter 5 a type inference algorithm for $\mathcal{JSR}$ is presented. First, the decidability of type inference for $\mathcal{JSR}$ is sketched. Second, a realistic type inference algorithm is outlined. The realistic algorithm is implemented and it appears to work well on examples.

Since the core calculus does not support all aspects of JavaScript to keep the formal system manageable, Chapter 6 presents some extensions of $\mathcal{JSR}$. It describes how to extend $\mathcal{JSR}$ to support methods, functions with multiple parameters, conditionals, loops and prototypes.

$\mathcal{JSR}$ is a static type system that is capable of statically ensuring the absence of null pointer errors for JavaScript programs. Hence, the type system may be used during software development of JavaScript programs to increase reliability of webpages. The effort to use the type system is manageable because the type inference reduces the annotation burden dramatically.

14.1.2 JSConTest

Chapter 8 presents a tour of JSConTest and outlines its features. JSConTest is a dynamic system that is based on testing and monitoring. As a consequence, it does not impose any restrictions regarding programming style on the programmer, it provides a direct feedback loop and it is gradually applicable. The chapter introduces type contrats, guided random testing, dependent contracts and presents access permission contracts with examples.

Chapter 9 formalizes access permission contracts and motivates the design discussions JSConTest's access permission contracts are based on. The most important design decision is the path-based interpretation of access permission contracts. A formalization of access permission contracts is developed that results in proofs of the heap consistency theorem and the stability of violation theorem.

Because it might be tedious to write down access permission contracts manually, an inference algorithm for access permission contracts can be helpful. Chapter 10 presents an inference algorithm, which is based on a heuristic.

Chapter 11 sketches the implementation of type contacts and access permission contracts.

Chapter 12 evaluates the different aspects of JSConTest. The evaluation is based on mutation testing, an approach to measuring the efficiency of test suites. It turns out that the detection rate of type contracts, as JSConTest provides them, spots a high number of mutants. If access permission contracts are added, the detection rate increases again. Hence, we can conclude that the lightweight approach of JSConTest is effective. We also evaluated the effect inference. It appears that for all examples we examined, access permission inference creates access permission contracts having a quality comparable to access permission contracts created by a human person.

14.2 Future Work

14.2.1 Recency

One possible further step is to implement the features of Chapter 6. With these extensions in place the type system would become applicable to more real world JavaScript programs. Thus, an evaluation of how good the type system handles typical initialization patterns in practice could be done. For this purpose the type system should be written in such a way that it is possible to analyze source code statically and at the same time, it should be able to introduce runtime checks at all places where a static type system is not capable of ensuring type correctness. One example for such a place is a call to `eval` with an unknown string value. For this purpose one can use the runtime monitoring facilities of JSConTest to observe type contracts at runtime that correspond to the static types known to $\mathcal{JSR}$.

A negative aspect of static type systems is that it can be complicated to find a bug in a program just based on a type error. Especially in the presence of type inference, the bug in the program does not have to be at the place at which the type error is reported. Therefore, the programmer needs a thorough understanding of the type checker and the type inference to correct his program based on a type error. The situation in a recency-aware calculus is even worse because function types depend a lot on the structure of the most recent heap. It is important to enable the programmer to easily understand why the type inference algorithm has come up with a particular structure of the most recent heap. One might apply ideas from [87] to compute all possible places of a type clash. This additional information might enable the programmer to find the source of a type error much easier.

Another aspect of recency abstraction is that the type system is restrictive with respect to its abstract locations. If a library is typed using an abstract location ℓ_1 for colors, and another library uses exactly the same kind of colors, but the typing of the latter library uses ℓ_2 to abstract over colors, the two libraries become incompatible because of the different abstractions. In order to be able to combine different libraries it might be helpful to define mapping from abstract locations to other abstract locations in order to support reuse of libraries and to support modular type inference.

14.2.2 JSConTest

There remain several opportunities to extend JSConTest. A question that comes to mind is, whether we can use access permission contracts to enforce security policies. The current implementation of JSConTest, which is based on a source-to-source transformation, imposes a runtime overhead and is incapable of handling dynamic code loading (e.g. code loaded by eval). These restrictions have a minor impact on the intended use of JSConTest, but in a security setting context, it is important to address these concerns. With security in mind, a location-based interpretation of access permission contracts seems more appropriate for specifying security policies of interest. Although the dissertation proposes an implementation strategy for location-based access permission contracts, it might also be desirable to create a declarative specification of a location-based access permission contract system. With such a specification at hand, a formal foundation for further investigations is laid. An interesting question certainly is to prove that there is no way to circumvent the security policies established by a location-based access permission contract system. Based on such a specification or the proposal of the dissertation, a browser implementation can be written. An implementation in the JavaScript virtual machine of a browser can be much faster than the implementation that is based on a source-to-source transformation simply because it can access data only available to the virtual machine and it is simpler to hide meta data from the JavaScript program. It would be interesting to compare the performance of the path-based system and the location-based system.

The dissertation used JSConTest in many cases studies, but it might be desirable to collect additional data about the efficiency of the approach. It would be interesting to investigate how much software development for larger applications benefits from JSConTest with a case study.

Another opportunity for further development it to statically verify parts of contracts and access permission contracts. With this in place, it might be possible to reduce the runtime overhead of monitoring significantly. If it is possible to validate many object read and write operations for code with heavy load without the need to apply a whole program analysis, the approach will integrate well with just-in-time compilers as employed in state of the art JavaScript engines.

Bibliography

[1] Jonathan Aldrich and Craig Chambers. "Ownership Domains: Separating Aliasing Policy from Mechanism." In: *Proceedings of the 18th European Conference on Object-Oriented Programming. ECOOP '04*. Ed. by Martin Odersky. Vol. 3086. LNCS. Oslo, Norway: Springer, June 2004, pp. 1–25 (cit. on p. 198).

[2] Andres Almiray. *JsTester*. [Online, accessed 04-Nov-2011]. 2011. URL: `http://jstester.sourceforge.net/` (cit. on p. 197).

[3] Rita Z. Altucher and William Landi. "An Extended Form of Must Alias Analysis for Dynamic Allocation." In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '95*. San Francisco, CA, USA: ACM Press, 1995, pp. 74–84. DOI: `10.1145/199448.199466` (cit. on p. 115).

[4] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. "Towards Type Inference for JavaScript." In: *Proceedings of the 19th European Conference on Object-Oriented Programming. ECOOP '05*. Vol. 3586. LNCS. Glasgow, Scotland: Springer, July 2005 (cit. on p. 117).

[5] James H. Andrews, Lionel C. Briand, and Yvan Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In: *Proceedings of the 27th International Conference on Software Engineering. ICSE '05*. St. Louis, MO, USA: ACM, 2005, pp. 402–411. DOI: `10.1145/1062455.1062530` (cit. on p. 185).

[6] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. "Tool Support for Randomized Unit Testing." In: *Proceedings of the 1st International Workshop on Random Testing. RT '06*. Portland, Maine: ACM, 2006, pp. 36–45. DOI: `10.1145/1145735.1145741` (cit. on pp. 185, 196).

[7] Andrea Arcuri and Lionel Briand. "Adaptive random testing: an illusion of effectiveness?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 265–275. DOI: `10.1145/2001420.2001452` (cit. on p. 196).

[8] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. "A Framework for Automated Testing of Javascript Web Applications." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 571–580. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985871` (cit. on p. 196).

[9] Gogul Balakrishnan and Thomas W. Reps. “Recency-Abstraction for Heap-Allocated Storage.” In: *Proceedings of the 13th International Symposium on Static Analysis. SAS ’06*. Ed. by Kwangkeun Yi. Vol. 4134. LNCS. Seoul, Korea: Springer, 2006, pp. 221–239 (cit. on pp. 3, 5, 17, 115).

[10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. “The Spec# Programming System: An Overview.” In: *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices. CASSIS ’04*. Vol. 3362. LNCS. Springer, 2004, pp. 49–69 (cit. on p. 200).

[11] Kent Beck and Erich Gamma. *JUnit*. [Online, accessed 03-Nov-2011]. 2011. URL: `http://www.junit.org/` (cit. on p. 197).

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004. ISBN: 978-3-642-05880-6 (cit. on p. 2).

[13] Pamela Bhattacharya and Iulian Neamtiu. “Assessing programming language impact on development and maintenance: a study on C and C++.” In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Honolulu, HI, USA: ACM, 2011, pp. 171–180. DOI: `10.1145/1985793.1985817` (cit. on p. 1).

[14] Kevin Bierhoff and Jonathan Aldrich. “Modular Typestate Checking of Aliased Objects.” In: *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA ’07*. Montreal, QC, CA: ACM Press, New York, 2007, pp. 301–320 (cit. on p. 199).

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. “Korat: Automated Testing Based on Java Predicates.” In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA ’02*. Roma, Italy: ACM, 2002, pp. 123–133. DOI: `10.1145/566172.566191` (cit. on p. 196).

[16] John Boyland, James Noble, and William Retert. “Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only.” In: *Proceedings of the 15th European Conference on Object-Oriented Programming. ECOOP ’01*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. LNCS. Budapest, Hungary: Springer, June 2001, pp. 2–27. ISBN: 3-540-42206-4 (cit. on pp. 115, 199).

[17] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. “Multiple ownership.” In: *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA ’07*. Montreal, Quebec, Canada: ACM, 2007, pp. 441–460. DOI: `10.1145/1297027.1297060` (cit. on p. 198).

[18] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. "Analysis of Pointers and Structures." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '90.* White Plains, NY, USA: ACM, June 1990, pp. 296–310. DOI: `10.1145/93542.93585` (cit. on p. 115).

[19] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. "ARTOO: Adaptive Random Testing for Object-Oriented Software." In: *Proceedings of the 30th International Conference on Software Engineering. ICSE '08.* Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. Leipzig, Germany, ACM, 2008, pp. 71–80 (cit. on p. 196).

[20] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. "Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports." In: *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–166. DOI: `10.1109/ISSRE.2008.18` (cit. on p. 196).

[21] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs." In: *Proceedings International Conference of Functional Programming 2000. ICFP '00.* Ed. by Philip Wadler. Montreal, Canada: ACM Press, New York, Sept. 2000, pp. 268–279. DOI: `10.1145/351240.351266` (cit. on pp. 2, 195).

[22] TIOBE Programming Community. *TIOBE Programming Community Index for October 2011.* [Online, accessed 11-Oct-2011]. 2011. URL: `http://www.tiobe.com/content/paperinfo/tpci/index.html` (cit. on p. 1).

[23] Douglas Crockford. *JavaScript: The Good Parts.* O'Reilly Media, Inc., 2008. ISBN: 0596517742 (cit. on p. 11).

[24] Douglas Crockford. *JSLint.* [Online, accessed 11-Oct-2011]. 2002. URL: `http://www.jslint.com/` (cit. on p. 2).

[25] Christoph Csallner and Yannis Smaragdakis. "JCrasher: An Automatic Robustness Tester for Java." In: *Software—Practice & Experience* 34.11 (Sept. 2004), pp. 1025–1050 (cit. on p. 195).

[26] Leonardo De Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver." In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS '08. Part of ETAPS '08.* Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. URL: `http://dl.acm.org/citation.cfm?id=1792734.1792766` (cit. on p. 198).

[27] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. "ADSandbox: sandboxing JavaScript to fight malicious websites." In: *Proceedings of the 2010 ACM Symposium on Applied Computing.* SAC '10. Sierre, Switzerland: ACM, 2010, pp. 1859–1864. DOI: `10.1145/1774088.1774482` (cit. on p. 1).

[28] Justin DeWind. *JSMock.* [Online, accessed 04-Nov-2011]. 2007. URL: `http://jsmock.sourceforge.net/` (cit. on p. 197).

[29] Theo D'Hondt, ed. *Proceedings of the 24th European Conference on Object-Oriented Programming. ECOOP '10.* Vol. 6183. LNCS. Maribor, Slovenia: Springer, 2010.

[30] W. Dietl and Peter Müller. "Universes: Lightweight Ownership for JML." In: *Journal of Object Technology. JOT* 4.8 (Oct. 2005), pp. 5–32 (cit. on p. 198).

[31] Carl Eastlund. "DoubleCheck Your Theorems." In: *ACL2 2009.* Boston, MA, 2009 (cit. on pp. 2, 195).

[32] *ECMAScript Language Specification.* `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`. ECMA International, ECMA-262, 3rd edition. Dec. 1999 (cit. on p. 112).

[33] *Eiffel: Analysis, Design and Programming Language.* ECMA International, ECMA-367, 2nd edition. June 2006 (cit. on p. 200).

[34] Úlfar Erlingsson and Fred B. Schneider. "SASI Enforcement of Security Policies: A Retrospective." In: *Proceedings of the 1999 New Security Paradigms Workshop.* Caledon Hills, Ontario, Canada, Sept. 1999. URL: `http://www.cs.cornell.edu/home/ulfar/sasifinal.ps.gz` (cit. on p. 199).

[35] Manuel Fähndrich and Songtao Xia. "Establishing Object Invariants with Delayed Types." In: *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA '07.* Montreal, QC, CA: ACM Press, New York, 2007, pp. 337–350. DOI: `10.1145/1297105.1297052` (cit. on p. 116).

[36] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. "DrScheme: A Pedagogic Programming Environment for Scheme." In: *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs. PLILP '97.* Ed. by Hugh Glaser and Herbert Kuchen. Vol. 1292. LNCS. Southampton, England: Springer, Sept. 1997 (cit. on p. 195).

[37] Matthew Finifter, Joel Weinberger, and Adam Barth. "Preventing Capability Leaks in Secure JavaScript Subsets." In: *Proceedings of Network and Distributed System Security Symposium.* Internet Society. 2010, pp. 375–388 (cit. on p. 199).

[38] Firebug Working Group. *FireBug.* [Online; accessed 9-May-2011]. 2011. URL: `https://addons.mozilla.org/de/firefox/addon/firebug/` (cit. on p. 197).

[39] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. "The Essence of Compiling with Continuations." In: *Proceedings of the 1993 Programming Language Design and Implementation.* Albuquerque, NM, USA, June 1993, pp. 237–247. ISBN: 0-89791-598-4 (cit. on p. 26).

[40] Silicon Forks. *JSCoverage.* [Online, accessed 04-Nov-2011]. 2010. URL: `http://siliconforks.com/jscoverage/` (cit. on p. 197).

[41] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. "Flow-sensitive Type Qualifiers." In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '02.* Berlin, Germany: ACM Press, 2002, pp. 1–12. DOI: `10.1145/512529.512531` (cit. on pp. 104, 116).

[42] Edward Fredkin. "Trie Memory." In: *Communications of the ACM* 3 (9 Sept. 1960), pp. 490–499. ISSN: 0001-0782. DOI: `10.1145/367390.367400` (cit. on p. 169).

[43] David Gifford and John Lucassen. "Integrating Functional and Imperative Programming." In: *Proceedings 1986 ACM Conference on Lisp and Functional Programming.* 1986, pp. 28–38 (cit. on p. 198).

[44] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In: *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation. PLDI '05.* Chicago, IL, USA: ACM Press, 2005, pp. 213–223. DOI: `10.1145/1065010.1065036` (cit. on p. 196).

[45] *google-caja: A source-to-source translator for securing JavaScript-based web content.* `http://code.google.com/p/google-caja/` (cit. on p. 199).

[46] Aaron Greenhouse and John Boyland. "An Object-Oriented Effects System." In: *Proceedings of the 13th European Conference on Object-Oriented Programming. ECOOP '99.* Ed. by Rachid Guerraoui. Vol. 1628. LNCS. Lisbon, Portugal: Springer, 1999, pp. 205–229 (cit. on p. 198).

[47] Salvatore Guarnieri and Benjamin Livshits. "GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code." In: *Proceedings of the 18th conference on USENIX security symposium.* SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 151–168. URL: `http://dl.acm.org/citation.cfm?id=1855768.1855778` (cit. on p. 1).

[48] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. "Saving the world wide web from vulnerable JavaScript." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 177–187. DOI: `10.1145/2001420.2001442` (cit. on p. 1).

[49] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. "Using static analysis for Ajax intrusion detection." In: *Proceedings of the 18th international conference on World wide web.* WWW '09. Madrid, Spain: ACM, 2009, pp. 561–570. DOI: `10.1145/1526709.1526785` (cit. on p. 1).

[50] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of JavaScript." In: *Proceedings of the 24th European Conference on Object-Oriented Programming. ECOOP '10.* Ed. by Theo D'Hondt. Vol. 6183. LNCS. Maribor, Slovenia: Springer, 2010 (cit. on pp. 25, 141).

[51] Phillip Heidegger. *JavaScript.* [Online, accessed 06-Mrz-2012]. 2011. URL: `http://proglang.informatik.uni-freiburg.de/JavaScript/` (cit. on p. 5).

[52] Phillip Heidegger. *JSConTest.* [Online, accessed 06-Mrz-2012]. 2011. URL: `http://proglang.informatik.uni-freiburg.de/jscontest/` (cit. on p. 5).

[53] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. "Access permission contracts for scripting languages." In: *Proceedings 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '12.* Philadelphia, PA, USA: ACM, Jan. 2012, pp. 111–122. DOI: `10.1145/2103656.2103671` (cit. on p. 5).

[54] Phillip Heidegger and Peter Thiemann. "A Heuristic Approach for Computing Effects." In: *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns. TOOLS '11.* Ed. by Judith Bishop and Antonio Vallecillo. Vol. 6705. LNCS. Zurich, Switzerland: Springer, 2011, pp. 147–162 (cit. on p. 5).

[55] Phillip Heidegger and Peter Thiemann. "Contract-driven Testing of JavaScript Code." In: *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns.* TOOLS '10. Malaga, Spain: Springer, 2010, pp. 154–172. URL: `http://portal.acm.org/citation.cfm?id=1894386.1894395` (cit. on p. 5).

[56] Phillip Heidegger and Peter Thiemann. "Recency Types for Analyzing Scripting Languages." In: *Proceedings of the 24th European Conference on Object-Oriented Programming. ECOOP '10.* Ed. by Theo D'Hondt. Vol. 6183. LNCS. Maribor, Slovenia: Springer, 2010 (cit. on pp. 4, 95, 141).

[57] Phillip Heidegger and Peter Thiemann. "Recency Types for Dynamically-Typed Object-Based Languages." In: *Proceedings of the International Workshop on Foundations of Object-Oriented Languages.* FOOL '09. Savannah, Georgia, USA, Jan. 2009. URL: `http://www.cs.cmu.edu/~aldrich/FOOL09/heidegger.pdf` (cit. on pp. 4, 95).

[58] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* 12 (1969), pp. 576–580 (cit. on pp. 2, 198).

[59] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification.* Third. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 1999. URL: `http://www.ecma-international.org/publications/standards/Ecma-327.htm` (cit. on pp. 9, 134).

[60] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification.* 5th. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 2009. URL: `http://www.ecma-international.org/publications/standards/Ecma-327.htm` (cit. on p. 9).

[61] E. C. M. A. International. *ECMA-357: ECMAScript for XML (E4X) Specification.* Second. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 2005. URL: `http://www.ecma-international.org/publications/standards/Ecma-357.htm` (cit. on p. 9).

[62] ISO/IE FDIS 16262:2002. *Information technology - ECMAScript language specification.* International Organization for Standardization, Geneva, Switzerland, 2002. URL: `http://www.isotopicmaps.org/` (cit. on p. 9).

[63] Daniel Jackson and Craig A. Damon. "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector." In: *IEEE Transactions on Software Engineering* 22 (7 1996), pp. 484–495. DOI: `10.1109/32.538605` (cit. on p. 196).

[64] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. "Single and loving it: Must-alias analysis for higher-order languages." In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '98.* Ed. by Luca Cardelli. San Diego, CA, USA: ACM Press, Jan. 1998, pp. 329–341 (cit. on p. 115).

[65] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Interprocedural Analysis with Lazy Propagation." In: *Static Analysis - 17th International Symposium, SAS 2010.* Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Perpignan, France: Springer, 2010, pp. 320–339. ISBN: 978-3-642-15768-4 (cit. on p. 104).

[66] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript." In: *Proceedings of the 16th International Static Analysis Symposium. SAS '09.* Vol. 5673. LNCS. Springer-Verlag, Aug. 2009 (cit. on p. 117).

[67] Trevor Jim, Nikhil Swamy, and Michael Hicks. "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies." In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007.* Ed. by Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy. Banff, Alberta, Canada: ACM, 2007, pp. 601–610 (cit. on pp. 1, 197).

[68] Neil D. Jones and Stephen S. Muchnick. "Flow Analysis and Optimization of Lisp-like Languages." In: *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '79.* ACM Press, 1979, pp. 244–256 (cit. on pp. 3, 115).

[69] JSTestDriver. *JSTestDriver*. [Online; accessed 9-May-2011]. 2011. URL: http://code.google.com/p/js-test-driver/ (cit. on p. 177).

[70] Matthew Kehrt and Jonathan Aldrich. "A Theory of Linear Objects." In: *Proceedings of the International Workshop on Foundations of Object-Oriented Languages. FOOL '08*. San Francisco, CA, USA, Jan. 2008. URL: http://fool08.kuis.kyoto-u.ac.jp/kehrt.pdf (cit. on p. 116).

[71] Emre Kiciman and Benjamin Livshits. "AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications." In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 17–30. DOI: 10.1145/1294261.1294264 (cit. on p. 197).

[72] Casey Klein and Robert Bruce Findler. "Randomized Testing in PLT Redex." In: *Proceedings of the Workshop on Scheme and Functional Programming 2009*. Boston, MA, USA, 2009 (cit. on p. 195).

[73] Pivotal Labs. *Jasmine*. [Online, accessed 03-Nov-2011]. 20XX. URL: http://pivotal.github.com/jasmine/ (cit. on p. 197).

[74] Pivotal Labs. *JSUnit*. [Online, accessed 03-Nov-2011]. 2001. URL: http://www.jsunit.net/ (cit. on p. 197).

[75] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "JML: A Notation for Detailed Design." In: *Behavioral Specifications of Businesses and Systems*. Ed. by Haim Kilov, Bernhard Rumpe, and Ian Simmonds. Kluwer, 1999, pp. 175–188 (cit. on p. 200).

[76] Hermann Lehner. "A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking." PhD thesis. ETH Zurich, Switzerland, 2011 (cit. on p. 2).

[77] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. "A dynamic evaluation of the precision of static heap abstractions." In: *Proceedings of the 25th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '10*. Ed. by Shail Arora and Gary T. Leavens. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 411–427. DOI: 10.1145/1869459.1869494 (cit. on p. 116).

[78] Panagiotis Louridas. "JUnit: Unit Testing and Coding in Tandem." In: *IEEE Software* 22.4 (2005), pp. 12–15. DOI: 10.1109/MS.2005.100 (cit. on p. 2).

[79] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. "AdJail: practical enforcement of confidentiality and integrity policies on web advertisements." In: *Proceedings of the 19th USENIX conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, pp. 24–24. URL: http://dl.acm.org/citation.cfm?id=1929820.1929852 (cit. on pp. 1, 197).

[80] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Isolating JavaScript with Filters, Rewriting, and Wrappers." In: *Proceedings of the 14th European Conference on Research in Computer Security. ESORICS '09.* Saint-Malo, France: Springer-Verlag, 2009, pp. 505–522 (cit. on pp. 1, 199).

[81] Leo A. Meyerovich and Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser." In: *IEEE Symposium on Security and Privacy.* May 2010 (cit. on p. 200).

[82] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997. ISBN: ISBN 0-262-63181-4 (cit. on p. 123).

[83] Ghassan Misherghi and Zhendong Su. "HDD: hierarchical delta debugging." In: *Proceedings of the 28th International Conference on Software Engineering.* ICSE '06. Shanghai, China: ACM, 2006, pp. 142–151. DOI: `10.1145/1134285.1134307` (cit. on p. 195).

[84] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004. ISBN: 0471469122 (cit. on p. 2).

[85] Netscape Communications Corporation. *Client-Side JavaScript Reference.* 1999. URL: `http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.3/reference/` (cit. on p. 9).

[86] Netscape Communications Corporation. *Core JavaScript 1.5 Reference.* 2000. URL: `http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference` (cit. on p. 9).

[87] Matthias Neubauer and Peter Thiemann. "Discriminative Sum Types Locate the Source of Type Errors." In: *Proceedings of the International Conference of Functional Programming 2003. ICFP '03.* Ed. by Olin Shivers. Uppsala, Sweden: ACM Press, New York, 2003, pp. 15–26 (cit. on p. 205).

[88] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* 2nd corrected printing. Springer Verlag, 2005 (cit. on pp. 17, 198).

[89] James Noble, Jan Vitek, and John Potter. "Flexible Alias Protection." In: *Proceedings of the 12th European Conference on Object-Oriented Programming. ECOOP '98.* Ed. by Eric Jul. Vol. 1445. LNCS. Brussels, Belgium: Springer, 1998, pp. 158–185 (cit. on p. 198).

[90] Node.js. *Node.js.* Mar. 10, 2011. URL: `http://nodejs.org/` (cit. on pp. 9, 11).

[91] OCaml. *Objective Caml.* `http://caml.inria.fr/ocaml/index.en.html`. 2007 (cit. on p. 123).

[92] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures." In: *Proceedings of the 15th International Workshop on Computer Science Logic. CSL '01.* Vol. 2142. LNCS. Paris, France: Springer-Verlag, 2001, pp. 1–19. URL: `http://portal.acm.org/citation.cfm?id=647851.737404` (cit. on p. 56).

[93] *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA '07.* Montreal, QC, CA: ACM Press, New York, 2007.

[94] Carlos Pacheco. "Directed Random Testing." Ph.D. Cambridge, MA, USA: MIT Department of Electrical Engineering and Computer Science, June 2009 (cit. on pp. 127, 195).

[95] Carlos Pacheco and Michael D. Ernst. "Randoop: feedback-directed random testing for Java." In: *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA '07.* Montreal, Quebec, Canada: ACM, 2007, pp. 815–816. DOI: `10.1145/1297846.1297902` (cit. on p. 127).

[96] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation." In: *Proceedings of the 29th International Conference on Software Engineering.* ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. DOI: `10.1109/ICSE.2007.37` (cit. on p. 127).

[97] Larry C. Paulson. "Isabelle: The Next 700 Theorem Provers." In: *Logic and Computer Science.* Ed. by P. Odifreddi. Academic Press, 1990, pp. 361–385 (cit. on p. 2).

[98] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003. ISBN: ISBN-13: 9780521826143 | ISBN-10: 0521826144 (cit. on p. 123).

[99] PHP Group. *PHP: Hypertext Preprocessor.* 2007. URL: `http://www.php.net` (cit. on p. 9).

[100] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002 (cit. on pp. 29, 50, 52, 56, 68, 90–92).

[101] PLT. *Racket.* [Online, accessed 03-Nov-2011]. 2011. URL: `http://racket-lang.org/` (cit. on p. 195).

[102] Xin Qi and Andrew C. Myers. "Masked Types for Sound Object Initialization." In: *Proceedings 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '09.* Ed. by Benjamin Pierce. Savannah, GA, USA: ACM Press, 2009, pp. 53–65. DOI: `http://doi.acm.org/10.1145/1480881.1480890` (cit. on p. 117).

[103] John Quelch. *Quantifiying the Economic Impact of the Internet.* [Online, accessed 6-Oct-20111]. 2009. URL: `http://hbswk.hbs.edu/item/6268.html` (cit. on p. 1).

[104] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. "BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML." In: *ACM Transactions on the Web* 1.3 (2007), p. 11. DOI: `10.1145/1281480.1281481` (cit. on p. 199).

[105] John Resig. *FireUnit.* [Online, accessed 04-Nov-2011]. 2008. URL: `http://fireunit.org/` (cit. on p. 197).

[106] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proceedings of the 17th IEEE Symposium on Logic in Computer Science.* LICS'02. Los Alamitos: IEEE Computer Society, July 2002, pp. 55–74 (cit. on p. 56).

[107] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. "An analysis of the dynamic behavior of JavaScript programs." In: *Proceedings of the 2010 ACM Conference on Programming Language Design and Implementation. PLDI '10.* Toronto, Ontario, Canada: ACM, 2010, pp. 1–12. DOI: `10.1145/1806596.1806598` (cit. on p. 116).

[108] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values." In: *Proceedings of the first ACM SIGPLAN Symposium on Haskell. Haskell '08.* Victoria, BC, Canada: ACM, 2008, pp. 37–48. DOI: `10.1145/1411286.1411292` (cit. on pp. 2, 196).

[109] Lucas Serpa Silva, Yi Wei, Manuel Oriol, and Bertrand Meyer. "Evotec: Evolving the Best Testing Strategies for Contract-Equipped Programs." In: *Proceedings of the 18th Asia Pacific Software Engineering Conference.* APSEC 2011. Hanoi, 2011 (cit. on p. 196).

[110] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic." In: *Proceedings of the 23th European Conference on Object-Oriented Programming. ECOOP '09.* Ed. by Sophia Drossopoulou. Vol. 5653. LNCS. Genova, Italy: Springer, 2009, pp. 148–172 (cit. on p. 198).

[111] Frederick Smith, David Walker, and J. Gregory Morrisett. "Alias Types." In: *Proceedings of the 9th European Symposium on Programming. ESOP '00.* Ed. by Gert Smolka. Vol. 1782. LNCS. Berlin, Germany: Springer, 2000, pp. 366–381 (cit. on p. 115).

[112] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309 (cit. on p. 68).

[113] The Mozilla Organization. *Rhino: JavaScript for Java.* 2007. URL: `http://www.mozilla.org/rhino/` (cit. on p. 9).

[114] Peter Thiemann. "Towards a Type System for Analyzing JavaScript Programs." In: *Proceedings of the 14th European Symposium on Programming. ESOP '05.* Vol. 3444. LNCS. Edinburgh, Scotland: Springer, Apr. 2005, pp. 408–422 (cit. on pp. 1, 12, 117).

[115] Tiest Vilee. *rhinounit.* [Online, accessed 04-Nov-2011]. 2008. URL: `http://code.google.com/p/rhinounit/` (cit. on p. 197).

[116] W3Techs. *Usage of client-side programming languages for websites.* [Online, accessed 6-Oct-2011]. 2011. URL: `http://w3techs.com/technologies/overview/client_side_language/all` (cit. on p. 1).

[117] David Walker and Greg Morrisett. "Alias Types for Recursive Data Structures." In: *Proceedings of the ACM Workshop on Types in Compilation. TIC '00.* Ed. by Robert Harper. Vol. 2071. LNCS. Montréal, Canada: Springer, Sept. 2000, pp. 177–206 (cit. on p. 115).

[118] Yi Wei, Serge Gebhardt, Bertrand Meyer, and Manuel Oriol. "Satisfying Test Preconditions through Guided Object Selection." In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation.* ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 303–312. DOI: `10.1109/ICST.2010.34` (cit. on p. 196).

[119] Wikipedia. *Determinism — Wikipedia, The Free Encyclopedia.* [Online; accessed 03-Mar-2011]. 2011. URL: `http://en.wikipedia.org/w/index.php?title=Determinism&oldid=416544383` (cit. on p. 33).

[120] Nicholas C. Zakas. *Single Linked List.* [Online; accessed 21-Mar-2011]. 2009. URL: `https://github.com/nzakas/computer-science-in-javascript` (cit. on p. 189).

[121] Andreas Zeller. *Why Programs Fail. A Guide to Systematic Debugging.* 1st. Morgan Kaufmann Publishers, 2005 (cit. on p. 195).

[122] Andreas Zeller. "Yesterday, my program worked. Today, it does not. Why?" In: *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 253–267. DOI: `10.1145/318773.318946` (cit. on p. 195).

[123] Tian Zhao, Jens Palsberg, and Jan Vitek. "Lightweight Confinement for Featherweight Java." In: *Proceedings of the 18th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPLSA '03.* Anaheim, CA, USA: ACM Press, New York, 2003, pp. 135–148. DOI: `10.1145/949305.949318` (cit. on p. 198).

Index