UNIVERSITY OF CALIFORNIA, SAN DIEGO

Precise Type Checking for JavaScript

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Panagiotis Vekris

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel R. Buss
Professor Sorin Lerner
Professor Todd Millstein
Professor Yannis Papakonstantinou

2017

The Dissertation of Panagiotis Vekris is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

List of Tables ix

Acknowledgements x

Vita xii

Abstract of the Dissertation xiii

Chapter 1 Introduction 1
1.1 Dynamic Scripting Languages 2
1.2 Type Systems 4
1.2.1 Traits of Dynamic Type Systems 4
1.2.2 Static Types for Dynamic Languages 8
1.3 Existing Solutions 9
1.3.1 Foundations of Typing for Dynamic Languages 9
1.3.2 Typing JavaScript 12
1.3.3 Mainstream Type Checkers 14
1.4 Our Approach 15
1.4.1 Flow: Precise Constraint-Based Type Inference 15
1.4.2 Refinement Types for TypeScript 17
1.5 Contributions 22

Chapter 2 Flow: Precise Type Inference for JavaScript 23
2.1 Goals 23
2.2 Overview 24
2.3 Language FLOWCORE 28
2.3.1 Syntax 28
2.3.2 Types, Effects and Constraints 30
2.4 Constraint System 32
2.4.1 Constraint Generation 33
2.4.2 Propagation 41
2.4.3 Consistency 43
2.5 Runtime Semantics 45
2.5.1 Reduction Rules 47
2.6 Metatheory 47
2.6.1 Declarative Type System 49
2.6.2 Type Safety 50
2.7 Implementation of Type Inference 50
2.8 Related Work 52

Chapter 3 Trust, but Verify: Two-Phase Typing for Dynamic Languages 54
3.1 Overview 58
3.1.1 Value-based Overloading 58

3.1.2 Refinement Types . . . 60
3.1.3 Phase 1: Trust . . . 62
3.1.4 Phase 2: Verify . . . 64
3.1.5 Two-Phase Inference . . . 67
3.2 Syntax and Operational Semantics of TBV . . . 69
3.2.1 Source Language (λ_{src}) . . . 69
3.2.2 Target Language (λ_{tgt}) . . . 71
3.3 Phase 1: Trust . . . 73
3.3.1 Source Language Type checking and Elaboration . . . 73
3.3.2 Source and Target Language Consistency . . . 78
3.4 Phase 2: Verify . . . 81
3.4.1 Refinement Type Checking . . . 82
3.4.2 Two-Phase Type Safety . . . 84
3.5 Related Work . . . 85

Chapter 4 Refinement Types for TypeScript . . . 87
4.1 Overview . . . 88
4.1.1 Applications . . . 90
4.1.2 Analysis . . . 94
4.2 Formal System . . . 99
4.2.1 Source Language ($\mathrm{I}_{\mathsf{rsc}}$) . . . 99
4.2.2 Intermediate Language (λ_{rsc}) . . . 100
4.2.3 Static Single Assignment (SSA) Transformation . . . 101
4.2.4 Static Semantics . . . 104
4.2.5 Type Safety . . . 109
4.3 Scaling to TypeScript . . . 110
4.3.1 Types . . . 110
4.3.2 Reflection . . . 112
4.3.3 Interface Hierarchies . . . 113
4.3.4 Imperative Features . . . 115
4.4 Evaluation . . . 117
4.4.1 Benchmarks . . . 118
4.4.2 Transducers (A Case Study) . . . 121
4.4.3 Unhandled Cases . . . 122
4.5 Related Work . . . 124

Chapter 5 Conclusions and Future Work . . . 127
5.1 Flow . . . 127
5.2 Refinement Types for TypeScript . . . 128

Chapter A Flow: Precise Type Inference for JavaScript . . . 130

Chapter B Trust, but Verify: Two-Phase Typing for Dynamic Languages . . . 160

Chapter C Refinement Types for TypeScript . . . 188

Bibliography . . . 234

LIST OF FIGURES

Figure 1.1. JavaScript Function with Overloaded Behavior 7

Figure 2.1. Modern JavaScript Examples: `null` and `undefined` 25

Figure 2.2. Modern JavaScript Examples: Algebraic Data Types 26

Figure 2.3. FLOWCORE Expressions 29

Figure 2.4. FLOWCORE Type and Effect Syntax 30

Figure 2.5. FLOWCORE Constraint Syntax 31

Figure 2.6. Expression Constraint Generation in FLOWCORE (Variables and Functions) 34

Figure 2.7. Auxiliary Meta-functions for Function Logistics in FLOWCORE 35

Figure 2.8. Auxiliary Environment Operations in FLOWCORE 36

Figure 2.9. Expression Constraint Generation in FLOWCORE (Logical Operations) 37

Figure 2.10. Expression Constraint Generation in FLOWCORE (Records) 38

Figure 2.11. Statement Constraint Generation in FLOWCORE 39

Figure 2.12. Constraint Propagation in FLOWCORE 42

Figure 2.13. Runtime Definitions in FLOWCORE 46

Figure 2.14. Operational Semantics of FLOWCORE (Expressions) 48

Figure 2.15. Operational Semantics of FLOWCORE (Statements) 49

Figure 3.1. Computing the Minimum-valued Index with Higher-Order Functions 55

Figure 3.2. An Example Program with Value-Based Overloading 59

Figure 3.3. The Prevalence of Value-Based Overloading 61

Figure 3.4. Source (left) and Target (right) Program in First Phase Elaboration. 63

Figure 3.5. Language λ_{src}: Syntax and Operational Semantics 70

Figure 3.6. Basic Type Well-Formedness for λ_{src} 71

Figure 3.7. Language λ_{tgt}: Syntax and Operational Semantics 72

Figure 3.8. Elaboration Typing rules 74

Figure 3.9. Refined Type Checking for λ_{tgt} 82

Figure 4.1. Computing the Min-Valued Index with `reduce` 90
Figure 4.2. Specialization of `$reduce` Function 94
Figure 4.3. Example Adapted from D3: Two-Dimensional Arrays 98
Figure 4.4. Syntax of I_{rsc} 100
Figure 4.5. Syntax of λ_{rsc} 101
Figure 4.6. SSA Transformation in RSC 102
Figure 4.7. Type Language in RSC 104
Figure 4.8. Static Typing Rules for λ_{rsc} 106
Figure 4.9. Type Hierarchies in the `tsc` Compiler 114
Figure 4.10. Type Invariant Predicate Definition 115
Figure 4.11. Array Interface with Mutability Annotations in Refined TypeScript 116
Figure 4.12. Initialization Outside the Constructor in Refined TypeScript 117
Figure 4.13. Sample Adapted from Transducers Benchmark 122
Figure 4.14. Complex Constructor Pattern Example 123
Figure 4.15. Function Computing Distinct Elements of an Array 123
Figure 4.16. Alternative `reduce` Function 124
Figure A.1. Expression Typing in FLOWCORE (Variables and Functions) 133
Figure A.2. Expression Typing in FLOWCORE (Logical Operators and Records) 134
Figure A.3. Statement Typing in FLOWCORE 135
Figure A.4. Evaluation Context Typing in FLOWCORE 136
Figure A.5. Runtime Stack Typing in FLOWCORE 137
Figure A.6. Heap Typing in FLOWCORE 138
Figure A.7. Runtime Configuration Typing in FLOWCORE 138
Figure C.1. Syntax and Runtime Configuration of I_{rsc} 189
Figure C.2. Operational Semantics for I_{rsc} (adapted from Safe TypeScript [87]) 190
Figure C.3. Syntax and Runtime Configuration for λ_{rsc} 191

Figure C.4. Reduction Rules for λ_{rsc} 191
Figure C.5. Additional SSA Transformation Rules in RSC 193
Figure C.6. Runtime Configuration Translation in RSC 194
Figure C.7. Runtime Stack Translation in RSC 194
Figure C.8. Runtime Term Translation in RSC 195
Figure C.9. Evaluation Context Translation Rules in RSC 196
Figure C.10. Heap and Value Translation Rules in RSC 197
Figure C.11. Structural Constraints in RSC (adapted from [76]) 198
Figure C.12. Well-Formedness Rules in RSC 199
Figure C.13. Subtyping Rules in RSC 200
Figure C.14. Typing Runtime Configurations for λ_{rsc} 200

LIST OF TABLES

Table 3.1. The Prevalence of Value-Based Overloading 60

Table 4.1. Benchmark Results for `rsc` (Annotations) 119

Table 4.2. Changes Made on `rsc` Benchmarks 121

ACKNOWLEDGEMENTS

I am hugely grateful to my advisor Ranjit Jhala for his support and guidance throughout my years as a graduate student. Ranjit's unique talent to focus on the right questions and matters was key in steering my work in the right direction and making research a lot less stressful and a lot more fun. Thanks to my committee members Sorin Lerner, Sam Buss, Todd Millstein and Yannis Papakonstantinou for their useful feedback and discussion. I have also been extremely fortunate to have worked with Gavin Bierman and Avik Chaudhuri during some really productive internships, that have opened up a number of opportunities.

Being part of the CSE department and the Programming Languages group has been a great joy and honour. I would like to thank Ravi Chugh for being a great friend and patient enough to impart some of his wisdom when I was still new to the world of type systems and JavaScript. Thanks to Ben Cosman for his perseverance and courage in being an invaluable user of RSC and for providing essential feedback. To Alexander Bakst, Alan Leung, Don Jang, Dimo Bounov and the rest of my labmates for the interesting discussions we have shared and for being awesome friends over the years.

A lot of the credit for making this work possible goes to my parents, Paraskevas and Eirini. At every step of my life their example and support has fueled me with courage and inspiration. Last but not least, I am grateful to Joanna for being next to me through clear skies and stormy weather.

Work Adapted in This Dissertation

Chapter 2 in part is currently under submission for publication of the material as it may appear in **Fast and Precise Type Checking for JavaScript.** Chaudhuri, Avik; Vekris, Panagiotis. The dissertation author was an author of this paper.

Chapter 3 contains material adapted from the publication **Trust, but Verify: Two-Phase Typing for Dynamic Languages** appearing in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP'15).* Prague, Czech Republic, July 2015. Vekris, Panagiotis; Cosman, Benjamin; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 4 contains material adapted from the publication **Refinement types for TypeScript** appearing in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI '16). Santa Barbara, CA, USA, June 2016. Vekris, Panagiotis; Cosman, Benjamin; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

VITA

2011	Diploma, National Technical University of Athens
2014	Master of Science, University of California, San Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. **Refinement types for TypeScript.** In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Santa Barbara, CA, USA, June 2016.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. **Trust, but Verify: Two-Phase Typing for Dynamic Languages.** In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP'15).* Prague, Czech Republic, July 2015.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. **Safe & Efficient Gradual Typing for TypeScript.** In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* Mumbai, India, January 2015.

Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. **Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs.** In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems (HotPower'12).* Berkeley, CA, USA, October 2012.

Prodromos Gerakios, Nikolaos Papaspyrou, Konstantinos Sagonas, and Panagiotis Vekris. **Dynamic Deadlock Avoidance in Systems Code Using Statically Inferred Effects**. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. Cascais, Portugal, October 2011.

ABSTRACT OF THE DISSERTATION

Precise Type Checking for JavaScript

by

Panagiotis Vekris

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Ranjit Jhala, Chair

Dynamic scripting languages have recently experienced a dramatic growth. JavaScript in particular is one of the main technologies powering the web. As web applications grow in complexity so does the need for means to guarantee their correctness. Testing has been a valuable ally, but falls short with respect to program coverage and formal correctness guarantees. To complement this approach we propose static type-based analysis. Our goals are early bug detection, code intelligence for editors and verifying specifications; all with modest annotation effort from the developer. The biggest challenge is the dynamic nature of JavaScript: overloaded functions, closures, and mutability both at the stack and heap level.

In this dissertation, we describe our solutions to the problem of type checking JavaScript in three main contributions. First, we present the constraint-based type inference engine powering Flow, a static type checker for JavaScript. Here constraint generation accounts for uses of values

throughout the program, and constraint propagation corresponds to the notion of subtyping. Detecting bugs amounts to finding inconsistencies in the propagated constraint set. We present a formal core that supports type refinement based on runtime tests, higher-order functions, mutable variables and capture-by-reference, and prove it sound. Second, we tackle the problem of value-based overloading, where functions dynamically reflect upon and behave according to the types of their arguments. We present a novel two-phased approach to type checking that breaks the circular dependency between value and type reasoning in heavily dynamic languages. Our technique enables the straightforward composition of simple type checkers with program logics. Leveraging this advancement is our third contribution, Refined TypeScript (RSC), a refinement type system for TypeScript that enables static verification of higher-order, imperative programs. We develop a formal core of RSC that delineates the interaction between refinement types and mutability, both on local variables and objects. The core is proven sound and extended to account for features of real-world TypeScript programs. We evaluate our checker on a set of benchmarks, including parts of the Octane benchmarks and the TypeScript compiler.

Chapter 1

Introduction

Scripting languages like Perl, Bash and Python were originally designed to serve as "glue code" connecting system modules written in mostly static "system" languages like C and C++. JavaScript in particular was originally designed to assemble web components in browsers as they became dynamically available. With the prevalence of web technologies, JavaScript attained great popularity as the main technology for developing the business logic in web applications. Today, JavaScript is supported by default in all browsers and it even powers the server-side components of several web applications (most notably through the node.js platform). As applications and their respective codebases grow in size and complexity, so does the task of guaranteeing that they are free of errors.

Traditionally, developers of dynamic languages relied on testing to establish the correctness of their programs. To supplement these approaches, static analysis techniques, such as type systems, have recently seen increasing popularity in the development work-cycle. Despite superficially contradicting the mantra of rapid prototyping and development, type systems offer a plethora of benefits. Besides assisting in the early discovery of errors, they act as a concise form of documentation, provide the foundation for auto-completion and refactoring services, and with the introduction of more expressive underlying theories, operate as a means of verification with respect to a set of specifications. Porting existing typing techniques from the statically typed world, however, is at best a challenging task on its own and at worse impossible without major intervention. Bridging this gap is the primary focus of this dissertation.

In this chapter we first provide an overview of the main characteristics that identify dynamic languages (Section 1.1). Subsequently, we provide a general outline the native typing support that one could expect from a dynamically typed language, and then elaborate on the

benefits associated with introducing a static typing principle (Section 1.2). To motivate our contributions in this area, we present the highlights in the literature of typing approaches in the dynamic language setting, as well as their shortcomings that worked as starting block for our contribution (Section 1.3). Finally, we give a summary of our approach in developing a precise constraint-based type inference for JavaScript, and a technique for verifying program specifications using value based reasoning in TypeScript (Section 1.4).

1.1 Dynamic Scripting Languages

Deploying and maintaining web applications and their server-side counterparts call for a flexible developer environment that promotes rapid prototyping. This imposes clear restrictions in the design of the involved programming languages. Tratt [105] gives an excellent overview of the attributes that make dynamically languages interesting from a design perspective. The emerging design decisions directly affect our ability to reason about program correctness, which is the primary focus of this dissertation. In this section we focus on JavaScript and examine the main features that classify it as a dynamic language.

High-Level. Languages like JavaScript make extensive use of high-level features, once only found in the functional language realm, such as container constructs or functions as first class values. For example, objects can be used as dictionaries, with primitive support for the typical operations of lookup and update, and arrays come with support for a wide range of primitive operations. In several "systems" languages, like C or Java, such operations would have to be imported via standard or external libraries. Higher-order constructs facilitate the production of clean, correct and maintainable code.

Consider, for example, the following code segment that loops over the elements of an array `a`, while an accumulator `acc` is initialized and gets updated in each iteration of the loop:

```
let acc = ... ; // Initialization
for (let i = 0; i < a.length; i++) {
  acc = f(acc, a[i], i);  // Update
}
```

Modern dynamic languages let programmers factor the looping pattern into a higher-order `reduce` function (obtained by using the code segment above as function body), which frees them from the burden of manipulating indices and thereby prevents the attendant "off-by-one"

mistakes. Instead, the programmer can compute for example the sum of the elements of the array by supplying an appropriate iterator function `plus`:

```
let plus = (x, y) => x + y;
```

to the `reduce` function as follows:

```
let sum = reduce(a, plus, 0);
```

Building precise program analyses critically depends on our ability to describe the specifications of high-level constructs succinctly and accurately. In Chapters 3 and 4 we provide our approach in providing such specification by combining type systems and program logics.

Memory. In terms of *memory management* JavaScript is an automatically garbage collected language. This comes as a relief for developers and web users, as memory management is particularly error-prone in lower-level languages. In the context of a web browser this would make a web page unresponsive or even crash. Alas, accessing an array out of bounds for example can still have undesirable consequences. Typically, it leads to the surfacing of the `undefined` value that is often the culprit for an exception further down the line. In Chapter 4 we show how an expressive type system can help guard against this class of errors.

Meta-Programming. Central to the meta-programming capabilities of JavaScript is the ability of the program to inspect itself and alter its own behavior, referred to as *reflection*. In JavaScript a programmer can inspect the runtime type information of a value to determine what kind of operations are permitted on it. This is often done by using the `typeof` or `instanceof` operators, to recover its type tag (discussed more extensively later on) or test whether a constructor belongs to the prototype chain of an object. In addition, the language allows developers to dynamically alter the behavior of objects by making for example their properties writeable, or updating their getter function. The approaches presented in this work track these language capabilities to refine their program reasoning.

One of the most controversial features of JavaScript is the `eval` primitive that allows code represented as a string to be executed. `eval` is widely considered a bad programming practice [1], since it can slow down execution and if misused enable malicious code to be run with the caller's privileges. As far as static program reasoning is concerned, handling `eval`, and other constructs in the same family, is considered impractical.

[1] http://javascript.crockford.com/code.html

Closures. Finally, functions in JavaScript can refer to variables that are bound in their scope of definition. In the following simple code segment

```
let a = 1;
let f = () => { a = null; };
f();
```

calling `f()` has the effect of updating `a` with `null`. This complicates reasoning about closures, since in order to track the behavior of `f` a mere type specification is not enough; the analysis needs to keep track of effects. Chapter 2 focuses on this problem and provides a solution to it.

1.2 Type Systems

A notable omission in the previous section and one of the most important aspects that distinguishes dynamic scripting languages is the type system. In broad terms, a type system ensures that operations are applied to data in a reasonable way. For example in most languages the minus operator (-) denotes arithmetic subtraction. A common way to provide a specification for this operation is to assign it the type:

$$\text{-} :: (\mathsf{Num}, \mathsf{Num}) \rightarrow \mathsf{Num}$$

This specifies that the operation *requires* two numbers as it arguments and *guarantees* that the result will also be a number. It is the type system's responsibility to only allow numbers to be passed as arguments. Any other type should be rejected either when the program is compiled or executed. Several decades of research on type systems have led to a variety of type-based solutions for reasoning about programs. This dissertation builds on and extends this literature by providing static techniques for typing languages like JavaScript. But before we proceed to our discussion on static typing, we give an overview of the typing support that is provided with the language's runtime.

1.2.1 Traits of Dynamic Type Systems

The main axes that we survey here are: the time of type checking, types hierarchies, safety guarantees, flexibility and implicit conversions. This discussion should also help appreciate the complications involved in designing analyses for a language like JavaScript, as well as the compromises that need to be made in doing so.

Time of Type Checking. JavaScript is a *dynamically typed* language, which means that type checking happens at runtime (as opposed to statically typed where checking is done at compile-time). This is why dynamically typed languages are typically not compiled, but interpreted. In addition, unlike statically typed languages, there is no typing language interface exposed to the developer, for example there is no way for her to specify that a variable is going to hold numeric values. Variables are not associated with types, but runtime values are; a notion also known as *duck typing*. Before an operation such as a method call, the receiving object is checked to determine whether it contains the specific method, and only then is the call deemed safe.

In the absence of an explicit static type language, the runtime tags values with typing information. These *type tags* are used to perform latent checks like the above. The language of tags is limited compared to that of types in statically typed languages. For instance in JavaScript each value is associated with exactly one of the following tags: `"undefined"`, `"number"`, `"string"`, `"boolean"`, `"function"` and `"object"`. One can query the tag of a JavaScript value by calling `typeof` on it. While the information revealed by such checks is rather coarse, more complex queries can be composed by combining simple queries:

```
typeof x === "object" && typeof x.f === "number"
```

If this expression evaluates to `true`, this means that `x` is an object containing a numeric field `f`, effectively establishing that `x` is of type $\{\ldots, \texttt{f} : \texttt{number}, \ldots\}$. Patterns like these are extremely pervasive in JavaScript codebases, so any type-based analysis ought to recognize and take them into account.

Performance. Lifting the burden of static typing can speed up the development process, but the trade-off comes in the form of an overhead when performing runtime type tests. In most statically typed languages a variable declared as `int` is guaranteed to only hold integer values, without the need of a runtime operation to establish that. To bridge this performance gap, dynamic language designers have integrated their runtimes with Just In Time (JIT) compilation. This process dynamically compiles commonly used ("hot") code segments to machine code to improve performance as opposed to interpreting the same code.

Type Hierarchy Structure. A common distinction between static type systems is based on the way they handle checking for equivalence and subtyping (deciding whether one type can be used in place of another). On the one hand, we have systems that answer these queries based

on structural equivalence or inclusion. These are known as *structural* systems and are common in functional languages like ML or Haskell. On the other hand, we have *nominal* systems that determine ordering among types based on programmer specified declarations, *e.g.* Java with its explicitly specified class hierarchies. JavaScript has traits from both principles. Depending on the kind of type query we can expose a different behavior. To support the structural aspect, a query that tests the existence of the same names of fields in two objects may succeed, even if the objects were created by unrelated constructors. In this sense, duck typing is the dynamic equivalent of structural typing. On the other hand queries like the `instanceof` operator have a nominal flavor, since the exact name match of the constructor is essential.

Safety Guarantees. As far as *type safety* is concerned, we are interested in the implied guarantee that typed programs do not get stuck. Such situations include "method not found" errors or passing incompatible arguments to primitive operations. In static languages errors like these typically lead to crashes. In C for example attempting to access locations outside a program's memory segment will lead to a segmentation fault, even for programs that have passed C's type checker. For this reason C is considered a *weakly* typed language. Languages like Java, on the other hand, have more sophisticated runtime systems that check for out-of-bounds accesses and throw a suitable exception in the case of one. The same thing happens when trying to cast an object to a incompatible class. In other words, programs can reach exceptional states but in more predictable ways. Languages like this are considered *strongly typed*.

Classifying a language like JavaScript as strongly or weakly typed is not straight-forward. Since the language does not have a static type system and does not come with any static safety guarantees. Unlike the behavior of C programs, however, a failure in one of the type checks prior to an operation will not cause the entire application to crash at runtime. In several cases (*e.g.* when trying to call a non-function) this will manifest as an error message or (more commonly) it will lead to a silent type coercion (*e.g.* when trying to add a string and a number) and the program will resume execution.

Flexibility. One of the main selling points of dynamic type systems is the ability to perform quick refactorings, bypassing the need for massive code or interface restructuring. Unlike statically typed settings, variables are allowed to hold values of varying types. Take for example the function in Figure 1.1 that negates its argument using a numeric or boolean operator based on the tag of its input. (This function assumes that inputs may only be of numeric or

```
1 function negate(x) {
2   if (typeof x === "number")
3     return 0 - x;
4   else
5     return !x;
6 }
```

Figure 1.1. JavaScript Function with Overloaded Behavior

boolean type.)

Static typing requires assigning a single type to `x` for the entire scope of this function, which leads to the following dilemma: The first use of `x` in line 3 requires `x` to have a numeric type, where as the negation in line 5 requires it to be boolean.

One way to type this program is with *untagged unions*. However, with statically typed languages this approach has its shortcomings. On the one hand, several mainstream *strongly typed* languages, like Java and OCaml, do not support untagged unions. Therefore, checking a function like this would be impossible. Note that this case could be handled with the use of a tagged (or disjoin) union, but this would require defining a data type

```
type IntOrBool = I of integer | B of bool
```

and then explicitly matching values of this type against the two constructors `I` and `B` before using the underlying values.

On the other hand, *weakly typed* languages like C, allow for untagged unions but in a type-unsafe way: the underlying value is not tagged so the real sort of the data needs to be explicitly handled by the programmer. To make things worse, static checkers for languages like these rarely take into account the conditional checks that often guard operations like the above, so accidentally swapping the order of the numeric and boolean negation could easily go undetected.

These idioms are pervasive in dynamic languages. To address them, a type checker needs to allow variables to have multiple possible types, typically through untagged unions. To make this choice practical, the checker need to account for runtime conditional tests that narrow down the set of possible values of the involved expressions. Reynolds [88] was among the first to realize this necessity when checking untyped languages.

Finally, another common idiom in JavaScript are *strong updates*, *i.e.* assignments that

change the type of the updated variable. Take for example the following code snippet that is often answered in the beginning of function bodies and serves as the initializer of an optional parameter `x`:

```
x = x || 0;
```

This operation evaluates to the original value of `x`, if `x` is initially *truthy* (a value is truthy if it is not one of `false`, `0`, `""`, `null`, `undefined`, and `NaN`). Otherwise the value of `x` will be `0`. This operation completely changes the type of `x`: if `x` was initially of type `null` then after it will be a number. An analysis that handles cases like this is called *flow-sensitive*.

1.2.2 Static Types for Dynamic Languages

Having examined the main attributes of dynamic typing a plausible question is born: Is there any merit from marrying static typing with a dynamically typed language like JavaScript? To answer this question we consider the benefits we get with static typing.

IDE Support. One of the major benefits of using static type systems is in enhancing the developer experience through integration in text editors. Help to developers comes in many forms. Type errors get *detected* on the fly and are often accompanied with possible suggestion that might resolve the issue. The developer is also offered possible expression *completions*, based on type information of the part that has already been typed. Finally, API documentation is readily available based on class and interface hierarchy information.

The use of an IDE with the support of the type checker and compiler has been the de facto way of developing in many statically typed programming environments, *e.g.* Java and C#. In recent years developing JavaScript in an IDE has gained significant popularity, largely due to the integration with powerful static checkers like the one integrated in WebStorm [2] or Flow, or full-blown compiler infrastructures such as TypeScript (discussed in detail later on). A great incentive to use the systems is the availability of typed interfaces for a large number of popular JavaScript libraries for both TypeScript [3] and Flow [4].

Safety. Statically typed languages, like Java, ML and Haskell, often come with *type safety* guarantees (at least for a core part of their specification). As it was coined by Milner [73]: "Well typed programs don't go wrong." In addition, sound analyses, are usually a requirement for

[2] https://www.jetbrains.com/webstorm/
[3] http://definitelytyped.org/
[4] https://github.com/flowtype/flow-typed

semantics preserving code transformations. To enjoy similar guarantees as the complexity of scripts increases, developers often opt in for paying the burden of adding type annotations to their programs. Of course these guarantees come modulo the compromises that the type system makes with respect to *soundness*. Unfortunately, expressive type systems are at best hard and often impossible to prove sound [110], which leads to systems that are deliberately unsound [10] or sound up to some syntactic restrictions. Even unsound systems, however, can be useful as a means of *bug detection*.

Performance. An undisputed factor for a pleasant user experience in web applications is performance. JIT compilation can greatly boost JavaScript's performance, but its effectiveness is undermined by highly dynamic code, as unpredictable object structures hinder type specialization by the compiler [2]. At the same time, in ahead-of-time compilers, such as `asm.js` for JavaScript, fixed object layouts can enable aggressive optimizations. Both of these features are made possible by strong static type systems, like SJS [19].

1.3 Existing Solutions

The benefits of checking and inferring types for dynamic language have long been appreciated. Therefore, before delving into the contributions of this dissertation, we explore the highlights of this rich research area.

1.3.1 Foundations of Typing for Dynamic Languages

In this section we explore the main ideas that have been the bedrock for most current approaches in type systems for dynamic languages, including techniques in this current work.

Soft typing. This is one of the first attempts to bridge the gap between static and dynamic typing. The goal here is to retain the expressiveness of dynamic typing on the one hand, but also offer some of the benefits of static typing in error reporting and optimization capabilities on the other. Cartwright and Fagan [15] originally use *soft typing* to infer types in the context of a higher-order imperative language. They attempt to incorporate static analysis to statically type dynamic languages. However, when a program cannot be proven safe statically, it is not rejected. Instead, runtime checks are inserted and hence type safety is restored dynamically. One of the key priorities here is that the system is prudent in adding costly runtime checks.

The original work on soft typing inference was heavily influenced by Hindley-Milner

style type inference [27]. As such, it suffered from the known issue of large and complicated inferred types, and undecipherable error messages that were of little help to the developer. To circumvent this problem, Aiken et al. [5] present a soft type system based on set-based analysis. Subtyping here is realized through constraints between sets of values, that are more natural to the programmer than ML style unification constraints. In addition, they improve their accuracy by introducing *conditional types*. Here the type of an expression e can be constrained by the result of a conditional check in the context of e. Consider for example the expression:

$$\lambda y.\textbf{case } y \textbf{ of true} : \textbf{zero} \mid \textbf{false} : \textbf{succ}(\textbf{zero})$$

By introducing a conditional type $\tau_1 \,?\, \tau_2$ (to be read as τ_1 if τ_2), they infer the type:

$$\alpha \rightarrow (\textbf{zero} \,?\, (\alpha \wedge \textbf{true})) \vee (\textbf{succ}(\textbf{zero}) \,?\, (\alpha \wedge \textbf{false}))$$

This type captures the dependency of the return type on the input type. If a type **true** (resp. **false**) is passed as argument then the return type is the literal type **zero** (resp. **succ**$(zero)$). Despite being more accurate, their inference system, is not complete (*i.e.* correct programs may be flagged as erroneous). So, in order to avoid rejecting program in a dynamic language due to type errors at compiler time, they resort to runtime checks in the same way as Cartwright and Fagan [15].

Henglein and Rehof [55] build up on this work by extending soft typing's monomorphic typing to polymorphic coercions and by providing a translation of Scheme programs to ML. Finally, Wright and Cartwright [111] develop and evaluate Soft Scheme, a soft type system for all of R4RS Scheme including features that previously had not been addressed, such as uncurried procedures of fixed and variable arity, assignment, and continuations.

Gradual Typing. These works foreshadow the notion of *gradual typing* [94] that allows the programmer to control the boundary between static and dynamic checking depending on the trade-off between the need for static guarantees and deployability. The key difference between soft and gradual typing is in the involvement of the developer in controlling the parts of the program that are statically checked, as opposed to those that are checked at runtime. In particular, the designers of soft typing strived to retain the dynamic feeling of the target language by not requiring any type annotation from the programmer. Gradual typing on the other hand allows the developer to decide whether a portion of the program is type checked at compile time or

runtime, by adding or removing type annotations. Conceptually, in its extremes gradual typing borders soft typing on the one side and full static typing on the other.

Occurrence Typing. Returning to purely static enforcement, Tobin-Hochstadt and Felleisen [103] present an explicitly typed extension of Scheme (now known as Typed Racket), based on the notion of occurrence typing. In this approach, the type system takes advantage of conditional tests, to narrow down the type of unions in parts of the program dominated by the checks. Examine for example a variation of the `negate` example we saw in lines 1 – 5:

```
(lambda ([x: U Number Boolean])
  (if (number? x) (- 0 x) (not x)))
```

Here a primitive tag test `number?` is used as a predicate. In the then-branch of this conditional expression variable `x` will have the narrower type `Number`, and so it can be used as an argument to the numeric subtraction. Typed Racket also allows developers to define their own type tests in the form of functions that encode predicates, known as *latent predicates*. However, this approach is limited in that it only allows for simple *fixed* predicates over variables. In later work, Tobin-Hochstadt and Felleisen [104] extend occurrence typing to also account for logical combinations of predicates as well as components of data structures.

Constrained Types. Set constraints have been used for the purpose of type inference by Aiken and Wimmers [4] and Aiken et al. [5], who adopt the set-theoretic model to infer types in a simple functional language. Trifonov and Smith [106] and Pottier [83] infer polymorphic recursively constrained types, but retain a simpler interpretation of type terms. In their work, ground types are regular terms, and subtyping is defined explicitly on terms. This enables various simplifications to their constraint sets, like garbage collection [32, 83, 84]. Flanagan and Felleisen [38] use a simpler type representation and, based on simplification algorithms that exploit the observable equivalence of constraint sets, perform *componential* set-based analysis.

Flow, the type checker we describe in Chapter 2, builds directly on work by Pottier [84], but does not infer polymorphic types. Instead, our formal core exposes features less frequently addressed in the context of set-constraint based analyses, such as variable updates and type refinement based on conditional checks.

Advances in algebraic foundations have spurred renewed interest in this rich area. Even though polymorphic type inference with subtyping is known to be undecidable [99], Dolan and Mycroft [29] infer compact principal types by keeping a strict separation between the types used

to describe inputs and those used to describe outputs (polarities).

Semantic subtyping. The vast majority of the approaches we have seen this far follow a syntactic approach in defining the subtyping relation by axiomatising it in a formal system. In the semantic approach, instead, one starts with a model of the language and an interpretation of types as subsets of the model. Then subtyping is simply defined as set inclusion. Semantic subtyping has been proposed in the context of functional languages for XML based programming [43], ML-like languages [17], and more recently for imperative object-oriented languages, where fields can be mutable [6].

1.3.2 Typing JavaScript

While the techniques above are more general and can apply to multiple programming settings, below are prominent attempts at creating static type systems for JavaScript in particular.

Early on. Thiemann [102] provides a type system for flagging suspicious type conversions, and Anderson et al. [7] develop an inference algorithm that tracks object evolution, allowing members to be added to an object after it has been created. This is achieved by annotating each member of an object type as either potential or definite. These approaches are limited in the subset of JavaScript that they support and do not take into account newer features of the language.

Recency Types. Heidegger and Thiemann [53] introduce the notion of *recency types* and apply it on a language that captures many of the features that are answered in a language like JavaScript: first-class functions, objects as property maps, and prototypes. They propose a system that infers precise singleton object types that are handled flow-sensitively and change during the objects' initialization phase. When these precise types need to be used in a flow-insensitive context, the system automatically detects this and subsumes them to summary object types. Zhao [115] develops a similar approach, but presents a more flexible type system with support for parametric polymorphism and singleton type objects that can be extended after they are assigned to variables of summary types.

Flow Typing. In the area of type refinement, Guha et al. [50] develop *flow typing* for an imperative dynamic language that uses control and state to reason about types. Their approach is influenced by occurrence typing, but to remain sound with respect to flow sensitive variable updates they combine their type analysis with an intraprocedural flow analysis, which tracks

stack and heap refinements. Facts established by this flow analysis can be used to narrow types during type checking. This work focuses on checking rather than type inference and does not track non-local effects (*e.g.* variable updates). Building on the idea of flow typing, Lerner et al. [69] present a framework for building type systems for JavaScript, engineered modularly to encourage experimentation. This framework offers several tunable parameters, such as subtyping variance, but is also limited to local type inference.

Program Logics. Reasoning about programs through logic has recently been empowered with the advances in SMT solver technology. A comprehensive discussion is included in Section 4.5, however, here, we include a short mention of the work of Chugh et al. [20] on Dependent JavaScript (DJS), a static type checker for a sizeable subset of JavaScript, including run-time type-tests, higher-order functions, extensible objects, prototype inheritance, and arrays. In DJS, the typical subtyping constraints are transformed to logical implications that are discharged through an SMT solver.

Static Objects. Abadi and Cardelli [1] and Palsberg and Schwartzbach [77] were among the first to explore the area of object semantics and type inference. Building up on these principles, Choi et al. [19] propose a static type system for ahead-of-time compilation of JavaScript that guarantees fixed object layout. Its type inference is based on very similar foundations as Flow (Chapter 2). SJS focuses mainly on taming legacy object-oriented features (constructor functions, open methods, and prototype inheritance). Chandra et al. [18] build on this work by adding support for abstract objects, first-class methods, and recursive objects, and prove their extensions sound. Their type system supports additional features such as polymorphic arrays, operator overloading, and intersection types in manually-written interface descriptors for library code, that they found important for building GUI applications.

Abstract Interpretation & Points-to Analysis. Inferring type related information has also benefited from the use of abstract interpretation [26]. TAJS [60, 61, 62, 8] is a whole-program dataflow analyzer. It is fully automated, in that it does not expect any user type annotations and offers high precision by being flow-sensitive and partly context- and path-sensitive, and using allocation site abstraction for objects and constant propagation for primitive values. JSAI [63] is a similar framework, but offers user-specified analysis sensitivity and a complete formalism for their concrete and abstract semantics of JavaScript. Finally, SAFE [67, 78] claims even better precision than the above tools thanks to its Loop-Sensitive Analysis.

One of the first attempts for Andersen style points-to analysis for a subset of JavaScript was proposed by Jang and Choe [59], while Guarnieri and Livshits [49] use points-to analysis to enforce security and reliability policies. Sridharan et al. [97], building on top of WALA [109], increase their points-to analysis accuracy, and therefore scalability, by tracking correlated dynamic property accesses.

1.3.3 Mainstream Type Checkers

The related work in this section concerns widely adopted checkers and tools for JavaScript and other popular dynamic languages like Scheme and PHP. This part is more suitable for comparison to Flow, presented in Chapter 2.

TypeScript [107]. is a widely used optionally typed superset of JavaScript. Like Flow it aims to improve developer productivity by providing tooltips through IDEs. Unlike Flow, it focuses only on finding "likely errors" and favors convenience over soundness [10]. Type inference in TypeScript is mostly local and in some cases contextual; it doesn't perform global type inference like Flow, so in general more annotations are needed. Whenever type annotations are missing, they are considered to be `any` (instead of being implicitly inferred). This type, also known as the "dynamic type", is surrounded by a set of very relaxed typing rules. Thus, many type errors are missed. Furthermore, even with fully annotated programs, TypeScript misses type errors because of unsound typing rules. For example, "bivariant" subtyping means that functions and instances of polymorphic classes can be passed to contexts that do not preserve their typing invariants. In practice, this means that TypeScript developers have to code defensively with dynamic checks, even when types are included.

One of the main strengths of the TypeScript ecosystem that has contributed to its wide adoption is the availability of more than 2,000 type definitions [5] for several mainstream JavaScript projects [114]. However, despite being used by numerous developers these definitions are not immune to errors. Feldthaus and Møller [35] present a hybrid analysis to find discrepancies between TypeScript interfaces and their JavaScript implementations that reveals 142 errors in the declaration files of 10 libraries.

Finally, attempts have been made to mitigate the unsoundness that the language features by design. Rastogi et al. [87] extend TypeScript with an efficient gradual type system that offers a

[5] As of May, 2017

stricter set of static rules and a run-time that performs the residual dynamic checks needed for soundness; and Richards et al. [90] present a variant of TypeScript that allows developers chose between writing untyped code (*i.e.* `any`-typed code), optionally typed code and concretely typed code.

Dart & Closure. Dart [28] is another language that shares the same philosophy: unsoundness is a deliberate choice motivated by the desire to balance convenience with bug-finding. Recent work [54] recovers soundness in Dart by integrating optional type annotations and applying a flow analysis to provide static type safety guarantees. Closure [23] is another widely used type system for JavaScript that focuses on transforming code to reduce its size. It is sound modulo similar assumptions as Flow, but lacks type inference.

Other Languages. Typed Racket [86] and Hack [52] (for PHP) are also quite close in spirit to Flow. In the former, optional typing is at the level of modules and occurrence typing is used to perform type refinement, similar to the one done by Flow. The main differences are that it lacks type inference and, compared to Flow, its treatment of mutable variables is far more simplistic—there is no distinction between mutability on the stack and on the heap. On the other hand, Flow heavily borrows from Hack's design and implementation for scaling to millions of lines of code.

1.4 Our Approach

In this dissertation we describe the design of two techniques for static analysis of JavaScript code:

- a technique for precise constraint-based type inference, and
- a technique for verifying program specifications using refinement types.

Before we give an overview of each of the two techniques developed in this dissertation, we outline the desired goal and shed light in the reasons why current literature falls short of fulfilling it.

1.4.1 Flow: Precise Constraint-Based Type Inference

Motivation. In this part we aim to provide a (fast) system for inferring precise types for JavaScript. So the main desiderata here are precision and automation (inference).

To make the notion of *precision* more clear, we would like our analysis to take into account runtime tests on the type of program variables, and use them to refine types in code segments guarded by these tests (path-sensitivity). Existing systems [69, 18], while supporting very sophisticated typing features, do not track such type narrowings. We would also like an analysis that can track mutations to local variables and use the type of the most recent assignment to maintain precision. Flow-sensitive analyses, like recency types [53], track this correctly but fail to account for dynamic tests.

On the other hand, several of the above techniques [69, 20] fail to fulfill our *inference* requirement, namely that the type analysis handles programs like the above without requiring type annotations from the user at function boundaries. They mostly rely on local type inference [81] and so require annotation as function signatures.

Finally, the related work in the area of abstract interpretation of Section 1.3.2 involves whole-program analysis that is typically limited in terms of scalability.

Challenges. Supporting type refinement in the presence of variable updates is not trivial. Handling the effect of these updates in the presence of closures escalates the situation even further. A sound analysis needs to account for the effect of functions in updating variables that are captured in their lexical scope. Such capabilities, however, are often outside the scope of traditional type systems.

Approach. Our technique fulfills be above goals despite the challenges. At the core of the approach lies a constraint-based type inference engine [106, 3, 83, 37]. Here, directed constraints are the equivalent of subtyping constraints in a type checking setting, or *flows* of type information if we consider the analysis a dataflow one. The two parts of a flow involve types built with the usual type constructors (Bool, $\rightarrow$, *etc.*), corresponding to parts of the program whose type can immediately be inferred (*e.g.* `true` can immediately be assigned the type Bool, or a function's type will have $\rightarrow$ as the top-level constructor); and *type variables* for parts of the program whose type cannot yet be determined (*e.g.* variables, or function parameters).

The first part of the analysis is *constraint generation*, which corresponds to the specification phase [3]. This part abstracts the information encoded in the program into a set of constraints among constructed types and type variables. We perform this phase while keeping track of the bindings of variables to type information in structures called *environments*. To make our analysis precise with respect to *strong updates*, *i.e.* assignments that change the type of the updated

variable, we make our environments *flow-sensitive*. This means that each variable assignment introduces a type binding of that variable to a fresh type variable constrained appropriately based on the assigned expression.

Compiling a system of constraints corresponds to setting up a system of dataflow facts. What needs to happen next is propagate the dataflow facts until we reach a fixpoint. This corresponds to the phase of *constraint propagation*. The rules that dictate this phase draw similarities to subtyping rules.

A common pitfall in these kinds of analyses is the *effect* of closures in updating variables that are in their lexical scope. Our approach to handle cases like this is twofold. First, for each function we infer a set of variables that get (transitively) updated in their body. Second, for each variable we keep two type entries in our environment: a special type that tracks the latest assignment, and a general one that accounts for all possible assignments. When processing a call to an effectful function, for each variable in the function's effect we introduce a new binding in the environment succeeding the call bound to a fresh type variable, to which we flow the most general version of the variable's type. This way variables updated in closures, will effectively obtain the conservative general type after a function call.

Finally, after repeatedly applying our propagation rules, our constraint system reaches a fixpoint (it becomes saturated). At this point we check the system for *consistency*. Any incompatibility between the constructors of the left- and right-hand side of a constraint is reported as a potential error.

To sum up, by narrowing variable types by taking into account dynamic type tests, our analysis becomes precise and by being aware of the effect of closures on variable updates it retains its soundness. Chapter 2 expands on these ideas with examples and presents a formal core for the Flow system, including a statement of type safety.

1.4.2 Refinement Types for TypeScript

Motivation. In the first part of this dissertation we focus on inferring types that capture relatively coarse invariants about JavaScript programs. These system are sufficient in catching bugs early while developing large scale applications. For this second part we turn our focus to the problem of *automatic program verification* for dynamic languages. Our goal is to provide the means for analyzing programs and determining whether they abide by certain desired

criteria (verification), while only requiring the developer to provide a small amount of essential annotations (automatic).

The field of automatic program verification has a long and rich history. Most techniques fall under the fields of abstract interpretation [26] and model checking [22]. The type system approach we have examined so far falls under the first category [25]. However, the flavor of system of the previous section (*i.e.* a mostly syntactic type system) falls short in *expressiveness* when it comes to reasoning about value dependent properties of programs and in particular relational properties. Take for example the following simple array access within a loop:

```
1 for (let i = 0; i < a.length; i++) {
2   assert(a[i] !== undefined);
3 }
```

To discharge the assertion, a program analysis would need to relate an invariant of the array `a`, namely its length, with values that the index `i` receives throughout the loop.

Approach. This gap in expressiveness of type systems is bridged with the introduction of *refinement type systems* [42]. Here, basic types are decorated with refinement predicates that constrain the values inhabiting the type. For example to define the non-negative integer numbers that are less that 100 we can write:

$$\{\nu\colon \texttt{number} \mid 0 \leq \nu \wedge \nu < 100\}$$

Value ν, known as the *value variable*, indicates the value which is described by this type. With this as a basic block, more complex types can be built, for example the type of an array with non-negative numbers is:

$$\{\nu\colon \texttt{number} \mid 0 \leq \nu\}\texttt{[]}$$

We can also express high-level invariants of containers, for example the type of a non-empty array would be:

$$\{\nu\colon \texttt{number[]} \mid 0 < \texttt{len}(\nu)\}$$

Typically the language of predicates refining types are logical formulas from an SMT decidable logic, which allows subtyping to be reduced to queries to an SMT solver. Since its inception refinement typing has mostly targeted functional languages [113, 66, 91]. More recently, its domain was extended to dynamic [11] and imperative languages [76, 20]. Dependent

JavaScript (DJS) [20] in particular combines nested refinements [21] with alias types [95], a restricted separation logic, to account for aliasing and flow-sensitive heap updates to obtain a static type system for a large portion of JavaScript. DJS, however, proved to be extremely difficult to use. First, the programmer had to spend a lot of effort on manual heap related annotations; a task that became especially cumbersome in the presence of higher-order functions. Second, nested refinements precluded the possibility of refinement inference, further increasing the burden on the user. X10 [76] is a language that extends an object-oriented type system with constraints on the immutable state of classes. This approach is limited in providing inference (similar to DJS) and handling variable updates in a flow-sensitive manner.

Our goal in this part of the dissertation is to overcome these pitfalls, by through simple transformations and a lightweight mutability tracking type system upon which we perform refinement type inference.

Challenges. Refinement type systems require a *base* type system in order to apply the refinements on. However, establishing base types in the first place can be challenging. In Figure 1.1 we saw how the implementation of `negate` depends on the tag of the input value. This trend of dynamic languages, where functions can dynamically reflect upon and behave according to the types of its arguments, we refer to as *value based overloading*. Thus, to establish basic types, the analysis must reason precisely about values, but in the presence of higher-order functions and polymorphism, this reasoning itself can require basic types.

Another cumbersome feature for analyses in languages like JavaScript is precisely handling *local variable updates*. Consider for example the code from lines 1 – 2, only this time transformed into a while-loop:

```
4  let i = 0;
5  while(i < a.length) {
6    assert(a[i] !== undefined);
7    i = i + 1;
8  }
```

Assume we naïvely attempt to assign types in a flow-insensitive manner. It will be impossible to discharge the assertion conditions. Indeed, to assign `i` a refinement type capturing all possible value assignments to it, we would have to account (i) for the initialization of `i` to `0` and (ii) all the values that `i` is updated with, until the loop condition `i < a.length` is no longer valid. In other words, we need to consider numbers from `0` up to the length of the array `a`. This range is more

aptly described with the type

$$\texttt{i} :: \{\, \nu\colon \texttt{number} \mid 0 \leq \nu \wedge \nu \leq \texttt{len(a)} \,\}$$

However, this type is not precise enough to prove the assertion. It will fail due to the requirement that the value of `i` be less than `len(a)`. Remedying this situation requires considering different versions of the iterator `i` for before and after the update ($\texttt{i}_1$ and $\texttt{i}_2$, respectively). A possible valid type assignment for these two variables is the following:

$$\texttt{i}_1 :: \{\, \nu\colon \texttt{number} \mid 0 \leq \nu \wedge \nu < \texttt{len(a)} \,\}$$

$$\texttt{i}_2 :: \{\, \nu\colon \texttt{number} \mid 0 < \nu \wedge \nu \leq \texttt{len(a)} \,\}$$

The version of `i` that takes part in the assertion is $\texttt{i}_1$, and this time we can easily verify that is respects the assertion requirements.

Finally, another rocky situation arises from the interaction of *object mutation* with our refinement logic. The problem arises due to the fact that objects in JavaScript, including arrays, are mutable. This means that their shape and invariants, such as the length of an array, may change at any point. Due to the functional nature of SMT solvers, the values that get embedded in our refinement logic, including for example arguments to the len operator that appear in our predicates, need to be immutable portions. Let for example the following code:

```
9  let a = [0, 1, 2];
10 a.pop();
11 assert(a[2] !== undefined);
```

Due to line 9 we might be tempted to assign `a` the type:

$$\texttt{a} :: \{\, \nu\colon \texttt{number[]} \mid \texttt{len}(\nu) = 3 \,\}$$

This would suffice to discharge the assertion in line 11, rendering our analysis unsound. The problem here is that the invariant encoded in the above refinement type is silently invalidated by the pop operation in line 10, since this method changes the length of the receiving array. Only program values whose invariants remain immune to mutation, should be embedded in the refinement logic.

Our Solutions. To handle value-based overloading we propose the framework of *two-phased typing*. The first "trust" phase performs classical, i.e. flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with dead-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase uses refinement typing, a flow- and path-sensitive analysis, that decorates the first phase's types with logical predicates to track value relationships and thereby verify the casts and establish other correctness properties for dynamically typed languages.

To tackle the remaining challenges pertinent to the imperative nature of JavaScript, namely local variable updates and object mutation, we present Refined TypeScript (RSC), a lightweight refinement type system for TypeScript, that enables static verification of higher-order, imperative programs. We develop a formal system for RSC that delineates the interaction between refinement types and mutability, and enables flow-sensitive reasoning by translating input programs to an equivalent intermediate SSA form. By establishing type safety for the intermediate form, we prove safety for the input programs. Next, we extend the core to account for imperative and dynamic features of TypeScript, including overloading, type reflection, ad hoc type hierarchies and object initialization. Finally, we evaluate RSC on a set of real-world benchmarks, including parts of the Octane benchmarks, D3, Transducers, and the TypeScript compiler. We show how RSC successfully establishes a number of value dependent properties, such as the safety of array accesses and downcasts, while incurring a modest overhead in type annotations and code restructuring

A Note on Scope

While the features targeted by existing analyses outlined in Section 1.3.2, *e.g.* prototype inheritance, reflection, and legacy patterns, have wide-spread support by most JavaScript implementations, they are rarely the flavor of the language that most JavaScript developers use. Instead, there is an ongoing trend towards more recent specification like ES6 or typed variants of the language like TypeScript. Developers prefer programming in higher-level (sub)languages that are later compiled to lower-level browser-executable code with the use of tools like Babel [6] and `tsc`. For example, instead of building classes using prototype inheritance now develop-

[6] https://babeljs.io/

ers can use class constructs, just like Java. Instead of imperative features like iterators, they use higher-order constructs like `reduce` and `map`. So, instead of focusing on the target of this compilation, this work focuses on the source.

1.5 Contributions

Concretely, this dissertation makes the following contributions:

- In Chapter 2 we present the design of Flow, a fast and precise type checker for JavaScript. The approach uses constrained-based type inference and supports a variety of common JavaScript idioms. We formalize inference to support refinements in a core fragment of JavaScript containing higher-order functions, mutable variables, runtime tests, and capture-by-reference. We prove our system sound with respect to a runtime semantics.

- In Chapter 3 we tackle the problem of typing a dynamic language by introducing the framework of two-phased typing. We first elaborate a source language with value-based overloading into a target language with dead-casts in lieu of overloading. We prove that the elaborated target preserves the semantics of the source, *i.e.* the dead-casts fail iff the source would hit a type error at run time. Finally, we apply standard refinement typing on the elaborated well-typed target to statically verify the dead-casts, yielding end-to-end soundness for our system.

- In Chapter 4 we examine the interaction of refinement types and mutability and local variable updates. We formalizes our approach via SSA translation and a declarative refinement type system that we prove sound. We extend the core language to TypeScript by describing how we account for its various dynamic and imperative features; in particular we show how RSC accounts for type reflection via intersection types, encodes interface hierarchies via refinements. We then evaluate our tool on a suite of real world programs.

- We finally conclude with a discussion of limitations and future directions.

Chapter 2

Flow: Precise Type Inference for JavaScript

JavaScript is one of the most popular languages for writing web and mobile applications today. The language facilitates fast prototyping of ideas via dynamic typing. The runtime provides the means for fast iteration on those ideas via dynamic compilation. This fuels a fast edit-refresh cycle, which promises an immersive coding experience that is quite appealing to creative developers.

However, evolving and growing a JavaScript codebase is notoriously challenging. Developers spend a lot of time debugging silly mistakes—like mistyped property names, out-of-order arguments, references to missing values, checks that never fail due to implicit conversions, and so on—and worse, unraveling assumptions and guarantees in code written by others. In many other languages, this overhead is mitigated by having a layer of types over the code and building tools for the developer that use type information. For example, types can be used to identify common bugs and to document interfaces of libraries. Our aim is to bring such type-based tooling to JavaScript.

2.1 Goals

In this chapter, we present the design of the type system underlying Flow, a static type checker for JavaScript developed at Facebook. The idea of using types to manage code evolution and growth in JavaScript (and related languages) is not new. In fact, several useful type systems have been built for JavaScript in recent years. The design and implementation of Flow are driven by the specific demands of real-world JavaScript development that we have observed in the industry at large.

- The type checker must be able to cover large parts of the code base without requiring too

many changes in the code itself. Developers want precise answers to code intelligence queries (the type of an expression, the definition reaching a reference, the set of possible completions at a point). Relatedly, they want to catch a large number of common bugs with very few false positives.

- The type checker must provide very fast responses, even on a very large codebase. Developers do not want any noticeable "compile-time" latency in their normal workflow, because that would defeat the whole purpose of using JavaScript.

To meet these demands, we had to make careful choices and solve technical challenges in Flow that go beyond related existing systems.

- We precisely model common JavaScript idioms that appear pervasively in a modern JavaScript codebase. For example, Flow understands the pattern

  ```
  x = x || 0;
  ```

  that can be used to initialize an optional parameter `x` in a function body. Handling a case like this necessitates support for type refinements: the system needs to recognize that the assigned value will be truthy *i.e. refined* with respect to the initial value of `x`. In addition, the analysis needs to distinguish the version of the variable before the assignment from the one after it, in a *flow-sensitive* manner. Conflating the types of the two versions into the union of the two would invalidate the effect of the type refinement. (More examples are shown below.)

- At the same time, we *do not* focus on reflection and legacy patterns that appear in a relatively small fraction (that is also usually stable and well-tested). Today, tools like Babel convert modern JavaScript to (lower-level) ES5 executed on browsers. Flow focuses on analyzing the source, instead of the target, of such translations (unlike many previous efforts that address ES5, or the even harder ES3).

2.2 Overview

We now introduce the main ideas behind Flow's design and implementation. We discuss how Flow precisely handles type refinement in the presence of local updates and closures.

```
1  function pipe(x, f) {
2    f(x);
3  }
4
5  let hello = (s) => console.log("hello", s);
6
7  pipe("world", hello);
8  pipe("hello", null);    // error
9
10 function checked_pipe(x, f) {
11   if (f != null) f(x);  // ok
12 }
```

Figure 2.1. Modern JavaScript Examples: `null` and `undefined`

Precise type checking

One of the main contributors of Flow's precision is path-sensitivity: the way types interact with runtime tests. The essence of many JavaScript idioms is to put together ad hoc sets of runtime values and to take them apart with shallow, structural (in)equality checks. In Flow, the set of runtime values that a variable may contain is described by its type, and a runtime test on that variable *refines* the type to a smaller set. This ability turns out to be quite powerful and general in practice.

In this chapter, we formalize refinements in a core subset of JavaScript. The system is particularly interesting because of the combination of *mutable* local variables and *closures* that capture them by reference. Next, we illustrate this system via a series of examples.

Type Refinement. Higher-order functions like `pipe` in (lines 1 – 2 of Figure 2.1) are quite common in JavaScript. A common pattern in JavaScript code is to use `null` as the default argument at a function call (line 8), or even to avoid passing the argument whatsoever. In the latter case the parameter is then bound to `undefined` in the body of the function. Unfortunately, this causes the dreaded "*null is not a function*" error to hit often. Fortunately, Flow finds these errors by following flows of `null` to calls in the code.

Checking for nullability is the idiomatic way to prevent such errors at runtime. In JavaScript, the check `f != null` is equivalent to `f !== null && f !== undefined`, which additionally rules out `undefined`, commonly used to denote missing values. Thankfully Flow understands that this code is safe. It refines the type of `f` to filter out `null` and `undefined` in

```
13  let nil = { kind: "nil" };
14  let cons = (head, tail) => {
15    return { kind: "cons", head, tail };
16  }
17
18  function sum(list) {
19    if (list.kind === "cons") {
20      return list.head + sum(list.tail); // ok
21    }
22    return 0;
23  }
24
25  sum(cons(6, cons(7, nil)));
26
27  function merge(x) {
28    x = x || nil;
29    return x.kind; // ok
30  }
31
32  function havoc(x) {
33    let reset = () => { x = null; }
34    x = x || nil;
35    reset();
36    return x.kind; // error
37  }
```

Figure 2.2. Modern JavaScript Examples: Algebraic Data Types

line 11, and thus knows that neither of these values can reach the call. Many other idiomatic variants also work, such as `f && f(x)`, where `f` is checked for "truthiness" (which rules out `null`, `undefined` and other primitive values such as `false`, `0`, and `""`) before calling.

Algebraic Data Types. Refinements also power a common technique to encode algebraic data types in JavaScript, which are used quite widely (to manage actions and dispatchers, data and queries, *etc.* in user interface libraries). Records of different shapes have a common property that specifies the "constructor", and other properties dependent on the constructor value. These records are then analyzed by "pattern matching"—inspecting and branching on the constructor value.

For example, consider the encoding of lists in lines 13 – 16 of Figure 2.2. A `sum` function (line 18) checks whether a list is non-empty before accessing properties specific to non-empty

lists. Following the calls to `sum`, Flow knows that the parameter `list` in line 18 can contain both kinds of objects—those whose `kind` property is `"cons"`, and those for which it is `"nil"`. The latter ones are filtered out by refining the type of `list` in line 19, so that the only objects reaching the property accesses of `head` and `tail` in line 20 are guaranteed to have those properties. Thus, Flow knows that this code is safe. Without refinements, on the other hand, the analysis would have over-conservatively concluded that `nil` can also flow to the property accesses, leading to spurious type errors.

Refinements are tracked by a flow-sensitive analysis, and interact in interesting ways with variable assignment. The common idiom in line 28 of `merge` ensures that a variable has a non-`null` default. Flow models the assignment by merging the refined type of `x` with the type of `nil` and updating the type of `x` with it.

On the other hand, refinements can be *invalidated* by assignments, which can even happen indirectly via calls Here the call to `reset` in line 35 updates the value of `x` with `null` which invalidates the refinement that preceded in line 34. While invalidating refinements is necessary for soundness, they should be preserved as much as possible to avoid spurious type errors. Flow tracks variable assignments as effects for precise invalidation. So in addition to a type signature, Flow infers an effect signature for function `reset` that contains in it the variable `x`. Any call to this function would result in the invalidation of refinements on variables that are contained in its effect.

Refinements and their invalidation carry over to higher-order functions. A function's effect is part of its signature and is applied every time the function is called. We also have limited support for refining mutable object properties, but those refinements are invalidated aggressively (*i.e.* our analysis is not heap-sensitive).

This concludes our overview of the Flow type checker. The remainder of this chapter formalizes Flow as a set-constraint based type inference engine. In Section 2.3 we present a core fragment of JavaScript containing higher-order functions, mutable variables, runtime tests, and capture-by-reference, called FLOWCORE. We present a type language for FLOWCORE (Section 2.3.2) and continue on by describing constraint generation (Section 2.4.1) and propagation (Section 2.4.2). We then connect our static checking procedure with a runtime semantics of FLOWCORE through a type soundness result (Section 2.6). We conclude with a discussion on implementation of type inference as a system of set-based constraints combined with unification

for optimization (Section 2.7).

2.3 Language FLOWCORE

We consider a minimal subset of JavaScript that includes functions, block-scoped mutable variables, primitive values and records. Notably, we leave out data structures like dictionaries and arrays, as well as object-oriented features like `this`, methods, classes, and inheritance. These parts of the language are mostly orthogonal to understanding refinements and their type inference is built on the same foundations, and while interesting, behave more or less similarly to previous work. So we can safely extend our model to include them, without significantly complicating our guarantees. Our focus is on formalizing type inference and refinement strengthening, with the exception of refinements on mutable fields that are not tracked through the heap. While compact, this fragment is expressive enough to model the examples of Section 2.2, which are used to illustrate how Flow uses predicate refinements to reduce the false positive rate, while remain sound with respect to variable updates.

2.3.1 Syntax

Figure 2.3 describes the language of expressions *expressions* e and *statements* s.

Expressions. We elide primitive values and operations. These may include numbers and arithmetic operations, booleans, and `undefined`. The syntax $p(x)$ draws from a fixed, possibly infinite set of unary *predicates* p on x. These model dynamic checks, such as `typeof x === "number"`, `x === undefined`, `x`, testing if an expression is *truthy*, or model tests like `x.f === "nil"` on records or strings. Note that in this system the last check does not imply a predicate on the value of `x.f`, but rather on `x` itself. The former would be a heap refinement, which Flow only supports in a limited fashion, and which is excluded from the formalism.

General-purpose functions (using the keyword `function`) are complicated in JavaScript: they can be additionally used as methods and as constructors. To simplify our exposition, we restrict our attention to *arrow* functions (essentially lambdas). We assume that a function body consists of a statement followed by the return of an expression. Functions that do not explicitly return anything can be thought of as implicitly returning `undefined`. (Flow's treatment of abnormal control flows via `return` is also interesting, but we omit them here.) We also include the logical conjunction (`&&`), disjunction (`||`) and negation (`!`) operators, as they are pervasive in

n	$\in$	*Consts*	**Constants**
x, y	$\in$	*Vars*	**Variables**
e	$::=$		**Expressions**
	$\mid$	x	variable
	$\mid$	n	constant
	$\mid$	$x = e$	assignment
	$\mid$	$(x) \Rightarrow \{s;\ \texttt{return}\ e\}$	arrow function
	$\mid$	$e_1(e_2)$	function call
	$\mid$	$p(x)$	predicate expression
	$\mid$	$e_1\ \texttt{\&\&}\ e_2$	logical and
	$\mid$	$e_1\ \texttt{||}\ e_2$	logical or
	$\mid$	$\texttt{!}\, e$	logical negation
	$\mid$	$\{f_1 : e_1, \ldots, f_n : e_n\}$	object literal
	$\mid$	$e.f$	field read
	$\mid$	$e_1.f = e_2$	field write
s	$::=$		**Statements**
	$\mid$	e	expression
	$\mid$	$\texttt{var}\ x = e$	variable declaration
	$\mid$	$\texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$	if-statement
	$\mid$	$s_1\ ;\ s_2$	sequencing
	$\mid$	`skip`	no-op

Figure 2.3. FLOWCORE Expressions

JavaScript and inform our refinement strategy.

Statements. We use `var` to introduce variables, and include statements for conditional execution and sequencing. We omit `const` because it is much simpler than `var`—refinements never need to be invalidated. We also omit `while` here; although it can be encoded with `if` and recursion, Flow's treatment of it is more precise.

A program can be modeled as an expression, *e.g.* of the form `((x) ⇒ {s; return e})(0)`.

Assumption. We assume an α-renaming pre-pass over the program's AST that would rename all variable identifiers to unique names, so that each variable identifier has a unique definition point which is either a `var` statement or an arrow definition. This is a fairly straightforward transformation for any preprocessor that helps avoid non-intentional capturing of variables

$\alpha, \beta, \gamma, \delta$	$\in$	$\mathcal{V}$	**Type Variables**
$\dot{\tau}$	$::=$		**Type Literals**
	$\mid$	b	base type
	$\mid$	$\tau_1 \xrightarrow{\epsilon} \tau_2$	arrow type
	$\mid$	$\{f_1 : \alpha_1, \ldots, f_n : \alpha_n\}$	object type
τ	$::=$		**Types**
	$\mid$	$\dot{\tau}$	type literal
	$\mid$	$\tau_1 \sqcup \tau_2$	union type
	$\mid$	α	type variable
ϕ	$\in$	$\mathcal{E}$	**Effect Variables**
$\dot{\epsilon}$	$::=$		**Effect Literals**
	$\mid$	$\bot$	empty effect
	$\mid$	x	program variable
ϵ	$::=$		**Effects**
	$\mid$	$\dot{\epsilon}$	effect literal
	$\mid$	$\epsilon_1 \sqcup \epsilon_2$	effect concatenation
	$\mid$	ϕ	effect variable

Figure 2.4. FLOWCORE Type and Effect Syntax

in exported closures, leading to unnecessary precision loss.

2.3.2 Types, Effects and Constraints

The basic ingredients of our constraint system are *types* τ and *effects* ϵ. Their syntax is described in Figure 2.4.

Types. The building blocks for constructing complex type structures are *type literals* $\dot{\tau}$. These include primitive types b (*e.g.* the primitive `number` and `void` for `undefined`), arrow types $\tau_1 \xrightarrow{\epsilon} \tau_2$ for functions, and record types $\{f_1 : \alpha_1, \ldots, f_n : \alpha_n\}$. Arrow types are annotated with an effect ϵ which describes a set of names x that may be assigned in the function's body or transitively in code that is executed when calling this function. A more proper introduction of effects follows. Types also feature a binary join operator $\sqcup$ denoting the disjunctive choice among its operands. Finally, types are ranged over by variables α, β, *etc.* taken from an enumerable set $\mathcal{V}$.

u_τ	$::=$		**Type Uses**
	$\mid$	α	type variable
	$\mid$	$\mathsf{Call}(\tau)$	function call
	$\mid$	$\mathsf{Pred}(P, \tau)$	predicate
	$\mid$	$\mathsf{Get}\,(f, \tau)$	field read
	$\mid$	$\mathsf{Set}\,(f, \tau)$	field write
u_ϵ	$::=$		**Effect Uses**
	$\mid$	ϕ	effect variable
	$\mid$	$\mathsf{Havoc}\,(\Gamma)$	havoc
P	$::=$		**Predicates**
	$\mid$	p	primitive predicate
	$\mid$	$\neg p$	primitive predicate negation
c	$::=$		**Constraints**
	$\mid$	$\tau \leq u_\tau$	type constraint
	$\mid$	$\epsilon \leq u_\epsilon$	effect constraint

Figure 2.5. FLOWCORE Constraint Syntax

Effects. The effect we are interested in tracking here is variable updates. Each language term is associated with an effect, as we will see later in constraint generation. This is (roughly) the set of variables that are (re)assigned within this term. The base constructors of effects are the empty effect $\bot$ and variable symbols x, corresponding to the variables that are updated. Like types, effects also feature a join operator $\sqcup$ denoting effectively concatenation of effects. Finally, effects are ranged over by variables ϕ taken from an enumerable set $\mathcal{E}$.

Environments. An *environment* Γ binds variables x to entries τ^α, meaning that its most recent assignment was of type τ, whereas the type variable α is used as the collective summary for all its (past, current, and future) assignments. Here τ is flow-sensitive—its value may change from one (flow-sensitive) environment to another—whereas α is invariant. For the rest of this section, we distinguish environment *extension* "$\Gamma,\, x\colon \tau^\alpha$ " (variable x is not bound in the original environment Γ), from environment *update* "$\Gamma[\, x \mapsto \tau^\alpha\,]$" (variable x was bound in Γ). In certain situations a more general form of environment entry $\tau^{\tau'}$ might be used, *i.e.* using a type τ' instead of a type variable as the general type. This is a mere convenience and not a fundamental difference.

Predicates. Key to our type refining process is the notion of predicates. A predicate P is a clause denoting a property of its implied argument. In our setting, syntactically it can be a *base* predicate p or its negation. Base predicates describe properties of constructed or primitive types. For the remaining sections we will keep these predicates abstract, but examples of these predicates are the ones implied by checks of the form `typeof` $\star$ `===` `"string"`, `typeof` $\star$ `===` `"number"`, $\star$`.f ===` `"null"`, *etc.*, where $\star$ is a program variable.

Constraints. A *constraint* c is a "flow" from a type τ (resp. effect ϵ) to a type *use* $\mathfrak{u}_\tau$ (resp. an effect use $\mathfrak{u}_\epsilon$). Type constraints generalize the notion of subtyping constraints. However, we chose to enforce some structural restrictions to the forms that can appear on the right-hand side of constraints, namely the uses. Type and effect variables can be uses themselves. We do not allow general types and effects to appear in place of a use. Instead we introduce constructors to wrap type or effect information regarding the operation that instigated the constraint. Uses account for data flow through function calls (Call), control flow refinement (Pred), object operations (Get, Set) and refinement invalidation (Havoc).

The use $\mathsf{Call}(\tau)$, where the only valid form for τ is $\tau_1 \xrightarrow{\epsilon} \tau_2$ corresponds to a function call with argument type τ_1, resulting in type τ_2; the effect ϵ models the effect of the receiving function. The use context $\mathsf{Pred}(\mathsf{P}, \tau)$ is used to *refine* an incoming type using *predicate* P, resulting in type τ. In other words, a constraint $\tau_0 \leq \mathsf{Pred}(\mathsf{P}, \tau)$ will only allow the parts of τ_0 that satisfy P to flow to τ. The uses for accessing and writing to a field, $\mathsf{Get}\,(\mathsf{f}, \tau)$ and $\mathsf{Set}\,(\mathsf{f}, \tau)$, are straightforward. For example, $\tau_0 \leq \mathsf{Get}\,(\mathsf{f}, \tau)$ accesses field f of τ_0 and propagates the result to τ. For effects we introduce Havoc, which takes an environment argument Γ. This flow involves variables that get updated (as incoming effect), so that their potential refinements in Γ are invalidated. This will be discussed later on in greater detail. The precise usage of each of these uses will become clearer in Section 2.4.2.

2.4 Constraint System

We present the static semantics of our formal fragment by means of a constraint generating type inference scheme. Our constraints encode type safety obligations that arise as values flow to operations through the program.

2.4.1 Constraint Generation

The core type inference judgments for expressions and statements in FLOWCORE are:

$$\Gamma \vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C \qquad\qquad \Gamma \vdash s : \epsilon \dashv \Gamma' \triangleright C$$

The derivation of a judgment relies on a set of constraints C as proof obligations, which appear on the right of the $\triangleright$ symbol. For both expressions and statements this judgment is *flow-sensitive* which is achieved by introducing an output environment Γ', in addition to the input environment Γ. The set of variable names assigned in e or s is modeled by ϵ. The case of expressions has two additional byproducts: a *type* τ and a *predicate mapping* ψ. The latter includes bindings from names to predicates that must hold when e is truthy, and symbolic operations over them (explained later):

ψ	$::=$	$\varnothing$	empty mapping
	$\mid$	$x \mapsto P$	variable binding
	$\mid$	$\psi_1 \wedge \psi_2$	conjunction
	$\mid$	$\psi_1 \vee \psi_2$	disjunction
	$\mid$	$\neg\psi$	negation
	$\mid$	$\psi \backslash \epsilon$	exclude effect

Below we describe in more detail the constraint generation, starting from the rules regarding handling of variables, and function definitions and calls (Figure 2.6).

Variables. The rules for reading and assigning a local variable (CG-VAR and CG-ASSIGN) involve looking up and updating the current type for the variable in the outgoing environment. This part is what makes this system *flow-sensitive*. A flow-insensitive system would use a single environment for each judgment. The assigned type would be merged to the same type used for the variable under update in the first place, making it less precise. In addition, reading a variable introduces a truthy predicate on it. This is useful under specific contexts such as when the variable is used as the condition part of an if-branch. Conversely, writing a variable forgets any refinement coming from expression e that concerns x.

Functions. Rule CG-FUN handles arrow functions by approximating the environment at

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C}$

$$\frac{}{\Gamma \vdash n : b_n \,;\, \bot \,;\, \varnothing \dashv \Gamma \triangleright \varnothing}\ [\text{CG-Const}] \qquad \frac{\Gamma(x) = \tau^{\alpha}}{\Gamma \vdash x : \tau \,;\, \bot \,;\, x \mapsto \mathsf{truthy} \dashv \Gamma \triangleright \varnothing}\ [\text{CG-Var}]$$

$$\frac{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C \qquad \Gamma'(x) = {\tau_0}^{\alpha}}{\Gamma \vdash x = e : \tau \,;\, \epsilon \sqcup x \,;\, \psi \backslash x \dashv \Gamma'[x \mapsto \tau^{\alpha}] \triangleright C \cup \{\tau \leq \alpha\}}\ [\text{CG-Assign}]$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma_1 = \Gamma_0,\, x : \alpha^{\alpha},\, \mathsf{locals}(s) \qquad \mathsf{erase}(\Gamma) = \Gamma_0 \qquad \Gamma_1 \vdash \{s\,;\, \texttt{return}\ e\} : \tau \,;\, \epsilon \dashv \Gamma_2 \triangleright C}{\Gamma \vdash (x) \Rightarrow \{s\,;\, \texttt{return}\ e\} : \alpha \xrightarrow{\epsilon} \tau \,;\, \bot \,;\, \varnothing \dashv \Gamma \triangleright C}\ [\text{CG-Fun}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \,;\, \epsilon_1 \,;\, \psi_1 \dashv \Gamma_1 \triangleright C_1 \\ \Gamma_1 \vdash e_2 : \tau_2 \,;\, \epsilon_2 \,;\, \psi_2 \dashv \Gamma_2 \triangleright C_2 \qquad \alpha, \phi \text{ fresh} \qquad \mathsf{widen}(\Gamma_2) = \Gamma_3 \triangleright C_w \\ \epsilon_1 \sqcup \epsilon_2 \sqcup \phi = \epsilon \qquad C_1 \cup C_2 \cup C_w \cup \left\{ \phi \leq \mathsf{Havoc}\,(\Gamma_3)\,,\, \tau_1 \leq \mathsf{Call}(\tau_2 \xrightarrow{\phi} \alpha) \right\} = C\end{array}}{\Gamma \vdash e_1\,(e_2) : \alpha \,;\, \epsilon \,;\, \varnothing \dashv \Gamma_3 \triangleright C}\ [\text{CG-Call}]$$

Figure 2.6. Expression Constraint Generation in FLOWCORE (Variables and Functions)

the beginning with the *flow-insensitive erasure* of the current environment (since we do not know where this function will be called). The meta-function erase computes this new environment by mapping each $x : \tau^{\alpha}$ to $x : \alpha^{\alpha}$ (Figure 2.7). In addition, to capture the hoisting of variables defined within the scope of the function to the beginning of the function body, we introduce the meta-function locals that takes as argument a statement and returns a mapping of all local variables to the undefined type. The inferred arrow type carries the effect of the body of the function. Note that due to α-renaming we retain full precision even though we are including the symbol x of the function's parameter in effect ϵ (in the event that it gets updated in the arrow body). The reason is that by performing this transformation, identifier x cannot appear outside the scope of the arrow function. As such it cannot affect the refining process. As an optimization we could consider removing it to keep our effect as minimal as possible.

Calls. Rule CG-CALL handles calls. We approximate the outgoing environment with a *flow-sensitive widening* of the current environment (instead of pessimistically erasing everything in scope). The meta-function widen (Figure 2.7) computes this new environment Γ' by mapping each $x : \tau^{\alpha}$ to $x : \beta^{\alpha}$ where β is a fresh type variable such that $\tau \leq \beta \leq \alpha$. For any variable x that

Erase $\boxed{\mathsf{erase}(\Gamma) = \Gamma'}$

$$\frac{}{\mathsf{erase}(\cdot) = \cdot}\ [\text{T-ERASE-E}] \qquad \frac{\mathsf{erase}(\Gamma) = \Gamma'}{\mathsf{erase}(\Gamma,\, x\colon \tau^{\alpha}) = \Gamma',\, x\colon \alpha^{\alpha}}\ [\text{T-ERASE-C}]$$

Widen $\boxed{\mathsf{widen}(\Gamma) = \Gamma' \triangleright C}$

$$\frac{}{\mathsf{widen}(\cdot) = \cdot \triangleright \varnothing}\ [\text{T-WIDEN-E}]$$

$$\frac{\mathsf{widen}(\Gamma) = \Gamma_0 \triangleright C_0 \qquad \beta \text{ fresh}}{\mathsf{widen}(\Gamma,\, x\colon \tau^{\alpha}) = \Gamma_0,\, x\colon \beta^{\alpha} \triangleright C_0 \cup \{\tau \leq \beta,\, \beta \leq \alpha\}}\ [\text{T-WIDEN-C}]$$

Figure 2.7. Auxiliary Meta-functions for Function Logistics in FLOWCORE

gets assigned during the function call, we must fall back to its erasure, *i.e.* we must flow α to β. For now this is achieved by flowing the effect ϕ of the call to $\mathsf{Havoc}\,(\Gamma')$. The actual erasure happens later at constraint propagation (Section 2.4.2). At this point, the type of the receive function is not known, so the incoming effect remains abstract. As we show in Section 2.4.2, when a function type flows to $\mathsf{Call}(\tau_2 \xrightarrow{\phi} \alpha)$, the effect ϕ is instantiated with the actual effect variables x carried over by the incoming function type. These variables trigger the erasure.

Environment Operations. Before delving into the remaining typing rules, we introduce some operations on environments (Figure 2.8).

Environment join ($\sqcup$) is a commutative operator that computes the *least upper bound* of a pair of environments with the same domain. Type entries bound to the same symbol in the input environments need to refer to the same program variable. This requirement allows us to assume that the general type of a variable x bound in both environments will be the same.

The next operation we define is *environment refinement* ($::$). The semantics of a refinement ψ is defined by how it refines environments, *i.e.* a constraint-producing judgment of the form $\Gamma :: \psi = \Gamma' \triangleright C$, where an environment Γ is strengthened by the predicates in ψ and result in an environment Γ', potentially including fresh variables that are constrained in C. When ψ is $x \mapsto P$, we update the relevant binding in the environment Γ to a fresh type β that is the result of the predicate refinement of the initial type τ with P (Rule ENV-REF-BIND). The rules that handle the typical logical operators (ENV-REF-AND and ENV-REF-OR) are straightforward. The latter rule

Environment Join $\boxed{\Gamma_1 \sqcup \Gamma_2 = \Gamma}$

$$\frac{}{\cdot \sqcup \cdot = \cdot}\ [\textsc{Env-Join-E}]$$

$$\frac{\Gamma_1 \sqcup \Gamma_2 = \Gamma}{(\Gamma_1,\, x\colon {\tau_1}^{\alpha}) \sqcup (\Gamma_2,\, x\colon {\tau_2}^{\alpha}) = \Gamma,\, x\colon (\tau_1 \sqcup \tau_2)^{\alpha}}\ [\textsc{Env-Join-C}]$$

Environment Refinement $\boxed{\Gamma :: \psi = \Gamma' \triangleright C}$

$$\frac{}{\Gamma :: \varnothing = \Gamma \triangleright \varnothing}\ [\textsc{Env-Ref-E}]$$

$$\frac{\Gamma(x) = \tau^{\alpha} \qquad \beta \text{ fresh}}{\Gamma :: x \mapsto P = \Gamma[x \mapsto \beta^{\alpha}] \triangleright \{\tau \leq \mathsf{Pred}(P, \beta)\}}\ [\textsc{Env-Ref-Bind}]$$

$$\frac{\Gamma :: \psi_1 = \Gamma_1 \triangleright C_1 \qquad \Gamma_1 :: \psi_2 = \Gamma_2 \triangleright C_2}{\Gamma :: (\psi_1 \wedge \psi_2) = \Gamma_2 \triangleright C_1 \cup C_2}\ [\textsc{Env-Ref-And}]$$

$$\frac{\Gamma :: \psi_1 = \Gamma_1 \triangleright C_1 \qquad \Gamma :: \psi_2 = \Gamma_2 \triangleright C_2 \qquad \Gamma_1 \sqcup \Gamma_2 = \Gamma_3}{\Gamma :: (\psi_1 \vee \psi_2) = \Gamma_3 \triangleright C_1 \cup C_2}\ [\textsc{Env-Ref-Or}]$$

$$\frac{\Gamma :: \psi = \Gamma_1 \triangleright C_1 \qquad \mathsf{widen}(\Gamma_1) = \Gamma_2 \triangleright C_2 \qquad \Gamma_3 = \{x\colon \beta^{\tau} \mid x\colon \tau^{\alpha} \in \Gamma,\, x\colon \beta^{\alpha} \in \Gamma_2\}}{\Gamma :: \psi \backslash \epsilon = \Gamma_2 \triangleright C_1 \cup C_2 \cup \{\epsilon \leq \mathsf{Havoc}(\Gamma_3)\}}\ [\textsc{Env-Ref-Eff}]$$

Figure 2.8. Auxiliary Environment Operations in FLOWCORE

depends on environment joins that were introduced earlier.

Refinements can be invalidated by effects. In Rule ENV-REF-EFF, we first refine Γ by ψ, and then apply the effect ϵ through the "havoc" mechanism on the resulting environment Γ_1. There is a slight discrepancy in the way this mechanism is applied in this case, since we only want to revert the effect of the refinement caused by ψ, and not fall back to the most general type. If "havoc" is triggered, then for every variable x bound in Γ_3, that happens to reach effect ϵ, we only flow type τ (that x was bound to in Γ before the refinement) to β, instead of the most general type α.

Finally, we can have refinements with logical connectives. The negation of $x \mapsto P$ is simply $x \mapsto \neg P$. Otherwise, we push negations inward as much as possible, by applying the

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C}$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \,;\, \epsilon_1 \,;\, \psi_1 \dashv \Gamma_1 \triangleright C_1 \\ \Gamma_1 :: \psi_1 = \Gamma_1' \triangleright C_2 \qquad \Gamma_1' \vdash e_2 : \tau_2 \,;\, \epsilon_2 \,;\, \psi_2 \dashv \Gamma_2 \triangleright C_2 \\ \alpha_1, \alpha \text{ fresh} \qquad (\psi_1 \backslash \epsilon_2) \wedge \psi_2 = \psi \qquad \Gamma_1 :: \neg\psi_1 = \Gamma_1'' \triangleright C_4 \qquad \Gamma_1'' \sqcup \Gamma_2 = \Gamma'\end{array}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \alpha_1 \sqcup \tau_2 \,;\, \epsilon_1 \sqcup \epsilon_2 \,;\, \psi \dashv \Gamma' \triangleright \bigcup C_i \cup \{\tau_1 \leq \mathsf{Pred}(\mathsf{falsy}, \alpha_1)\}} \text{ [CG-And]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \,;\, \epsilon_1 \,;\, \psi_1 \dashv \Gamma_1 \triangleright C_1 \\ \Gamma_1 :: \neg\psi_1 = \Gamma_1' \triangleright C_2 \qquad \Gamma_1' \vdash e_2 : \tau_2 \,;\, \epsilon_2 \,;\, \psi_2 \dashv \Gamma_2 \triangleright C_3 \\ \alpha_1, \alpha \text{ fresh} \qquad (\psi_1 \backslash \epsilon_2) \vee \psi_2 = \psi \qquad \Gamma_1 :: \psi_1 = \Gamma_1'' \triangleright C_4 \qquad \Gamma_1'' \sqcup \Gamma_2 = \Gamma'\end{array}}{\Gamma \vdash e_1 \,||\, e_2 : \alpha_1 \sqcup \tau_2 \,;\, \epsilon_1 \sqcup \epsilon_2 \,;\, \psi \dashv \Gamma' \triangleright \bigcup C_i \cup \{\tau_1 \leq \mathsf{Pred}(\mathsf{truthy}, \alpha_1)\}} \text{ [CG-Or]}$$

$$\frac{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C}{\Gamma \vdash \,!\, e : \texttt{boolean} \,;\, \epsilon \,;\, \neg\psi \dashv \Gamma' \triangleright C} \text{ [CG-Not]} \qquad \frac{}{\Gamma \vdash p(x) : \texttt{boolean} \,;\, \bot \,;\, x \mapsto p \dashv \Gamma \triangleright \varnothing} \text{ [CG-Pred]}$$

Figure 2.9. Expression Constraint Generation in FlowCore (Logical Operations)

following laws:

$$\neg(\psi_1 \wedge \psi_2) = \neg\psi_1 \vee \neg\psi_2$$

$$\neg(\psi \backslash \epsilon) = \neg\psi \backslash \epsilon$$

$$\neg(\psi_1 \vee \psi_2) = \neg\psi_1 \wedge \neg\psi_2$$

$$\neg(\neg\psi) = \psi$$

Logical operations. The typing rules of Figure 2.9 are interesting for their effect on predicate refinement.

In Rule CG-And, e_2 is analyzed under the refinement ψ_1 (since otherwise it would not be evaluated). The type inferred for the entire expression contains components from both e_1 and e_2. From the former is contains type α_1 that is a version of τ_1 refined by the falsy predicate, since it corresponds to the case where e_1 is actually falsy. From the latter it includes the type τ_2 as is. For the output environment we follow a similar strategy. The component that corresponds to e_1's output environment will be refined with $\neg\psi_1$, since otherwise we would be using the environment corresponding to e_2. With respect to the output predicate mapping, parts of ψ_1 that apply on names written in e_2 are forgotten when taking the conjunction with ψ_2.

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C}$

$$\frac{\Gamma \equiv \Gamma_0 \qquad \forall i \in [1,n] \,.\, \Gamma_{i-1} \vdash e_i : \tau_i \fatsemi \epsilon_i \fatsemi \psi_i \dashv \Gamma_i \triangleright C_i \qquad \alpha_i \text{ fresh}}{\Gamma \vdash \{f_1 : e_1, \ldots, f_n : e_n\} : \{f_1 : \alpha_1, \ldots, f_n : \alpha_n\} \fatsemi \bigsqcup \epsilon_i \fatsemi \varnothing \dashv \Gamma_n \triangleright \bigcup C_i \cup \bigcup \{\tau_i \leq \alpha_i\}} \text{ [CG-Rec]}$$

$$\frac{\Gamma \vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C \qquad \alpha \text{ fresh}}{\Gamma \vdash e.f : \alpha \fatsemi \epsilon \fatsemi \varnothing \dashv \Gamma' \triangleright C \cup \{\tau \leq \mathsf{Get}(f, \alpha)\}} \text{ [CG-FldRd]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 \vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv \Gamma_2 \triangleright C_2}{\Gamma \vdash e_1.f = e_2 : \tau_2 \fatsemi \epsilon_1 \sqcup \epsilon_2 \fatsemi \varnothing \dashv \Gamma_2 \triangleright C_1 \cup C_2 \cup \{\tau_1 \leq \mathsf{Set}(f, \tau_2)\}} \text{ [CG-FldWr]}$$

Figure 2.10. Expression Constraint Generation in FlowCore (Records)

Rule CG-Or is the dual of the above rule, and works similarly. Finally, rules CG-Not and CG-Pred are straightforward. The former just negates the refinement and the latter introduces a refinement from a runtime test p.

Records. The rules of Figure 2.10 for record type inference are mostly routine. During record creation the initializer types flow to the newly constructed record literal type. Subsequent assignments of type τ to a field f widen the type of f by introducing flows to the use $\mathsf{Set}(f, \tau)$.

In practice, Flow follows a slightly stricter approach. It "fixes" the type of an object at initialization and checks that all subsequent writes adhere to this type. This essentially amounts to checking for type annotations which is out of scope in this section of type inference.

Statements. The main difference compared to the respective expression rule is the omission of the assigned type and the refinement. Rule CG-VarDecl is a simplified version of the assignment rule seen earlier. Rule CG-If handles conditional statements. This rule uses the refinement ψ for the conditional expression e to refine the environments that are used to check each branch, with the appropriate sign in each case. The output environment is the join of the environments at the end of each branch.

Example

We now examine how the rules of Figures 2.6 – 2.11 handle the code in lines 13 – 37 in Figure 2.2. In the following we keep the produced type bindings on the left and constraint sets on the right. Whenever, a general type (exponent) is not made explicit, this means that it's not

Statement Constraint Generation $\boxed{\Gamma \vdash s : \epsilon \dashv \Gamma' \triangleright C}$

$$\frac{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C}{\Gamma \vdash e : \epsilon \dashv \Gamma' \triangleright C}\ [\text{CG-Exp}]$$

$$\frac{\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C \qquad \Gamma'(x) = {\tau_0}^{\alpha}}{\Gamma \vdash \texttt{var}\ x = e : \epsilon \sqcup x \dashv \Gamma'[x \mapsto \tau^{\alpha}] \triangleright C \cup \{\tau \leq \alpha\}}\ [\text{CG-VarDecl}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau \,;\, \epsilon \,;\, \psi \dashv \Gamma' \triangleright C_1 \qquad \Gamma' :: \psi = \Gamma_1 \triangleright C_2 \qquad \Gamma_1 \vdash s_1 : \epsilon_1 \dashv \Gamma_1' \triangleright C_3 \\ \Gamma' :: \neg\psi = \Gamma_2 \triangleright C_4 \qquad \Gamma_2 \vdash s_2 : \epsilon_2 \dashv \Gamma_2' \triangleright C_5 \qquad \Gamma_1' \sqcup \Gamma_2' = \Gamma'' \qquad \alpha \text{ fresh}\end{array}}{\Gamma \vdash \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} : \epsilon \sqcup \epsilon_1 \sqcup \epsilon_2 \dashv \Gamma'' \triangleright \bigcup C_i}\ [\text{CG-If}]$$

$$\frac{\Gamma \vdash s_1 : \epsilon_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 \vdash s_2 : \epsilon_2 \dashv \Gamma_2 \triangleright C_2}{\Gamma \vdash s_1 \,;\, s_2 : \epsilon_1 \sqcup \epsilon_2 \dashv \Gamma_2 \triangleright C_1 \cup C_2}\ [\text{CG-Seq}]$$

Figure 2.11. Statement Constraint Generation in FlowCore

important for that particular binding. Also, to avoid clutter, we do not define a new environment for each program point, but rather introduce different versions for variables that get updated or refined.

By applying Rule CG-Rec on line 13:

$$\texttt{nil} : \{\texttt{kind} : \alpha_1\} \qquad C \supseteq \{\,\texttt{"nil"} \leq \alpha_1\,\} \tag{2.1}$$

Here, `"nil"` is the string literal type denoting the exact string `"nil"`. For the function `cons` (lines 14 – 16) we get

$$\texttt{cons} : (\alpha_2, \alpha_3) \to O \qquad C \supseteq \{\,\texttt{"cons"} \leq \alpha_4,\ \alpha_2 \leq \alpha_5,\ \alpha_3 \leq \alpha_6\,\} \tag{2.2}$$

where $O \doteq \{\texttt{kind} : \alpha_4, \texttt{head} : \alpha_5, \texttt{tail} : \alpha_6\}$. We also define $\tau_{\texttt{cons}} \doteq (\alpha_2, \alpha_3) \to O$. The function's effect is empty, so omitted here. Moving on to function `sum`, before checking its body we introduce bindings for the (recursive) function itself and its parameter:

$$\texttt{sum} : \alpha_7 \to \tau_r,\ \texttt{list} : \alpha_7 \tag{2.3}$$

We define $\tau_{\texttt{sum}} \doteq \alpha_7 \to \tau_5$. Checking the conditional in line 19, `list` gets a more precise type, and is referred to as $\texttt{line}_1$ inside the then-branch:

$$\texttt{list}_1 \colon \beta_7 \qquad\qquad C \supseteq \{\, \alpha_7 \leq \mathsf{Pred}(p_c,\, \beta_7) \,\} \tag{2.4}$$

Here, $p_c \doteq \star.\texttt{kind} \texttt{ === } \texttt{"cons"}$ is the predicate of exact equality of the field `kind` with the string `"cons"`. The uses of `list` in line 20 produce the following constraints (here we focus on the interesting uses *i.e.* the two field accesses and the call):

$$C \supseteq \left\{ \begin{array}{l} \beta_7 \leq \mathsf{Get}\,(\texttt{head}, \gamma_1), \\ \beta_7 \leq \mathsf{Get}\,(\texttt{tail}, \gamma_2), \\ \tau_{\texttt{sum}} \leq \mathsf{Call}(\gamma_2 \to \delta_1) \end{array} \right\} \tag{2.5}$$

We omit the constraints pertinent to the return statements, since they are not crucial in this example. The compound calls in line 25 further produce the constraints (starting from deeper nesting levels):

$$C \supseteq \{\, \tau_{\texttt{cons}} \leq \mathsf{Call}((\texttt{number}, \{\texttt{kind} : \alpha_1\}) \to \delta_2) \,\} \tag{2.6}$$

$$C \supseteq \{\, \tau_{\texttt{cons}} \leq \mathsf{Call}((\texttt{number}, \delta_2) \to \delta_3) \,\} \tag{2.7}$$

$$C \supseteq \{\, \tau_{\texttt{sum}} \leq \mathsf{Call}(\delta_3 \to \delta_4) \,\} \tag{2.8}$$

In function `merge`, let x_1 correspond to the initial value for `x` and x_2 to the value after the update in line 28. Below, the first three constraints correspond to the use of the `||` operator and the last one to the field access in line 29:

$$x_1 \colon {\alpha_8}^{\alpha_8},\; x_2 \colon {\alpha_{11}}^{\alpha_8} \qquad C \supseteq \left\{ \begin{array}{l} \alpha_8 \leq \mathsf{Pred}(\mathsf{truthy},\, \beta_8), \\ \beta_8 \sqcup \tau_{\texttt{nil}} \leq \alpha_{11}, \\ \alpha_{11} \leq \mathsf{Get}\,(\texttt{kind}, \alpha_{10}) \end{array} \right\} \tag{2.9}$$

Finally, function `havoc` in lines 32–37 is similar to `merge` (so we won't repeat the common parts), but additionally, defines a function `reset`, that assigns `null` to `x`. Crucially, the type of `x` inside `reset` has been erased to α_8:

$$\texttt{reset}\colon () \xrightarrow{\texttt{x}} \texttt{void} \qquad\qquad C \supseteq \{\, \texttt{null} \leq \alpha_8 \,\} \tag{2.10}$$

This time the call to `reset` in line 35 needs to handle the function's effect, so a fresh variable ϕ is generated:

$$C \supseteq \left\{ \begin{array}{l} () \xrightarrow{\texttt{x}} \texttt{void} \leq \mathsf{Call}(() \xrightarrow{\phi} \texttt{void}), \\ \phi \leq \mathsf{Havoc}\,(\Gamma[\,\texttt{x} \mapsto \alpha_{11}{}^{\alpha_8}\,]) \end{array} \right\} \tag{2.11}$$

For the moment, we have merely constructed a flow network, but haven't reached any critical conclusions. In the next section, we'll see how we can use these facts to discover inconsistencies, and what guarantees we get if we don't find any.

2.4.2 Propagation

Thinking of our system as a dataflow analysis framework, constraint generation amounts to setting up a flow network. The next step is to allow the system to stabilize under a set of appropriate flow functions. This latter part is called *constraint propagation* and corresponds to exploring *all* potential data-flow paths and finding inconsistencies in them. Decomposing complex constraints intro simpler ones is done by the rules shown in Figure 2.12. We say that a constraint set C is in *closed form*, if it is closed with respect to these rules. In practice, we keep our constraint sets in closed form at all times during constraint generation; that is, for every new constraint that gets generated, we apply all eligible propagation rules until we reach a fixpoint.

If we consider the elements of C as subtyping constraints, then these rules amount to subtyping rules. Rules CP-TRANS-T and CP-TRANS-E express transitivity for types and effect accordingly. CP-JOIN-T and CP-JOIN-E decompose as usual flows from joins of elements.

Rule CP-CALL decomposes a flow of an arrow type to a calling context, into flows of (i) the argument's type τ_1' to the parameter type τ_1, (ii) the return type τ_2 to the call-site's type τ_2', and (iii) the function's effect ϵ to the call's effect ϵ'. This last byproduct often triggers the "havoc" mechanism, which carries out the task of applying a function's effect on the variables that are updated by it.

$$\{\tau \leq \alpha,\, \alpha \leq u_\tau\} \subseteq C \implies \tau \leq u_\tau \in C \qquad \text{(CP-TRANS-T)}$$

$$\{\epsilon \leq \phi,\, \phi \leq u_\epsilon\} \subseteq C \implies \epsilon \leq u_\epsilon \in C \qquad \text{(CP-TRANS-E)}$$

$$\tau_1 \sqcup \tau_2 \leq u_\tau \in C \implies \{\tau_1 \leq u_\tau,\, \tau_2 \leq u_\tau\} \subseteq C \qquad \text{(CP-JOIN-T)}$$

$$\epsilon_1 \sqcup \epsilon_2 \leq u_\epsilon \in C \implies \{\epsilon_1 \leq u_\epsilon,\, \epsilon_2 \leq u_\epsilon\} \subseteq C \qquad \text{(CP-JOIN-E)}$$

$$\tau_1 \xrightarrow{\epsilon} \tau_2 \leq \mathsf{Call}(\tau_1' \xrightarrow{\epsilon'} \tau_2') \in C \implies \left\{\tau_1' \leq \tau_1,\, \tau_2 \leq \tau_2',\, \epsilon \leq \epsilon'\right\} \subseteq C \qquad \text{(CP-CALL)}$$

$$x \leq \mathsf{Havoc}\,(\Gamma,\, x\colon \tau^\alpha) \in C \implies \alpha \leq \tau \in C \qquad \text{(CP-HAVOC)}$$

$$\dot{\tau} \leq \mathsf{Pred}(P,\, \alpha) \in C \wedge \mathsf{check}\,(\dot{\tau},\, P) \implies \dot{\tau} \leq \alpha \in C \qquad \text{(CP-P-BASE)}$$

$$\left\{\tau \leq \alpha,\, \tau'\langle\alpha\rangle^+ \leq \mathsf{Pred}(P,\, \beta)\right\} \subseteq C \implies \tau'\langle\tau\rangle^+ \leq \mathsf{Pred}(P,\, \beta) \in C \qquad \text{(CP-P-TRANS)}$$

$$\{\ldots, f : \alpha, \ldots\} \leq \mathsf{Get}\,(f, \beta) \in C \implies \alpha \leq \beta \in C \qquad \text{(CP-GET)}$$

$$\{\ldots, f : \alpha, \ldots\} \leq \mathsf{Set}\,(f, \tau) \in C \implies \tau \leq \alpha \in C \qquad \text{(CP-SET)}$$

Figure 2.12. Constraint Propagation in FLOWCORE

In particular, Rule CP-HAVOC, handles the case where a concrete effect, *i.e.* a variable x (that gets updated in a function), reaches a havoc operation (generated at a call site) on an environment with a binding on x (that is the environment at the call site). Effectively, this corresponds to erasing the type of the binding $x\colon \tau^\alpha$, by generating a flow from the flow-insensitive type α to τ. Note that this process may happen far away from the actual call-cite, which exemplifies the global character of the type inference. An observant reader might notice that a generated constraint of the form $\alpha \leq \tau$ violates our restriction on the form of constraints, namely that the right-hand side cannot be a general type τ. However, we have been careful when populating the arguments of the Havoc constructor. In both cases where it is introduced (CG-CALL and ENV-REF-EFF) it happens after a widening operation, which guarantees that the special type of an environment entry (and hence the right-hand side of the havoc-induced constraint) is a type variable.

Rule CP-P-BASE handles predicate refinement. The intuition here is that $\dot{\tau}$ should flow to α, if it succeeds in the check implied by P, *i.e.* if $\mathsf{check}\,(\dot{\tau}, P)$ is true. We have kept the representation of base predicates abstract, and so we will do with the definition of check. In

general, check should be able to decide if $\dot{\tau}$ satisfies P by inspecting its top-level constructor (for checks like `typeof` $\star$ `===` `"string"`), or by inspecting the type of some of $\dot{\tau}$'s fields.

Rule CP-P-TRANS is a technical one. It allows parts of types under refinement to be concretized. In $\tau'\langle\alpha\rangle$, the form $\tau'\langle\rangle$ is a *type context*, *i.e.* a type with a "hole" that is filled in with α, for example $\{f : \langle\rangle\}$. While rule CP-TRANS-T will fail to instantiate α, CP-P-TRANS allows type variables appearing under a type constructor (*e.g.* the object constructor) to be instantiated. However, not all substitutions are allowed, but only the ones where α is in a positive position with respect to type *polarity* [83, 29]. Section A.1.3 in the Appendix includes a formal definition of polarity and type contexts. The reason we require type variable α to appear in a positive position is to abide by our restriction that type joins cannot appear at the right-hand side of constraints. If we allowed the replacement of α from τ in any part of τ', this could potentially break this invariant in a later propagation. We will also see the importance of this rule in the upcoming example.

Finally, Rules CP-GET and CP-SET handling record field access and update are standard.

2.4.3 Consistency

The goal of running constraint generation and propagation is to eventually discover inconsistencies in the saturated constraint set. These effectively correspond to potential bugs in the use of the various operators, for example they could correspond to the case of a non-function value reaching the receiver position of a call. Below we present a formal description of consistency.

Definition 2.4.1 (Consistency)**.** *A closed constraint set is* consistent *if it does not contain any constraints in one of the forms:*

- $\dot{\tau} \leq \mathsf{Call}(\tau')$ *where* $\dot{\tau}$ *is* not *an arrow type (or an arrow-like type, e.g. the types of constructors objects in JavaScript). If a flow of this form is produced it would mean that a function call could be attempted with at non-function receiver.*
- $\dot{\tau} \leq \mathsf{Set}(f, \tau)$ *or* $\dot{\tau} \leq \mathsf{Get}(f, \tau)$ *where* $\dot{\tau}$ *is* not *an record type literal (or an object-like type) containing* f*.*

If our analysis finds an inconsistency, then this leads to an error report. Otherwise, if no inconsistency can be found then the input program enjoys the safety guarantees of Theorem 2.2.

Example

We continue where we left off in example of Section 2.4.1, by applying the rules from Figure 2.12 on C, in order to discover inconsistencies or prove the absence thereof.

Use of predicates. We start by applying CP-CALL on calls (2.5), (2.6), (2.7), (2.8), and the respective function definitions:

$$\mathsf{C} \supseteq \{\gamma_2 \leq \alpha_7, \tau_r \leq \delta_1\} \tag{2.12}$$

$$\mathsf{C} \supseteq \{\texttt{number} \leq \alpha_2, \{\texttt{kind} : \alpha_1\} \leq \alpha_3, \mathsf{O} \leq \delta_2\} \tag{2.13}$$

$$\mathsf{C} \supseteq \{\texttt{number} \leq \alpha_2, \delta_2 \leq \alpha_3, \mathsf{O} \leq \delta_3\} \tag{2.14}$$

$$\mathsf{C} \supseteq \{\delta_3 \leq \alpha_7, \tau_r \leq \delta_4\} \tag{2.15}$$

Now lets focus on the interesting case of handling the getters of (2.5). By transitivity (CP-TRANS-T) using (2.14), (2.15) and (2.4), the record type O flows to the predicate use:

$$\mathsf{C} \supseteq \{\{\texttt{kind} : \alpha_4, \texttt{head} : \alpha_5, \texttt{tail} : \alpha_6\} \leq \mathsf{Pred}(p_c, \beta_7)\} \tag{2.16}$$

We use Rule CP-P-TRANS on (2.2) and (2.16) to obtain:

$$\mathsf{C} \supseteq \{\{\texttt{kind} : \texttt{"cons"}, \texttt{head} : \alpha_5, \texttt{tail} : \alpha_6\} \leq \mathsf{Pred}(p_c, \beta_7)\} \tag{2.17}$$

This is now a successful test since the string literal type `"cons"` of field `kind` satisfies p_c and so:

$$\mathsf{C} \supseteq \{\{\texttt{kind} : \texttt{"cons"}, \texttt{head} : \alpha_5, \texttt{tail} : \alpha_6\} \leq \beta_7\} \tag{2.18}$$

Flow has thus discovered a path in which a "cons" object reaches the field accesses of line 20. However, this latest constraint has enabled new flows that could potentially cause inconsistencies, for example the recursive calls to `sum` on the `tail` of `list`. By (2.18) and (2.5), and applying CP-TRANS-T and CP-GET:

$$\mathsf{C} \supseteq \{\alpha_6 \leq \gamma_2\} \tag{2.19}$$

Indeed, by combining (2.13), (2.2), (2.19), (2.12) and (2.4) with CP-TRANS-T and the result with (2.1) with CP-P-TRANS :

$$\mathsf{C} \supseteq \{\{\texttt{kind} : \texttt{"nil"}\} \leq \mathsf{Pred}(\mathsf{p}_c, \beta_7)\} \tag{2.20}$$

This test, however, will fail, as it would at runtime, and so the "nil" object will not reach the getter for `head` or `tail` through α_6. Without the predicate refinement filtering out "nil" objects, we would have introduced a spurious error.

Refinements and Mutation. Last, we illustrate how Flow handles functions `merge` and `havoc`. We start by processing (2.11) with CP-CALL and then CP-HAVOC , which yields

$$\mathsf{C} \supseteq \{\alpha_8 \leq \alpha_{11}\} \tag{2.21}$$

This allows the `null` from the `reset` function to find its way to α_{11} from (2.10) and from there to the "get" operation through (2.9):

$$\mathsf{C} \supseteq \{\texttt{null} \leq \mathsf{Get}(\texttt{kind}, \alpha_{10})\} \tag{2.22}$$

This latter constraint signals a consistency violation, keeping Flow sound with respect to variable updates that invalidate prior refinements.

2.5 Runtime Semantics

Before we describe our safety result (Section 2.6) we present the runtime semantics for the formal fragment of Section 2.3. The semantics presented here is heavily based on that used by Rastogi et al. [87] that cover a subset of JavaScript, emphasizing on features of interest in each case, while abstracting away non-crucial features.

Runtime Values. Figure 2.13 contains the definitions for the runtime configurations. To account for heap-allocated values, we introduce *locations* ℓ that index runtime heaps. Together with constants they synthesize runtime *values*, which are normal form as far as execution is concerned.

Runtime State. There are three constituent parts that compose a *runtime state* S. The first part is the *heap* H, which includes bindings from locations to *heap values* $\dot{v}$, which in turn are either

e	::=	$\ldots$	**Runtime Expressions**
	|	ℓ	location
v	::=		**Values**
	|	n	constant
	|	ℓ	location
$\dot{v}$	::=		**Heap Values**
	|	v	value
	|	$\langle L, (x) \Rightarrow \{s\,;\,\texttt{return}\ e\}\rangle$	closure
	|	$\{f_1 : v_1, \ldots, f_n : v_n\}$	heap object
E	::=	$\langle\,\rangle \mid x = E \mid E(e) \mid \ell(E)$	**Evaluation Contexts**
	|	$E\,\&\&\,e \mid E \,\|\|\, e \mid !\,E$	
	|	$\{f_1 : v_1\,, \ldots, f_k : E_k\,, \ldots, f_n : e_n\}$	
	|	$E.f \mid E.f = e \mid v.f = E$	
	|	$\texttt{var}\ x = E \mid \texttt{if}\ (E)\ \{s_1\}\ \texttt{else}\ \{s_2\}$	
	|	$\texttt{return}\ E \mid E\,;\,s$	
H	::=	$\cdot \mid H, \ell \mapsto \dot{v}$	**Heaps**
X	::=	$\cdot \mid X, L.E$	**Stacks**
L	::=	$\cdot \mid L, x \mapsto \ell$	**Stores**
S	::=	$\langle H; X; L\rangle$	**States**
R	::=	$S; e \mid S; s \mid S; M$	**Configurations**

Figure 2.13. Runtime Definitions in FLOWCORE

values, closures, or heap objects. A *closure* is a pair containing a store L that binds all external variables available at the point of definition of the arrow function (capture by reference), and the function's code, which is a statement succeeded by a returned expression. The second part of the runtime state is the *stack* X, that contains a list of stack frames. Each stack frame includes a store containing the variables bound in the stack frame at the time execution left that frame, and an evaluation context E that holds the context that execution would jump into when returning to that stack frame. Evaluation contexts are defined in the usual way having the same structure as expressions or statements but with a hole $\langle\,\rangle$ at the position of the term that is about to be evaluated next. Finally, the runtime state includes a *store* L, that comprises bindings of variable

names to locations to allow closure to capture values by reference.

Runtime Configurations. We write our *runtime configurations* S (*i.e.* programs under execution) as pairs that contain a runtime state S, and a language term, which can either be an expression e, a statement s, or a function body `{s; return e}`. We conflate the notions of expressions and function bodies into a common notion using the symbol M, for compactness in stating our results.

2.5.1 Reduction Rules

Figures 2.14 and 2.15 contain a small-step operational semantics for programs in FLOW-CORE. The rules can have the following forms:

$$S;\, e \longrightarrow S';\, e' \qquad\qquad S;\, s \longrightarrow S';\, s'$$

Next we describe some of the most interesting rules. Rule RT-VAR shows the indirection in dereferencing variables. First the store L is looked up and then the resulting location is used to access the heap H. Similarly variable right have to go through the same process in Rule RT-ASGN.

When evaluating arrows, the current store R.L is saved as part of the created closure, along with the code of the function (Rule RT-ARROW). This store is restored when the function is called (Rule RT-CALL). The new store L' that will be used in the new stack frame also includes a binding for the function parameter x and bindings from all variables x_i defined in the body M, since their definition is hoisted to the top of the function body. We use metavariable locals to extract these variables. All new variables are bound to fresh locations ℓ_i. Locals have not been initialized yet, so their corresponding locations are bound to `undefined` in the initial heap H'.

The rest of the expression reduction rules are routine.

2.6 Metatheory

In order to prove type safety for our type system we first introduce a declarative type system that corresponds closely to the type inference system described in Section 2.4. Based on the declarative system we then formulate a type safety argument for the above language fragment via a progress and a preservation theorem [112], that connect type checking with the runtime semantics of Section 2.5. Essentially, we establish the fact that if a program has been

Expression Reduction Rules $\boxed{S;\, M \longrightarrow S';\, M'}$

$$\frac{\langle H;\, \cdot;\, L\rangle;\, e \longrightarrow \langle H';\, \cdot;\, L'\rangle;\, e'}{\langle H;\, X;\, L\rangle;\, E\langle e\rangle \longrightarrow \langle H';\, X;\, L'\rangle;\, E\langle e'\rangle}\ [\text{RT-ECTX}]$$

$$\frac{}{S;\, x \longrightarrow S;\, S.H(S.L(x))}\ [\text{RT-VAR}] \qquad \frac{H' = H[\,L(x) \mapsto v\,]}{\langle H;\, X;\, L\rangle;\, x = v \longrightarrow \langle H';\, X;\, L\rangle;\, v}\ [\text{RT-ASGN}]$$

$$\frac{\ell \text{ fresh} \qquad H' = H,\, \ell \mapsto \langle S.L,\, (x) \Rightarrow M\rangle}{S;\, (x) \Rightarrow M \longrightarrow S \triangleleft H';\, \ell}\ [\text{RT-ARROW}]$$

$$\frac{\begin{array}{c} H(\ell) = \langle L_0,\, (x) \Rightarrow M\rangle \qquad \ell', \overline{\ell_i} \text{ fresh} \qquad \overline{x_i} = \mathsf{locals}(M) \\ H' = H,\, \ell' \mapsto v,\, \overline{\ell_i \mapsto \texttt{undefined}} \qquad X' = X,\, L.E \qquad L' = L_0,\, x \mapsto \ell',\, \overline{x_i \mapsto \ell_i} \end{array}}{\langle H;\, X;\, L\rangle;\, E\langle \ell(v)\rangle \longrightarrow \langle H';\, X';\, L'\rangle;\, M}\ [\text{RT-CALL}]$$

$$\frac{S \equiv \langle H;\, X;\, L\rangle \qquad \dot{v} = H(L(x))}{S;\, p(x) \longrightarrow S;\, \delta_p(\dot{v})}\ [\text{RT-PRED-VAR}]$$

$$\frac{\mathsf{truthy}(v)}{S;\, v \,\&\&\, e \longrightarrow S;\, e}\ [\text{RT-AND-TRU}] \qquad \frac{\mathsf{falsy}(v)}{S;\, v \,\&\&\, e \longrightarrow S;\, v}\ [\text{RT-AND-FLS}]$$

$$\frac{\mathsf{truthy}(v)}{S;\, v \,||\, e \longrightarrow S;\, v}\ [\text{RT-OR-TRU}] \qquad \frac{\mathsf{falsy}(v)}{S;\, v \,||\, e \longrightarrow S;\, e}\ [\text{RT-OR-FLS}]$$

$$\frac{v' = \neg\mathsf{toBool}(v)}{S;\, !\, v \longrightarrow S;\, v'}\ [\text{RT-NEG}]$$

$$\frac{\ell \text{ fresh} \qquad H' = H,\, \ell \mapsto \{f_1 : v_1, \ldots, f_n : v_n\}}{S;\, \{f_1 : v_1, \ldots, f_n : v_n\} \longrightarrow S \triangleleft H';\, \ell}\ [\text{RT-RECORD}]$$

$$\frac{S.H(\ell) = \{\overline{f_i : v_i},\, f : v,\, \overline{f_j : e_j}\}}{S;\, \ell.f \longrightarrow S;\, v}\ [\text{RT-FLDRD}] \qquad \frac{H' = S.H[\,\ell \mapsto S.H(\ell)[\,f \mapsto v\,]\,]}{S;\, \ell.f = v \longrightarrow S \triangleleft H';\, v}\ [\text{RT-FLDWR}]$$

Figure 2.14. Operational Semantics of FLOWCORE (Expressions)

checked with the above algorithm and has been found consistent, then its execution will not lead to uncaught type errors (*e.g.* "undefined is not a function").

Statement Reduction Rules $\boxed{S;\ s \longrightarrow S';\ s'}$

$$\frac{H' = H[\ell \mapsto v]}{\langle H;\ X;\ L\rangle;\ \texttt{var}\ x = v \longrightarrow \langle H';\ X;\ L\rangle;\ \texttt{skip}}\ \text{[RT-Let]}$$

$$\frac{\mathsf{truthy}(v)}{S;\ \texttt{if}\ (v)\ \{s_1\}\ \texttt{else}\ \{s_2\} \longrightarrow S;\ s_1}\ \text{[RT-If-Tru]}$$

$$\frac{\mathsf{falsy}(v)}{S;\ \texttt{if}\ (v)\ \{s_1\}\ \texttt{else}\ \{s_2\} \longrightarrow S;\ s_2}\ \text{[RT-If-Fls]}$$

$$\frac{S.X = X',\ L.E \qquad S' = S.H;\ X';\ L}{S;\ \texttt{return}\ v \longrightarrow S';\ E\langle v\rangle}\ \text{[RT-Ret]}$$

$$\frac{}{S;\ \texttt{skip};\ s \longrightarrow S;\ s}\ \text{[RT-Skip]}$$

Figure 2.15. Operational Semantics of FlowCore (Statements)

2.6.1 Declarative Type System

This system assigns concrete types, *i.e.* types stripped off of type variables, to expressions and statements of FlowCore. In the following, the environments Δ and G both map variables to concrete types (not type entries like before). Δ has the same flow-sensitive behavior as before, while G is a flow-insensitive environment providing the most general type for each variable. The typing judgments for expressions and statements are:

$$\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta' \qquad\qquad \Delta \fatsemi G \Vdash s : \epsilon \dashv\!\Vdash \Delta'$$

The respective rules for these judgments are unsurprising and therefore deferred to Section A.2 of the appendix.

A substitution ρ maps type variables to concrete types, and is extended to environments in a point-wise manner. In Section A.1 we introduce subtyping for concrete types, which helps us map constraints c through a substitution ρ to subtyping relations over concrete types. We say that a substitution ρ satisfies a constraint set C if all subtyping constraints generated by mapping ρ over C are valid. In this case we write $\rho \vdash C$.

We argue about the soundness of our type inference system with respect the declarative system with the following lemma.

Lemma 2.1 (Type Inference Soundness)**.** *If*

(i) $\Gamma \vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C$

(ii) $\rho \vdash C$

then $\rho(\Gamma) \Vdash e : \rho(\tau) \fatsemi \rho(\epsilon) \fatsemi \psi \dashv\!\vdash \rho(\Gamma')$.

2.6.2 Type Safety

Before we state our type safety result for the declarative type system, we extend the type checking judgment to runtime configurations: $G \Vdash_\Sigma S; e : \tau$. Here G is a flow-insensitive environment mapping variables to their most general type throughout the entire program. The judgment is to be read as: under a *heap typing* Σ, mapping heap locations to types and a flow-insensitive environment G, a configuration $S; e$ is a assigned a type τ. Now we can finally state our type safety result.

Theorem 2.2 (Type Safety)**.** *For a configuration* $S; e$ *and heap typing* Σ*, if* $G \Vdash_\Sigma S; e : \tau$*, then:*

- ***(Preservation)*** *If* $S; e \longrightarrow S'; e'$ *, then there exists* Σ'*, such that* $G \Vdash_{\Sigma'} S'; e' : \tau'$.
- ***(Progress)*** *Either* e *is a normal form, or there exists a configuration* $S'; e'$ *such that* $S; e \longrightarrow S'; e'$.

Proof for the results of this section along with supporting lemmas can be found in the appendix.

2.7 Implementation of Type Inference

The implementation of Flow represents constraint sets as graphs, so we will use the terms constraint set and constraint graph interchangeably. In this section, we briefly discuss issues that arise from the representation of the constraint graph that enables an efficient computation of its closure, and conclude with a brief note on performance.

Implementing Unification. Let us refer to type and effect variables as "unknowns." Following Pottier [84], the constraint graph maps each unknown to a set of lower bounds and a set of upper bounds, each of which contains the unknown itself. The transitive propagation rules are specialized to exploit this structure to efficiently keep the constraint graph in closed form.

However, equality constraints are quite inefficient in this system: they are represented as a pair of subset constraints, which causes a cubic blowup in the transitive propagation rules. On the other hand, equality constraints are quite useful and common in Flow. They arise due to invariant typing of object properties, array elements, and type arguments of polymorphic classes. They directly model equations expressed by type aliasing. Finally, even though we formalize CP-HAVOC with a constraint of the form $\alpha \leq \tau$, we can replace it without loss of generality with $\alpha = \tau$.

To address the inefficiency, we generalize the constraint graph by considering each unknown to be in an equivalence class containing other unknowns it is unified with, and mapping each equivalence class to either "unresolved" bounds (as in Pottier), or to a "resolved" type or effect (as in unification). The transitive propagation rules generalize in a straightforward way. Overall, this simple optimization leads to $O(n)$ reduction in space and time complexity.

A note on performance. Behind the scenes, Flow relies on set-based analysis as a common low-level "assembly language" for encoding a wide variety of high-level analyses. Compared with pure unification, this affords far more precision, but is much less efficient (quasi-cubic vs. quasi-linear in program size). The key to Flow's speed is modularity: the ability to break the analysis into file-sized chunks that can be assembled later.

Fortunately, JavaScript is already written using files as modules, so we modularize our analysis simply by asking that modules have explicitly typed signatures. (We still infer types for the vast majority of code "local" to modules.) Coincidentally, developers consider this good software engineering practice anyway.

With modularity, we can aggressively parallelize our analysis. Furthermore, when files change, we can incrementally re-analyze only those files that depend on the changed files, and avoid re-analysis when their typed signatures have not changed. Together, these choices have helped scale the analysis to millions of lines of code.

Under the hood, Flow relies on a high-throughput low-latency systems infrastructure that enables distribution of tasks among parallel workers, and communication of results in parallel via shared memory. Combined with an architecture where the analysis of a codebase is updated automatically in the background on file system changes, Flow delivers near-instantaneous feedback as the developer edits and rebases code, even in a large repository.

2.8 Related Work

Work that directly relates to the tool and type checking techniques described in this chapter were discussed in Section 1.3.2. In this section we focus on two relevant topics that were not covered in the introduction: JavaScript semantics and constraint simplification.

JavaScript Semantics. The runtime semantics presented in Section 2.5 was adapted from the work of Rastogi et al. [87]. However, providing a complete, sound, thoroughly tested and executable semantics for a rapidly evolving language like JavaScript is a challenging task. Below we outline a brief overview of some of the main efforts to undertake it.

Early work by Herman and Flanagan [56] used ML as the specification language. However, they targeted ECMAScript 4, which was never approved as a standard. Maffeis et al. [70] define a non-mechanized semantics that covers almost all of ECMAScript 3. While handling a large portion of the language specification, this work lacks extensibility since any change to the semantics would have to be manually patched and so any comparison to implementation is impossible. Building up on this last work, Gardner et al. [45] reason about complex features of JavaScript by adapting ideas from separation logic and prove their reasoning sound. Guha et al. [50] present λ_{JS}, a small-step operational semantics for a core of JavaScript, to which they desugar ES3 code. This is extended to ES5 by Politz et al. [82], by accounting for accessors and `eval`.

Recent efforts focus on providing mechanized specifications of JavaScript. Bodin et al. [12] formulate the ES5 specification in the Coq proof assistant (JSCert) and extract a reference interpreter to OCaml (JSRef). Proving the interpreter correct with respect to the specification involved a considerable amount of labor. Park et al. [79] promise a more modest development effort. They provide a complete and executable semantics for ES5 (KJS) that uses the K framework [93]. Unlike previous work, KJS passes all of the core languages tests and offers modularity and extensibility.

Constraint Simplification. In Section 1.3.1 we discussed the foundations of constraint type inference [4, 106, 83, 37]. These research directions already include several simplification techniques [33, 37]. To further improve performance of inclusion constraint analyses, Fähndrich et al. [34] propose a technique for eliminating cycles in constraint graphs that is based on a non-standard graph representation called *inductive form*, and only traverses part of the paths

during the search for cycles. To address the problem of redundant paths in a constraint graph, Su et al. [98] propose *projection merging*, a technique intended to be used in conjunction with the above. In contrast, we directly implement unification constraints using union-find over a base representation of inclusion constraints.

Acknowledgements

This chapter in part is currently under submission for publication of the material as it may appear in **Fast and Precise Type Checking for JavaScript.** Chaudhuri, Avik; Vekris, Panagiotis. The dissertation author was an author of this paper.

Chapter 3

Trust, but Verify: Two-Phase Typing for Dynamic Languages

Higher-order constructs are increasingly adopted in *dynamic scripting* languages, as they facilitate the production of clean, correct and maintainable code. Consider, for example, the following (first-order) JavaScript function

```
1  function minIndexFO(a) {
2    if (a.length <= 0)
3      return -1;
4    var min = 0;
5    for (var i = 0; i < a.length; i++) {
6      if (a[i] < a[min])
7        min = i;
8    }
9    return min;
10 }
```

which computes the index of the minimum value in the array `a` by looping over the array, updating the `min` value with each index `i` whose value `a[i]` is smaller than the "current" `a[min]`. Modern dynamic languages let programmers factor the looping pattern into a higher-order `_reduce` function (Figure 3.1), which frees them from manipulating indices and thereby prevents the attendant "off-by-one" mistakes. Instead, the programmer can compute the minimum index by supplying an appropriate `f` to `reduce` as in `minIndex` also shown in Figure 3.1.

This trend towards abstraction and reuse poses a challenge to static program analyses: *how to precisely trace value relationships across higher-order functions and containers?* A variety of dataflow- or abstract interpretation- based analyses could be used to verify the safety of array accesses in `minIndexFO` by inferring the loop invariant that `i` and `min` are between `0` and `a.length`.

```
11 function _reduce(a, f, x) {
12   var res = x;
13   for (var i = 0; i < a.length; i++)
14     res = f(res, a[i], i);
15   return res;
16 }
17
18 function reduce(a, f, x) {
19   if (arguments.length === 3)
20     return _reduce(a, f, x);
21   return _reduce(a.slice(1), f, a[0]);
22 }
23
24 function minIndex(a) {
25   if (a.length <= 0)
26     return -1;
27   function step(min, cur, i) {
28     return cur < a[min] ? i:min;
29   }
30   return reduce(a, step, 0);
31 }
```

Figure 3.1. Computing the Minimum-valued Index with Higher-Order Functions

Alas, these analyses would fail on `minIndex`. The usual methods of procedure summarization apply to first-order functions, and it is not clear how to extend higher-order analyses like CFA to track the *relationships* between the values and closures that flow to `_reduce`.

An Approach: Refinement Types. Refinement types [113] hold the promise of a precise and compositional analysis for higher-order functions. Here, *basic* types are decorated with *refinement* predicates that constrain the values inhabiting the type. For example, we can define

$$\texttt{type idx}\langle\langle \texttt{x} \rangle\rangle = \{\, \nu \colon \texttt{number} \mid 0 \leq \nu \wedge \nu < \texttt{len(x)} \,\}$$

to denote the set of valid indices for an array `x` and can be used to type `_reduce` as

$$\texttt{\_reduce} :: \forall \alpha, \beta \,.\, (\texttt{a}\colon \alpha\texttt{[]}, \texttt{f}\colon (\beta, \alpha, \texttt{idx}\langle\langle \texttt{a} \rangle\rangle) \Rightarrow, \texttt{x}\colon \beta) \Rightarrow \beta$$

The above type is a precise *relational summary* of the behavior of `_reduce`: the higher-order `f` is

only invoked with valid indices for `a`. Consequently, `step` is only called with valid indices for `a`, which ensures array safety.

Problem: Value-based Overloading. A main attraction of dynamic languages is *value-based overloading*, where syntactic entities (*e.g.* variables) may be bound to multiple types at run-time, and furthermore, computations may be customized to particular types, by reflecting on the values bound to variables. For example, it is common to simplify APIs by overloading the `reduce` function to make the initial value `x` optional; when omitted, the first array element `a[0]` is used instead (Figure 3.1). Here, `reduce` really has *two* different function types: one with 3 parameters and another one with 2. Furthermore, `reduce` *reflects* on the size of `arguments` to select the behavior appropriate to the calling context.

Value-based overloading conflicts with a crucial prerequisite for refinements, namely that the language possesses an *unrefined* static type system that provides basic invariants about values which can then be refined using logical predicates. Unfortunately, as shown by `reduce`, to soundly establish basic typing we must reason about the logical relationships between values, which is exactly the problem we wished to solve via refinement typing. In other words, value-based overloading creates a chicken-and-egg problem: refinements require us to first establish basic typing, but the latter itself requires reasoning about values (and hence, refinements!).

Solution: Trust but Verify. We introduce *two-phased typing*, a new strategy for statically analyzing dynamic languages. The key insight is that we can completely decouple reasoning about *basic* types and *refinements* into distinct phases by converting "type errors" from the first phase into "assertion failures" for the second. Two-phase typing starts with a source language where value-based overloading is specified using *intersections* and (untagged) *unions* of the different possible (run-time) types.

The first phase performs classical, *i.e.* flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with `dead`-casts which allow the first phase to proceed, *trusting* that the expressions have the casted types. In other words, the first phase *elaborates* [31] the source language with intersection and (untagged) union types, into a target ML-like language with classical products, (tagged) sums and `dead`-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase carries out *refinement*, *i.e.* flow- and path-sensitive inference, to decorate the basic types (from the first phase)

with predicates that precisely track relationships about values, and uses the refinements to *verify* the casts and other properties, discharging the assumptions of the first phase.

For example, `reduce` is described as the intersection of two contexts, *i.e.* function types which take two and three parameters respectively. The trust-phase checks the body under both contexts (separately). In each context, one of the calls to `_reduce` is "ill-typed". In the context where the function takes two inputs, the call using `x` is undefined; when the function takes three inputs, there is a mismatch in the types of `f` and `a[0]`. Consequently, each ill-typed expression is wrapped with a *cast* which obliges the verify phase to prove that the call is dead code in that context, thereby verifying overloading in a cooperative manner.

Benefits. While it is possible to account for value-based overloading in a single phase, the currently known methods that do so are limited to the extremes of types and program logics. At one end, systems like Typed Racket [104] and Flow Typing [50] extend classical type systems to account for a fixed set of `typeof`-style tests, but cannot reason about general value tests (*e.g.* the size of `arguments`) that often appear in idiomatic code. At the other end, systems like System D [21] embed the typing relation in an expressive program logic, allowing general value tests, but give up on basic type structure, thereby sacrificing inference, causing a significant annotation overhead. In contrast, our approach separates the concerns of basic typing and reasoning about values, thereby yielding several concrete benefits by *modularizing* specification, verification and soundness.

- ***Specification:*** Instead of a fixed set of type-tests, two-phase typing handles complex value relationships which can be captured inside refinements in an expressive logic. Furthermore, the *expressiveness* of the basic type system and logics can be extended independently, *e.g.* to account for polymorphism, classes or new logical theories, directly yielding a more expressive specification mechanism.

- ***Verification:*** Two-phase typing enables the straightforward composition of simple type checkers (uncomplicated by reasoning about values) with program logics (relying upon the basic invariants provided by typing – *e.g.* the parametric polymorphism needed to verify `minIndex`). Furthermore, two-phase typing allows us to compose basic typing with abstract interpretation [91], which drastically lowers the annotation burden for using refinement types.

- ***Soundness:*** Finally, our elaboration-based approach makes it straightforward to establish soundness for two-phased typing. The first phase ignores values and refinements, so we can use classical methods to prove the elaborated target is "equivalent to" the source. The second phase uses standard refinement typing techniques on the well-typed elaborated target, and hence lets us directly reuse the soundness theorems for such systems [66] to obtain end-to-end soundness for two-phased typing.

Contributions. Concretely, in this chapter we make the following contributions. First, we informally illustrate (Section 3.1) how two-phase typing lets us statically analyze dynamic, value-based overloading patterns drawn from real-world code, where, we empirically demonstrate, value-based overloading is ubiquitous. Second, we formalize two-phase typing using a core calculus, TBV, whose syntax and semantics are detailed in Section 3.2. Third, we formalize the first phase (Section 3.3), which *elaborates* [31] a source language with value-based overloading into a target language with dead-casts in lieu of overloading. We prove that the elaborated target preserves the semantics of the source, *i.e.* the dead-casts fail iff the source would hit a type error at run time. Finally, we demonstrate how standard refinement typing machinery can be applied to the elaborated well-typed target (Section 3.4) to statically verify the dead-casts, yielding end-to-end soundness for our system.

3.1 Overview

We begin with an overview illustrating how we soundly verify value-based overloading using our novel two-phased approach.

3.1.1 Value-based Overloading

Consider the TypeScript code in Figure 3.2. The function `negate` behaves as follows. When a `number` is passed as input, indicated by passing in a *non-zero*, *i.e.* "truthy" `flag`, the function flips its sign by subtracting the input from `0`. Instead, when a `boolean` is passed in, indicated by a *zero*, *i.e.* "falsy" `flag`, the function returns the boolean negation. Hence, the calls made to assign `a` and `b` are legitimate and should be statically accepted. However, the calls made to assign `c` and `d` lead to run-time errors (assuming we eschew implicit coercions), and hence, should be rejected.

The function `negate` distils value-based overloading to its essence: a run-time test on

```
32 function negate(number, number): number;
33 function negate(number, boolean): boolean;
34 function negate(flag, x) {
35   if (flag) return 0-x;
36   return !x;
37 }
38
39 var a = negate(1,1);    // OK
40 var b = negate(0,true); // OK
41 var c = negate(0,1);    // ERR
42 var d = negate(1,true); // ERR
```

Figure 3.2. An Example Program with Value-Based Overloading

one parameter's value is used to determine the type of, and hence the operation to be applied to, another value. Of course in JavaScript, one could use a single parameter and the `typeof` operator for this particular simple case, and design analyses targeted towards a fixed set of type tests, *e.g.* using variants of the `typeof` operator [104, 50]. However, arbitrary value tests – such as tests of the size of `arguments` shown in `reduce` in Figure 3.1 – can be and are used in practice. Thus, we illustrate the generality of the problem and our solution *without* using the `typeof` operator (which is a special case of our solution).

Prevalence of Value-based Overloading. The code from Figure 3.1 is not a pathological toy example. It is adapted from the widely used D3 visualization library. The advent of TypeScript makes it possible to establish the prevalence of value-based overloading in real-world libraries, as it allows developers to specify overloaded signatures for functions. (Even though TypeScript does not verify those signatures, it uses them as trusted interfaces for external JavaScript libraries and code completion.) The Definitely Typed repository [1] contains TypeScript interfaces for a large number of popular JavaScript libraries. We analyzed the TypeScript interfaces to determine the prevalence of value-based overloading. Intuitively, every function or method with multiple (overloaded) signatures or optional arguments has an implementation that uses value-based overloading.

We summarize next the results of our study. On Table 3.1 we show the fraction of overloaded functions in the 10 benchmarks analyzed by Feldthaus *et al.* [35]. The data shows that over 25% of the functions in 4 of 10 libraries use value-based overloading, and an even

[1] http://definitelytyped.org

Table 3.1. The Prevalence of Value-Based Overloading. Libraries are taken from the survey of Feldthaus *et al.* [35]. **#Funs** is the number of functions in the signature, **%Ovl** is %-functions with *multiple* signatures, **%Opt** is %-functions with *optional* arguments, and **%Any** is %-functions with either of these features.

File	#Funs	%Ovl	%Opt	%Any
box2d	529	0	3	**3**
ace	484	1	5	**6**
pixi	123	0	12	**12**
fabricjs	371	5	9	**13**
threejs	1022	1	24	**24**
leaflet	414	12	38	**41**
underscore	344	25	34	**45**
sugar	446	29	37	**48**
d3	475	43	17	**52**
jquery	226	52	31	**67**

larger fraction is overloaded in libraries like jQuery and D3. On Figure 3.3 we summarize the occurrence of overloading across all the libraries in Definitely Typed. The data shows, for example, that in more than 25% of the libraries, *more than* 25% of the functions are overloaded with multiple types. The figure jumps to nearly 55% of functions if we also include optional arguments.

The signatures in Definitely Typed have not been soundly checked against[2] their implementations. Hence, it is possible that they mischaracterize the semantics of the actual code, but modulo this caveat, we believe the study demonstrates that value-based overloading is ubiquitous, and so to soundly and statically analyze dynamic languages, it is crucial that we develop techniques that can precisely and flexibly account for it.

3.1.2 Refinement Types

Types and Refinements. A basic refinement type T is a basic type, *e.g.* `number`, refined with a logical formula from an SMT decidable logic – for our purposes, the quantifier-free logic of uninterpreted functions and linear integer arithmetic (QF_UFLIA [96]). For example,

$$\{\nu\colon \texttt{number} \mid \nu \neq 0\}$$

[2] Feldthaus *et al.* [35] describe an effective but unsound inconsistency detector.

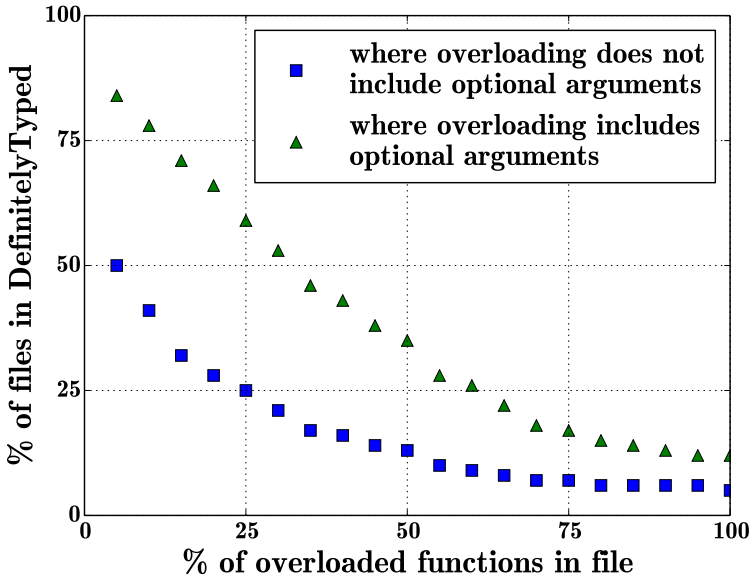

Figure 3.3. The Prevalence of Value-Based Overloading. Overloading across *all* files in Definitely Typed. A point (x, y) means y% of files have more than x% overloaded functions.

describes the *subset* of numbers that are non-zero. We write τ to abbreviate the trivially refined type $\{\nu\colon \tau \mid \mathit{true}\}$, *e.g.* `number` is an abbreviation for $\{\nu\colon \texttt{number} \mid \mathit{true}\}$.

Summaries: Function Types. We can specify the behavior of functions with refined function types, of the form

$$(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow T$$

where arguments are named x_i and have types T_i and the output is a T. In essence, the *input* types T_i specify the function's preconditions, and the *output* type T describes the postcondition. Furthermore, each input type and the output type can *refer to* the arguments x_i which yields precise function contracts. For example,

$$(x\colon \{\nu\colon \texttt{number} \mid 0 \leq \nu\}) \Rightarrow \{\nu\colon \texttt{number} \mid x < \nu\}$$

is a function type that describes functions that *require* a non-negative input, and *ensure* that the output is greater than the input.

Example. Returning to `negate` in Figure 3.2, we can define two refinements of `number`

```
type tt = {v: number | v ≠ 0}        "truthy" numbers
type ff = {v: number | v = 0}        "falsy" numbers
```

which are used to specify a refined type for `negate` shown on the left in Figure 3.4.

Problem: A Circular Dependency. While it is easy enough to specify a type signature, it is another matter to verify it, and yet another matter to ensure soundness. The challenge is that value-based overloading introduces a circular dependency between types and refinements. The soundness of basic types requires (*i.e.* is established by) the refinements, while the refinements themselves require (*i.e.* are attached to) basic types. In classical refinement systems like DML [113], basic types are established *without* requiring refinements. A classical refinement system is thus a conservative extension of the corresponding non-refined language, *i.e.* removing the refinements from a DML program, yields valid, well-typed ML. Unfortunately, value-based overloading removes this crucial property, posing a circular dependency between types and refinements.

Solution: Two-Phase Checking. We break the cycle by typing programs in two phases. In the first, we *trust* the basic types are correct and use them (ignoring the refinements) to elaborate source programs into a target overloading-free language. Inevitably, value-based overloading leads to "errors" when typing certain sub-expressions in the wrong context, *e.g.* subtracting a `boolean`-valued `x` from `0`. Instead of rejecting the program, the elaboration wraps ill-typed expressions with dead-casts, which are assertions stating the program is well-typed *assuming* those expressions are dead code. In the second phase we reuse classical refinement typing techniques to *verify* that the dead-casts are indeed unreachable, thereby discharging the assumptions made in the first phase.

3.1.3 Phase 1: Trust

The first phase *elaborates* the source program into an equivalent typed target language with two key properties: First, the target program is simply typed – *i.e.* has *no* union or intersection types, but just classical ML-style sums and products. Second, source-level type errors are elaborated to target-level dead-casts. The right side of Figure 3.4 shows the elaboration of

```
function negate(flag: tt, x: num): num
function negate(flag: ff, x: bool): bool
function negate(flag, x) {
  if (flag) return 0-x;
  return !x;
}

var a = negate(1, 1);    //OK
var b = negate(0, true); //OK
var c = negate(0, 1);    //ERROR
var d = negate(1, true); //ERROR
```

```
let negate1 (flag: tt) (x: num): num =
  if flag then 0-x
  else !dead(x)

let negate2 (flag: ff) (x: bool): bool =
  if flag then 0-dead(x)
  esle !x

let negate = (negate1, negate2)

let a = (fst negate) 1 1     (* OK *)
let b = (snd negate) 0 true  (* OK *)
let c = (fst negate) 0 1     (* ERROR *)
let d = (snd negate) 1 true  (* ERROR *)
```

Figure 3.4. Source (left) and Target (right) Program in First Phase Elaboration.

the source from the left side. While we formalize the elaboration declaratively using a single judgment form (Section 3.3), it comprises two different steps. Critically, each step, and hence the entire first phase, is *independent* of the refinements – they are simply carried along unchanged.

A. Clone. In the first step, we create separate clones of each overloaded function, where each clone is assigned a single conjunct of the original overloaded type. For example, we create two clones `negate1` and `negate2` respectively typed using the two conjuncts of the original `negate`. The binder `negate` is replaced with a *tuple* of its clones. Finally, each use of `negate` extracts the appropriate element from the tuple before issuing the call.

Since the trust phase must be independent of refinements, the overload resolution in this step uses *only* the basic types at the call-site to determine which of the two clones to invoke. For example, in the assignment to `a`, the source call `negate(1,1)` – which passes in two `number` values, and hence, matches the first overload (conjunct) – is elaborated to the target call `(fst negate) 1 1`. In the assignment to `d`, the source call `negate(1,true)` – which passes in a `number` and a `boolean`, and hence matches the second overload – is elaborated to the target call `(snd negate) 1 true`, even though `1` does *not* have the refined type `ff`.

B: Cast. In the second step we check – using classical, unrefined type checking – that each clone adheres to its specified type. Unlike under usual intersection typing [89, 31], in our context these checks almost surely "fail". For example, `negate1` *does not* type check as the parameter `x` has type `number` and so we cannot compute `!x`. Similarly, `negate2` fails because `x` has type `boolean` and so `0-x` is erroneous. Rather than reject the program, we wrap such failures with dead-casts. For example, the above occurrences of `x` elaborate to `dead(x)` on the right in Figure 3.4.

Intuitively, the *value relationships* established at the call-sites and guards ensure that the failures will not happen at run-time. However, recall that the first phase's goal is to decouple reasoning about types from reasoning about values. Hence, we just *trust* all the types but use dead-casts to *explicate* the value-relationship obligations that are needed to establish typing: namely that the dead-casts are indeed dead code.

3.1.4 Phase 2: Verify

The second phase takes as input the elaborated program emitted by the first phase, which is essentially a classical *well-typed* ML program with assertions and without any value-overloading. Hence, the second phase can use any existing program logic [41, 14], refinement typing [113, 66, 91, 9], or contracts & abstract interpretation [75] to check that the target's assertions never fail, which, we prove, ensures that the source is type-safe.

To analyze programs with closures, collections and polymorphism, (*e.g.* `minIndex` from Figure 3.1) we perform the second phase using the refinement types that are carried over unchanged by the elaboration process of the first phase. Intuitively, refinement typing can be viewed as a generalization of classical program logics where *assertions* are generalized to type bindings, and the rule of *consequence* is generalized as subtyping. While refinement typing is a previously known technique, to make this work self-contained, we illustrate how the second phase verifies the dead-casts in Figure 3.4.

Refinement Type Checking. A refinement type checker works by building up an *environment* of type bindings that describe the machine state at each program point, and by checking that at each call-site, the actual argument's type is a refined *subtype* of the expected type for the callee, under the context described by the environment at that site. The subtyping relation for basic types is converted to a logical *verification condition* whose validity is checked by an SMT solver. The subtyping relation for *compound* types (*e.g.* functions, collections) is decomposed, via co- and contra-variant subtyping rules, into subtyping constraints over *basic* types, which can be discharged as above.

Typing dead-Casts. To use a standard refinement type checker for the second phase of verification, we only need to treat dead as a primitive operation with the refined type:

$$\mathsf{dead} :: \forall \alpha, \beta \,.\, (\{\, \nu \colon \alpha \mid \mathit{false} \,\}) \Rightarrow \beta$$

That is, we assign dead the *precondition false* which states there are *no* valid inputs for it, *i.e.* that it should never be called (akin to `assert(false)` in other settings).

Environments. To verify dead-casts, the refinement type checker builds up an environment of type binders describing *variables* and *branch conditions* that are in scope at each program point. For example, the dead call in `negate1`, has the environment

$$\Gamma_1 \doteq \texttt{flag}: \texttt{tt}, \texttt{x}: \texttt{number}, \texttt{g}_1 : \{\nu : \texttt{boolean} \mid \texttt{flag} = 0\} \tag{3.1}$$

where the first two bindings are the function parameters, whose types are the input types. The third binding is from the "else" branch of the `flag` test, asserting the branch condition `flag` is "falsy" *i.e.* equals 0. At the dead call in `negate2` the environment is:

$$\Gamma_2 \doteq \texttt{flag}: \texttt{ff}, \texttt{x}: \texttt{boolean}, \texttt{g}_1 : \{\nu : \texttt{boolean} \mid \texttt{flag} \neq 0\} \tag{3.2}$$

At the assignments to `a`, `b` and `c` the environments are respectively

$$\Gamma_a \doteq \texttt{negate}: \mathsf{T}_{\texttt{negate}} \tag{3.3}$$

$$\Gamma_b \doteq \Gamma_a, \alpha : \texttt{number} \tag{3.4}$$

$$\Gamma_c \doteq \Gamma_b, \beta : \texttt{boolean} \tag{3.5}$$

where $\mathsf{T}_{\texttt{negate}}$ abbreviates the *product* type of the (elaborated) tuple `negate`.

$$\mathsf{T}_{\texttt{negate}} \doteq ((\texttt{tt}, \texttt{number}) \Rightarrow \texttt{number}) \times ((\texttt{ff}, \texttt{boolean}) \Rightarrow \texttt{boolean}) \tag{3.6}$$

Subtyping. At each function call-site, the refinement type system checks that the *actual* argument is indeed a subtype of the *expected* one. For example, the dead calls inside `negate1` and `negate2` yield the respective subtyping obligation:

$$\Gamma_1 \vdash \{\nu : \texttt{number} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu : \texttt{number} \mid \mathit{false}\} \tag{3.7}$$

$$\Gamma_2 \vdash \{\nu : \texttt{boolean} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu : \texttt{boolean} \mid \mathit{false}\} \tag{3.8}$$

The obligation states that the type of the argument `x` should be a subtype of the input type of dead. Similarly, at the assignments to `a`, `b` and `c` the first arguments generate the respective subtyping obligations:

$$\Gamma_a \vdash \{\, \nu\colon \texttt{number} \mid \nu = 1 \,\} \quad \sqsubseteq \{\, \nu\colon \texttt{number} \mid \nu \neq 0 \,\} \tag{3.9}$$

$$\Gamma_b \vdash \{\, \nu\colon \texttt{number} \mid \nu = 0 \,\} \quad \sqsubseteq \{\, \nu\colon \texttt{number} \mid \nu = 0 \,\} \tag{3.10}$$

$$\Gamma_c \vdash \{\, \nu\colon \texttt{number} \mid \nu = 0 \,\} \quad \sqsubseteq \{\, \nu\colon \texttt{number} \mid \nu \neq 0 \,\} \tag{3.11}$$

Verification Conditions. To verify subtyping obligations, we convert them into logical verification conditions (VCs), whose validity determines whether the subtyping holds. A subtyping obligation

$$\Gamma \vdash \{\, \nu\colon b \mid p \,\} \sqsubseteq \{\, \nu\colon b \mid q \,\}$$

translates to the VC

$$[\![\Gamma]\!] \Rightarrow (p \Rightarrow q)$$

where $[\![\Gamma]\!]$ is the conjunction of the refinements of the binders in Γ. For example, the subtyping obligations (3.7) and (3.8) yield the respective VCs:

$$(\texttt{flag} \neq 0 \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{3.12}$$

$$(\texttt{flag} = 0 \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{3.13}$$

Here, the conjunct *true* arises from the trivial refinements *e.g.* the binding for `x`. The above VCs are deemed valid by an SMT solver as the hypotheses are inconsistent, which proves the call is indeed dead code. Similarly, (3.9) and (3.10) respectively yield VCs

$$\mathit{true} \Rightarrow \nu = 1 \Rightarrow \nu \neq 0 \tag{3.14}$$

$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu = 0 \tag{3.15}$$

which are deemed valid by SMT, verifying the assignments to α, β. However, by (3.11)

$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu \neq 0 \tag{3.16}$$

which is invalid, ensuring that we *reject* the call that assigns to `c`.

3.1.5 Two-Phase Inference

Our two-phased approach readily lends itself to abstract interpretation based *refinement inference* which can drastically lower the programmer annotations required to verify various safety properties, *e.g.* reducing the annotations needed to verify array bounds safety in ML programs from 31% of code size to under 1% [91]. Here we illustrate how inference works in the presence of value-based overloading. Suppose we are *not* given the refinements for the signature of `negate` but only the unrefined signature (either given to us explicitly as in TypeScript, inferred via dataflow analysis [50], or inferred via the techniques outlined in Chapter 2). As inference is difficult with incorrect code, we omit the erroneous statements that assign to `c` and `d`.

Refinement inference proceeds in three steps. First, we create *templates* which are the basic types decorated with *refinement variables* κ in place of the unknown refinements. Second, we perform the *trust* phase to elaborate the source program into a well-typed target free of overloading. Remember that this phase uses only the basic types and is oblivious to the (in this case unknown) refinements. Third, we perform the *verify* phase which now generates VCs over the refinement variables κ. These VCs – *logical implications* between the refinements and κ variables – correspond to so-called Horn constraints over the κ variables, and can be solved via abstract interpretation [39, 91].

0. Templates. Let us revisit the program from Figure 3.2, with the goal of inferring the refinements. Recall that the (unrefined) type of `negate` is:

$$\begin{aligned}&\texttt{negate} :: (\texttt{number}, \texttt{number}) \Rightarrow \texttt{number}\\ &\qquad \wedge\ (\texttt{number}, \texttt{boolean}) \Rightarrow \texttt{boolean}\end{aligned}$$

We create a *template* by refining each base type with a (distinct) refinement variable:

$$\begin{aligned}&\texttt{negate} :: (\{\,\nu\colon \texttt{number} \mid \kappa_1\,\}, \{\,\nu\colon \texttt{number} \mid \kappa_2\,\}) \Rightarrow \{\,\nu\colon \texttt{number} \mid \kappa_3\,\}\\ &\qquad \wedge\ (\{\,\nu\colon \texttt{number} \mid \kappa_4\,\}, \{\,\nu\colon \texttt{boolean} \mid \kappa_5\,\}) \Rightarrow \{\,\nu\colon \texttt{boolean} \mid \kappa_6\,\}\end{aligned}$$

1. Trust. The trust phase proceeds as before, propagating the refinements to the signatures of the elaborated target, yielding the code on the right in Figure 3.4 except that `negate1`

and `negate2` have the respective templates:

$$\texttt{negate1} :: (\{\,\nu\colon \texttt{number} \mid \kappa_1\,\}, \{\,\nu\colon \texttt{number} \mid \kappa_2\,\}) \Rightarrow \{\,\nu\colon \texttt{number} \mid \kappa_3\,\}$$

$$\texttt{negate2} :: (\{\,\nu\colon \texttt{number} \mid \kappa_4\,\}, \{\,\nu\colon \texttt{boolean} \mid \kappa_5\,\}) \Rightarrow \{\,\nu\colon \texttt{boolean} \mid \kappa_6\,\}$$

2. Verify. The verify phase proceeds as before, but using templates instead of the types. Hence, at the dead-cast in `negate1` and `negate2`, and the calls to `negate` that assign to `a` and `b`, instead of the VCs (3.12), (3.13), (3.14) and (3.15), we get the respective Horn constraints:

$$(\kappa_1\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow\ \nu = \texttt{x} \ \Rightarrow\ \mathit{false} \tag{3.17}$$

$$(\kappa_4\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow\ \nu = \texttt{x} \ \Rightarrow\ \mathit{false} \tag{3.18}$$

$$\mathit{true} \Rightarrow\ \nu = 1 \ \Rightarrow\ \kappa_1 \tag{3.19}$$

$$\mathit{true} \Rightarrow\ \nu = 0 \ \Rightarrow\ \kappa_4 \tag{3.20}$$

These constraints are identical to the corresponding VCs except that κ variables appear in place of the unknown refinements for the corresponding binders. We can solve these constraints using fixpoint computations over a variety of abstract domains such as monomial predicate abstraction [39, 91] over a set of ground predicates which are arithmetic (in)equalities between program variables and constants, to obtain a solution mapping each κ to a concrete refinement:

$$\kappa_1 \ \doteq\ \nu = 0 \qquad \kappa_4 \ \doteq\ \nu \neq 0 \qquad \kappa_2, \kappa_3, \kappa_5, \kappa_6 \ \doteq\ \mathit{true}$$

which, when plugged back into the templates, allow us to infer types for `negate`.

Higher-Order Verification. Our two-phased approach generalizes directly to offer precise analysis for *polymorphic, higher-order* functions. Returning to the code in Figure 3.1, our two-phased inference algorithm infers the refinement types

$$\begin{aligned}
\texttt{_reduce} \ &:: \ \forall \alpha, \beta\,.\,(\texttt{a}\colon \alpha\texttt{[]}, \texttt{f}\colon (\beta, \texttt{idx}\langle\langle \texttt{a} \rangle\rangle) \Rightarrow \beta, \texttt{x}\colon \beta) \Rightarrow \beta \\
\texttt{reduce} \ &:: \ \forall \alpha\,.\,(\texttt{a}\colon \alpha\texttt{[]}^{+}, \texttt{f}\colon (\alpha, \alpha, \texttt{idx}\langle\langle \texttt{a} \rangle\rangle) \Rightarrow \alpha) \Rightarrow \alpha \\
&\wedge \ \forall \alpha, \beta\,.\,(\texttt{a}\colon \alpha\texttt{[]}, \texttt{f}\colon (\beta, \alpha, \texttt{idx}\langle\langle \texttt{a} \rangle\rangle) \Rightarrow \beta, \texttt{x}\colon \beta) \Rightarrow \beta
\end{aligned}$$

where $\texttt{idx}\langle\langle \texttt{a} \rangle\rangle$ describes *valid indices* for array a, and $\alpha\texttt{[]}^+$ describes non-empty arrays:

$$\begin{aligned} \texttt{idx}\langle\langle \texttt{a} \rangle\rangle &\doteq \{ \nu\texttt{:}\ \texttt{number} \mid 0 \leq \nu < \texttt{len(a)} \} \\ \alpha\texttt{[]}^+ &\doteq \{ \nu\texttt{:}\ \alpha\texttt{[]} \mid 0 < \texttt{len}(\nu) \} \end{aligned}$$

The above type is a precise *summary* for the higher-order behavior of `_reduce`: it describes the relationship between the input array a, the step ("callback") function f, and the initial value of the accumulator, and stipulates that the output satisfies the same *properties* β as the input x. Furthermore, it captures the fact that the callback f is only invoked on inputs that are valid indices for the array a that is being reduced. Consequently, Liquid Types [91], for example, would automatically infer

$$\begin{aligned} \texttt{step} &\doteq \forall\alpha.(\texttt{idx}\langle\langle \texttt{a} \rangle\rangle, \alpha, \texttt{idx}\langle\langle \texttt{a} \rangle\rangle) \Rightarrow \texttt{idx}\langle\langle \texttt{a} \rangle\rangle \\ \texttt{minIndex} &\doteq \forall\alpha.(\alpha\texttt{[]}) \Rightarrow \texttt{number} \end{aligned}$$

thereby verifying the safety of array accesses in the presence of higher order functions, collections, and value-based overloading.

3.2 Syntax and Operational Semantics of TBV

Next, we formalize two-phase typing via a core calculus TBV comprising a *source* language λ_{src} *with* overloading via union and intersection types, and a simply typed *target* language λ_{tgt} *without* overloading, where the assumptions for safe overloading are explicated via dead-casts. In Section 3.3, we describe the first phase that elaborates source programs into target programs, and finally, in Section 3.4 we describe how the second phase verifies the dead-casts on the target to establish the safety of the source. Our elaboration follows the overall compilation strategy of Dunfield [31] except that we have value-based overloading instead of an explicit "merge" operator [89], and consequently, our elaboration and proofs must account for source level "errors" via dead-casts.

3.2.1 Source Language (λ_{src})

Terms. We define a source language λ_{src}, with syntax shown in Figure 3.5. Expressions include variables, functions, applications, a ternary conditional construct, and primitive constants

Syntax of λ_{src}

v	$::=$		**Values**
	$\mid$	n	constant
	$\mid$	$(x) \Rightarrow e$	arrow function
e	$::=$		**Expressions**
	$\mid$	v	value
	$\mid$	x	variable
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	conditional
	$\mid$	$e_1(e_2)$	call
τ	$::=$		**Types**
	$\mid$	b	primitive type
	$\mid$	$\tau_1 \to \tau_2$	arrow type
	$\mid$	$\tau_1 \wedge \tau_2$	intersection
	$\mid$	$\tau_1 \vee \tau_2$	union

Reduction Rules for λ_{src} $\boxed{e \longrightarrow e'}$

$$\frac{e \longrightarrow e'}{E\langle e\rangle \longrightarrow E\langle e'\rangle}\ [\text{E-ECTX}] \qquad \frac{n \equiv \texttt{true} \implies e \equiv e_1 \quad n \equiv \texttt{false} \implies e \equiv e_2}{\texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e}\ [\text{E-COND}]$$

$$\frac{}{n(v) \longrightarrow [\![n]\!](v)}\ [\text{E-APP-1}] \qquad \frac{}{((x) \Rightarrow e)(v) \longrightarrow [v/x]\,(e)}\ [\text{E-APP-2}]$$

Figure 3.5. Language λ_{src}: Syntax and Operational Semantics

n which include numbers $0, 1, \ldots$, operators $+, -, \ldots$, *etc.*

Operational Semantics. In Figure 3.5 we also define a standard small-step operational semantics for λ_{src} with a left-to-right order of evaluation, based on evaluation contexts

$$E \ ::= \ \langle\rangle \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid E(e) \mid v(E)$$

Types. Figure 3.5 shows the types τ in the source language. These include primitive types b, arrow types $\tau_1 \to \tau_2$ and, most notably, intersections $\tau_1 \wedge \tau_2$ and (untagged) unions $\tau_1 \vee \tau_2$. Note that the source level types are *not* refined, as crucially, the first phase *ignores* the refinements when carrying out the elaboration.

Well-Formed Types $\boxed{\vdash \tau}$

$$\vdash b \qquad \frac{\vdash \tau_1 \qquad \vdash \tau_2}{\vdash \tau_1 \to \tau_2} \qquad \frac{\vdash \tau_1 \qquad \vdash \tau_2 \qquad \mathsf{tag}(\tau_1) = \mathsf{tag}(\tau_2)}{\vdash \tau_1 \wedge \tau_2}$$

$$\frac{\vdash \tau_1 \qquad \vdash \tau_2 \qquad \mathsf{tag}(\tau_1) \cap \mathsf{tag}(\tau_2) = \varnothing}{\vdash \tau_1 \vee \tau_2}$$

$$\begin{array}{rcl} \mathsf{tag}(\mathsf{Num}) & = & \{\texttt{"number"}\} \\ \mathsf{tag}(\mathsf{Bool}) & = & \{\texttt{"boolean"}\} \\ \mathsf{tag}(\tau \to \tau') & = & \{\texttt{"function"}\} \end{array} \qquad \begin{array}{rcl} \mathsf{tag}(\tau \wedge \tau') & = & \mathsf{tag}(\tau) \\ \mathsf{tag}(\tau \vee \tau') & = & \mathsf{tag}(\tau) \cup \mathsf{tag}(\tau') \end{array}$$

Figure 3.6. Basic Type Well-Formedness for λ_{src}

Tags. As is common in dynamically typed languages, runtime values are associated with *type tags*, which can be inspected with a type test (cf. JavaScript's `typeof` operator). We model this notion by associating each type with a set of possible tags. The multiplicity arises from unions. The meta-function $\mathsf{tag}(\tau)$, defined in Figure 3.6, returns the possible tags that values of type τ may have at runtime.

Well-Formedness. In order to resolve overloads statically, we apply certain restrictions on the form of union and intersection types, shown by the judgment $\vdash \tau$ formalized in Figure 3.6. For convenience of exposition, the parts of an untagged union need to have distinct runtime tags, and intersection types require all conjuncts to have the same tag.

3.2.2 Target Language (λ_{tgt})

The target language λ_{tgt} eliminates (value-based) overloading and thereby provides a basic, well-typed skeleton that can be further refined with logical predicates. Towards this end, unions and intersections are replaced with classical *tagged unions*, *products* and dead-casts, that encode the requirements for basic typing.

Terms. Figure 3.7 shows the terms w of λ_{tgt}, which extend the source language with the introduction of pairs, projections, injections, a case-splitting construct and a special constant term $\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(w)$ which denotes an erroneous computation. Intuitively, a $\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(w)$ is produced in the elaboration phase whenever the actual type τ_1 for a term w is incompatible with an expected type τ_2.

Operational Semantics. As in the source language we define evaluation contexts for

Syntax of λ_{tgt}

w	::=	$\ldots$	**Expressions**
	$\mid$	(w_1, w_2)	pair
	$\mid$	$\texttt{proj}_1 w \mid \texttt{proj}_2 w$	projection
	$\mid$	$\texttt{inj}_1\ w \mid \texttt{inj}_2\ w$	injection
	$\mid$	$\texttt{case}\ w\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2$	case
	$\mid$	$\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(w)$	deadcast
v	::=	$\ldots$	**Values**
	$\mid$	$\texttt{inj}_1\ v \mid \texttt{inj}_2\ v$	value injection
	$\mid$	(w_1, w_2)	value pair
	$\mid$	$\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(v)$	deadcast
T	::=		**Refinement Types**
	$\mid$	$\{v\colon b \mid P\}$	base refinement type
	$\mid$	$x\colon \mathsf{T}_1 \to \mathsf{T}_2$	arrow type
	$\mid$	$\mathsf{T}_1 + \mathsf{T}_2$	sum type
	$\mid$	$\mathsf{T}_1 \times \mathsf{T}_2$	product type

Reduction Rules for λ_{tgt} $\boxed{w \longrightarrow w'}$

$$\frac{w \longrightarrow w'}{\mathcal{E}\langle w\rangle \longrightarrow \mathcal{E}\langle w'\rangle}\ [\text{TE-ECTX}] \qquad \frac{k \in \{1, 2\}}{\texttt{proj}_k(w_1, w_2) \longrightarrow w_k}\ [\text{TE-PROJ}] \qquad \frac{v \not\equiv \mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(v')}{n(v) \longrightarrow [\![n]\!](v)}\ [\text{TE-APP-1}]$$

$$\frac{}{((x) \Rightarrow w)(v) \longrightarrow [v/x](w)}\ [\text{TE-APP-2}]$$

$$\frac{k \in \{1, 2\}}{\texttt{case}\ \texttt{inj}_k\ v\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 \longrightarrow [v/x_k](w_k)}\ [\text{TE-CASE}]$$

Figure 3.7. Language λ_{tgt}: Syntax and Operational Semantics

λ_{tgt}:

$$\begin{array}{rcl} \mathcal{E} & ::= & \langle\rangle \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ w_1\ \texttt{else}\ w_2 \mid \mathcal{E}(w) \mid v(\mathcal{E}) \mid \texttt{inj}_k\ \mathcal{E} \\ & \mid & \texttt{proj}_k\mathcal{E} \mid \mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(\mathcal{E}) \mid \texttt{case}\ \mathcal{E}\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 \end{array}$$

and use them to define a small-step operational semantics for the target in Figure 3.7. Note how evaluation is allowed in dead-casts and $\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(v)$ *is* a value.

Types. The target language is checked against a refinement type checker. Thus, we modify the type language to account for the new language terms and refinements. *Basic Refinement Types* are of the form $\{ \nu \colon b \mid P \}$, consisting of the same basic types b as source types, and a logical predicate P (over some decidable logic), which describes the properties that values of the type must satisfy. Here, ν is a special *value variable* that describes the inhabitants of the type, that does not appear in the program, but can appear inside the refinement P. Function types are of the form $x \colon T_1 \to T_2$, to express the fact that the refinement predicate of the return type T_2 may refer to the value of the argument x. Sum and product types have the usual structure found in ML-like languages.

3.3 Phase 1: Trust

Terms of λ_{src} are elaborated to terms of λ_{tgt} by a judgment:

$$\Gamma \vdash e : \tau \hookrightarrow w$$

This is read: under the typing assumptions in Γ, term e of the source language is assigned a type τ and elaborates to a term w of the target language. This judgment follows closely Dunfield's elaboration judgment [31], but with crucial differences that arise due to dynamic, value-based overloading, which we outline below.

Elaboration Ignores Refinements. A key aspect of the first phase is that elaboration is based solely on the basic types, *i.e.* does *not* take type refinements into account. Hence, the types assigned to source terms are transparent with respect to refinements; or more precisely, they work just as placeholders for refinements that can be provided as user specifications. These specifications are propagated *as is* during the first phase along with the respective basic types they are attached to. Due to this transparency of refinements we have decided to omit them entirely from our description of the elaboration phase.

3.3.1 Source Language Type checking and Elaboration

Figure 3.8 shows the rules that formalize the elaboration process. In this formulation we follow a bidirectional approach [81] to make our rules more algorithmic and restrict the context under which dead-casts can occur. At a high-level, following Dunfield [31], unions

Elaboration Typing Rules $\boxed{\Gamma \vdash e \uparrow \tau \hookrightarrow w}$ $\boxed{\Gamma \vdash e \downarrow \tau \hookrightarrow w}$

$$\frac{\vdash \tau \qquad \Gamma \vdash e \downarrow \tau \hookrightarrow w}{\Gamma \vdash (e : \tau) \uparrow \tau \hookrightarrow (w : \tau)}\ [\text{T-Annot}] \qquad \frac{\Gamma \vdash e \uparrow \tau \hookrightarrow w}{\Gamma \vdash e \downarrow \tau \hookrightarrow w}\ [\text{T-Weaken}]$$

$$\frac{}{\Gamma \vdash n \uparrow b \hookrightarrow n}\ [\text{T-Cst}] \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \uparrow \tau \hookrightarrow x}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash e \downarrow \texttt{boolean} \hookrightarrow w \qquad \Gamma \vdash e_1 \downarrow \tau \hookrightarrow w_1 \qquad \Gamma \vdash e_2 \downarrow \tau \hookrightarrow w_2}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \downarrow \tau \hookrightarrow \texttt{if } w \texttt{ then } w_1 \texttt{ else } w_2}\ [\text{T-If}]$$

$$\frac{\Gamma \vdash v \downarrow \tau_1 \hookrightarrow w_1 \qquad \Gamma \vdash v \downarrow \tau_2 \hookrightarrow w_2 \qquad \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash v \downarrow \tau_1 \wedge \tau_2 \hookrightarrow (w_1, w_2)}\ [\text{T-}\wedge\text{I}]$$

$$\frac{\Gamma \vdash e \uparrow \tau_1 \wedge \tau_2 \hookrightarrow w \qquad k \in \{1, 2\}}{\Gamma \vdash e \uparrow \tau_k \hookrightarrow \texttt{proj}_k w}\ [\text{T-}\wedge\text{E}] \qquad \frac{\Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \hookrightarrow w}{\Gamma \vdash (x) \Rightarrow e \downarrow \tau_1 \to \tau_2 \hookrightarrow (x) \Rightarrow w}\ [\text{T-Arrow}]$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \hookrightarrow w_1 \qquad \Gamma \vdash e_2 \downarrow \tau \hookrightarrow w_2}{\Gamma \vdash e_1(e_2) \uparrow \tau_2 \hookrightarrow w_1(w_2)}\ [\text{T-App}]$$

$$\frac{\Gamma \vdash e \uparrow \tau' \hookrightarrow w \qquad \mathrm{tag}(\tau) \cap \mathrm{tag}(\tau') = \varnothing}{\Gamma \vdash e \downarrow \tau \hookrightarrow \mathsf{dead}\downarrow_{\tau}^{\tau'}(w)}\ [\text{T-}\bot]$$

$$\frac{\Gamma \vdash e \downarrow \tau_k \hookrightarrow w \qquad \vdash \tau_1 \vee \tau_2}{\Gamma \vdash e \downarrow \tau_1 \vee \tau_2 \hookrightarrow \texttt{inj}_k\ w}\ [\text{T-}\vee\text{I}]$$

$$\frac{\begin{array}{c}\Gamma, x_1 : \tau_1 \vdash E\langle x_1 \rangle \downarrow \tau \hookrightarrow w_1 \\ \Gamma \vdash e_0 \uparrow \tau_1 \vee \tau_2 \hookrightarrow w_0 \qquad \Gamma, x_2 : \tau_2 \vdash E\langle x_2 \rangle \downarrow \tau \hookrightarrow w_2\end{array}}{\Gamma \vdash E\langle e_0 \rangle \downarrow \tau \hookrightarrow \texttt{case } w_0 \texttt{ of inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2}\ [\text{T-}\vee\text{E}]$$

Figure 3.8. Elaboration Typing rules

and intersections are translated to simpler typing constructs like sums and products (and the attendant injections, pattern-matches, and projections). Unlike the above work, which focuses on the classical intersection setting where overloading is explicit via a "merge" construct [89], we are concerned with the dynamic setting where overloading is value-based, leading to conventional type "errors".

Elaboration Modes: Checking and Inferring. One of the distinguishing features of our type system is its ability to not fail in cases where conventional static type system would raise type incompatibility errors, but instead elaborate the offending terms to the special error form $\mathsf{dead}\downarrow_{\tau_2}^{\tau_1}(w)$. These error forms do not appear indiscriminately, but only under checking rules ($\downarrow$).

However, the formulation of Figure 3.8 is too flexible in handling calls to overloaded functions. Consider, call $e_1\,(e_2)$ where e_1 has an overloaded signature. The checking algorithm would have to examine all possible conjuncts of the overload and check e_2 under each one. The problem arises since even for incompatible signatures Rule T-$\bot$ can be applied and allow typechecking to proceed.

Conceptually this approach is still sound. Erroneous programs will be caught in the second phase, when the dead-casts fail to be discharged. However, to keep the algorithm more tractable and predictable, we restrict this behavior, by tracking the use of intersection types. We revise Rule T-$\wedge$E as follows:

$$\frac{\Gamma \vdash e \uparrow \tau_1 \wedge \tau_2 \hookrightarrow w \qquad k \in \{1, 2\}}{\Gamma \vdash e \uparrow_? \tau_k \hookrightarrow \mathtt{proj}_k\, w}\ [\text{T-}\wedge\text{E-?}]$$

This speculative version (note the subscript "?" of $\uparrow$) of the judgment denotes that the current type was obtained by choice among parts of an intersection. We also introduce a revised version of the T-App Rule that handles speculative results:

$$\frac{\Gamma \vdash e_1 \uparrow_? \tau_1 \to \tau_2 \hookrightarrow w_1 \qquad \Gamma \vdash e_2 \uparrow \tau \hookrightarrow w_2}{\Gamma \vdash e_1\,(e_2) \uparrow \tau_2 \hookrightarrow w_1\,(w_2)}\ [\text{T-App-?}]$$

This change requires the type for e_2 to be inferred, instead of checked, crucially disabling Rule T-$\bot$. In fact, the only rules applicable here require e_2 to either be a constant, a variable or an

annotated expression. The rest of the rules preserve the speculative mode transparently.

Standard Rules. Rules T-CST, T-VAR are standard and preserve the structure of the source program. Rule T-IF expects the condition e of a conditional expression to be of boolean type, and checks each branch of the conditional under the same type τ.

Intersections. In Rule T-$\wedge$I the choice of the type we assign to a value v causes different elaborated terms v_k, as different typing requirements cause the addition of dead-casts at different places. This rule is intended to be used primarily for abstractions, so it's limited to accept values as input. Rule T-$\wedge$E for eliminating intersections replaces a term e that is originally typed as an intersection with a projection of that part of the pair that has a matching type. By T-$\wedge$I values typed at an intersection get a pair form.

Unions. Rule T-$\vee$I for union introduction is standard. The union elimination rule, taken from Dunfield's elaboration scheme [31], is more involved. It first assumes an expression e_0, for which we can infer a union type $\tau_1 \vee \tau_2$. Then requires finding an evaluation context E, such that when filled in with a variable x typed at either τ_1 or τ_2, the resulting expression context can be checked under type τ. If such E exists then $E\langle e_0 \rangle$ can be checked under type τ. While the rule is inherently non-deterministic, it suffices for our purposes of describing the elaboration process; see Dunfield's subsequent work on untangling type checking of intersections and unions [30] for an algorithmic variant via a let-normal conversion.

Abstraction and Application. Rule T-ARROW assumes the arrow type $\tau_1 \to \tau_2$ is given as annotation and is required to conform to the well-formedness constraints. At the crux of our type system is the Rule T-APP. We saw earlier how we disable dead-cast insertions when the function is an overloaded one. Below, we justify this choice using an example. If on the other hand, the type for e_1 is assigned without choosing among the parts of an intersection, then expression e_2 can be typed in checked mode, potentially producing dead-casts.

Trusting via dead-Casts. The cornerstone of the "trust" phase lies in the presence of the T-$\bot$ rule. As we mentioned earlier, this rule can only be used in checking mode. The main idea here is to allow cases that are obviously wrong, as far as the simple first phase type system is concerned; but, at the same time, include a dead-cast annotation and defer sound type checking for the second phase. The premises of this rule specify that a dead-cast annotation will only be used if the inferred and the expected type have different tags. One of the consequences of this decision is that it does not allow dead-casts induced by a mismatch between higher-order types,

as the tags for both types would be the same (most likely `"function"`). Thus, such mismatches are ill-typed and rejected in the first phase. This limitation is due to the limited information that can be encoded using the tag mechanism. A more expressive tag mechanism could eliminate this restriction but we omit this for simplicity of exposition.

Semantics of dead**-Casts.** To prove that elaboration preserves source level behaviors, our design of dead-casts preserves the property that the target gets stuck *iff* the source gets stuck. That is, source level type "errors" *do not lead to early* failures (*e.g.* at function call boundaries). Instead, dead-casts correspond to *markers* for all source terms that can potentially cause execution to get stuck. Hence, the target execution itself gets stuck at the same places as the source – *i.e.* when applying to a non-function, branching on a non-boolean or primitive application over the wrong base value, except that in the target, the stuckness can only occur when the value in question carries a dead marker. Consider the source program $(\texttt{(x)} \Rightarrow \texttt{x}\ \texttt{1})\ \texttt{0}$ which gets stuck *after* the top-level application, when applying 1 to 0. It could be elaborated to $(\texttt{(x)} \Rightarrow \texttt{x}\ \texttt{1})\ (\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}\ (\texttt{0}))$ (where τ_1 and τ_2 are respectively `number` and `number` $\rightarrow$ `number`) which also has a top-level application and gets stuck at the second, inner application.

Necessity of speculative mode. If we allowed the argument of an *overloaded* call-site to be typed in *checking* mode, then for the application `f(x)`, where `f` has been assigned the type $\mathsf{f}\colon \mathsf{I} \rightarrow \mathsf{I} \wedge \mathsf{B} \rightarrow \mathsf{B}$ and $\mathsf{x}\colon \mathsf{B}$ (where I and B stand for `number` and `boolean` respectively), the following derivation would be possible:

$$\dfrac{\dfrac{\vdots}{\cdots \vdash \mathsf{f} \uparrow \mathsf{I} \rightarrow \mathsf{I} \hookrightarrow \mathtt{proj}_1\,\mathsf{f}}\ [\text{T-}\wedge\text{E}] \qquad \dfrac{\cdots \vdash \mathsf{x} \uparrow \mathsf{B} \hookrightarrow \mathsf{x} \qquad \mathtt{tag}(\mathsf{B}) \cap \mathtt{tag}(\mathsf{I}) = \varnothing}{\cdots \vdash \mathsf{x} \downarrow \mathsf{I} \hookrightarrow \mathsf{dead}\downarrow^{\mathsf{B}}_{\mathsf{I}}\ (\mathsf{x})}\ [\text{T-}\bot]}{\mathsf{f}\colon \mathsf{I} \rightarrow \mathsf{I} \wedge \mathsf{B} \rightarrow \mathsf{B}, \mathsf{x}\colon \mathsf{B} \vdash \mathsf{f}(\mathsf{x}) \uparrow \mathsf{I} \hookrightarrow (\mathtt{proj}_1\,\mathsf{f})(\mathsf{dead}\downarrow^{\mathsf{B}}_{\mathsf{I}}\ (\mathsf{x}))}\ [\text{T-App}]$$

But, clearly, the intended derivation here is:

$$\dfrac{\dfrac{\vdots}{\cdots \vdash \mathsf{f} \uparrow \mathsf{B} \rightarrow \mathsf{B} \hookrightarrow \mathtt{proj}_2\,\mathsf{f}}\ [\text{T-}\wedge\text{E}] \qquad \cdots \vdash \mathsf{x} \uparrow \mathsf{B} \hookrightarrow \mathsf{x}}{\mathsf{f}\colon \mathsf{I} \rightarrow \mathsf{I} \wedge \mathsf{B} \rightarrow \mathsf{B}, \mathsf{x}\colon \mathsf{B} \vdash \mathsf{f}(\mathsf{x}) \uparrow \mathsf{B} \hookrightarrow (\mathtt{proj}_2\,\mathsf{f})(\mathsf{x})}\ [\text{T-App}]$$

Subtyping. This formulation has been kept simple with respect to subtyping. The only

notion of subtyping appears in the T-$\vee$I rule, where a type τ_1 is widened to $\tau_1 \vee \tau_2$. We could have employed a more elaborate notion of subtyping, by introducing a subtyping relation ($\leq$) and a subsumption rule for our typing elaboration. The rules for this subtyping relation would include, among others, function subtyping:

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

However, supporting subtyping in higher-order constructs would only be possible with the introduction of wrappers around functions to accommodate checks on the arguments and results of functions. So, assuming that a cast c represents a dynamic check the above rule would correspond to a cast producing relation ($\triangleright$):

$$\frac{\tau_1' \triangleright \tau_1 \rightsquigarrow c_1 \qquad \tau_2 \triangleright \tau_2' \rightsquigarrow c_2}{\tau_1 \to \tau_2 \triangleright \tau_1' \to \tau_2' \rightsquigarrow \lambda f.\lambda x.(c_2\ (f\ (c_1\ x)))}$$

This formulation would just complicate the translation without giving any more insight in the main idea of our technique, and hence we forgo it.

3.3.2 Source and Target Language Consistency

In this section, we present the theorems that precisely connect the semantics of source programs with their elaborated targets. The main challenges towards establishing those are that: (1) the source and target do not proceed in lock-step, a single step of the one may be matched by several steps of the other (for example evaluating a projection in the target language does not correspond to any step in the source language), and (2) we must design the semantics of the dead-casts in the target to ensure that dead-casts cause evaluation to get stuck iff some primitive operation in the source gets stuck. We address these, next, with a number of lemmas and state our assumptions.

Value Monotonicity. This lemma fills in the mismatch that emerges when (non-value) expressions in the source language elaborate to values in the target language. Informally, if a source expression e elaborates to a target value v, then e evaluates (after potentially multiple steps) to a value v that is related to the target value v with an elaboration relation under the same

type. Furthermore, all expressions on the path to the target value v elaborate to the same value and get assigned the same type.

Lemma 3.1 (Value Monotonicity)**.** *If* $\Gamma \vdash e : \tau \hookrightarrow v$*, then there exists* v *s.t.:*

(1) $e \longrightarrow^* v$

(2) $\Gamma \vdash v : \tau \hookrightarrow v$

(3) $\forall i\,(e \longrightarrow^* e_i)\,.\,\Gamma \vdash e_i : \tau \hookrightarrow v$

Proof. The first two parts are handled similarly to Dunfield [31, Lemma 11]. The last part is proved by induction on the length of the path $e \longrightarrow^* e_i$. Details of this proof can be found in the appendix (Lemma B.8). □

The reverse of the above lemma also comes in handy. Namely, given a value v that elaborates to an expression w and gets assigned the type τ, there exists a value in the target language v, such that v elaborates to v and get assigned the *same* type τ.

Lemma 3.2 (Reverse Value Monotonicity)**.** *If* $\Gamma \vdash v : \tau \hookrightarrow w$*, then there exists* v *s.t.:* $w \longrightarrow^* v$ *and* $\Gamma \vdash v : \tau \hookrightarrow v$*.*

Proof. Similar to proof of Lemma 3.1. □

This is an interesting result as it establishes that different derivations may assign the same type to a term and still elaborate it to different target terms. For example, one can assume derivations that consecutively apply the intersection introduction and elimination rules. It's easy to see that the same value v can be used in the following elaborations:

$$\cdot \vdash v : \tau_1 \wedge \tau_2 \hookrightarrow (v_1, v_2)$$

$$\cdot \vdash v : \tau_1 \wedge \tau_2 \hookrightarrow \underbrace{(\mathtt{proj}_1(v_1, v_2), \mathtt{proj}_2(v_1, v_2))}_{w}$$

Lemma 3.2 guarantees it will always be the case that $w \longrightarrow^* (v_1, v_2)$. It is up to the implementation of the type checking algorithm to produce an efficient target term.

Primitive Semantics. To connect the failure of the dead-casts with source programs getting stuck, we assume that the primitive constants are well defined for all the values of their input domain *but not* for dead-cast values. This lets us establish that primitive operations π

are invariant to elaboration. Hence, a source primitive application gets stuck iff the elaborated argument is a dead-cast. The forward version of this statement is the following assumption.

Assumption 3.3.1 (Primitive constant application)**.** *If*

(i) $\cdot \vdash n : \tau_1 \to \tau_2 \hookrightarrow n$

(ii) $\cdot \vdash v : \tau_1 \hookrightarrow v$

(iii) $v \not\equiv \mathsf{dead}\downarrow^{\cdot}_{\tau_1} (\cdot)$

then

(a) $n(v) \longrightarrow [\![n]\!](v)$

(b) $n(v) \longrightarrow [\![n]\!](v)$

(c) $\cdot \vdash [\![n]\!](v) : \tau_2 \hookrightarrow [\![n]\!](v)$

Substitution lemma. The proof of soundness relies upon the following substitution lemma.

Lemma 3.3 (Substitution)**.** *If* $\Gamma, x\colon \tau \vdash e : \tau' \hookrightarrow w$ *and* $\Gamma \vdash v : \tau \hookrightarrow v$ *then* $\Gamma \vdash [v/x](e) : \tau' \hookrightarrow [v/x](w)$.

Proof. Similar to the substitution proof of Dunfield [31, Lemma 12]. □

We use the above lemmas and assumptions to obtain a consistency result, analogous to Dunfield's Consistency Theorem [31], which states that the elaboration produces terms that are *consistent* with the source in that each step of the target is matched by a corresponding step of the source, *i.e.* the behaviors of the target *under-approximate* the behaviors of the source.

Theorem 3.4 (Consistency)**.** *If* $\cdot \vdash e : \tau \hookrightarrow w$ *and* $w \longrightarrow w'$ *then there exists* e' *such that* $e \longrightarrow^* e'$ *and* $\cdot \vdash e' : \tau \hookrightarrow w'$.

Proof. The proof of this theorem is by induction on the derivation $\cdot \vdash e : \tau \hookrightarrow w$, adapting the proof scheme given by Dunfield [31], and using Lemma 3.1. Details of this proof can be found in the appendix (Theorem B.12). □

While this suffices to prove *soundness* – intuitively if the target does not "go wrong" then the source cannot "go wrong" either – it is not wholly satisfactory as a trivial translation that converts every source program to an ill-typed target also satisfies the above requirement. So, unlike Dunfield [31], we also establish a completeness result stating that if the source term steps, then the elaborated program will also eventually step to a corresponding (by elaboration) term. Theorem 3.5 declares that behaviors of the elaborated target *over-approximate* those of the source, and hence, in conjunction with Theorem 3.4, ensure that the source "goes wrong" iff the target does.

Theorem 3.5 (Reverse Consistency)**.** *If* $\cdot \vdash e : \tau \hookrightarrow w$ *and* $e \longrightarrow e'$ *then there exists* w' *such that* $\cdot \vdash e' : \tau \hookrightarrow w'$*, and* $w \longrightarrow^{+} w'$*.*

Proof. Similar to the proof of Theorem 3.4, using adapted versions of the lemmas used by Dunfield [31] and Lemma 3.2. Again, details can be found in the appendix (Theorem B.13). □

3.4 Phase 2: Verify

At the end of the first phase, we have elaborated the source with value based overloading into a classically well-typed target with conventional typing features and dead-casts which are really assertions that explicate the *trust assumptions* made to type the source. Thanks to Theorems 3.4 and 3.5 we know the semantics of the target are equivalent to the source. Thus, to verify the source, all that remains is to prove that the target will not "go wrong", that is to prove that the dead-casts are indeed never executed at run-time.

One advantage of our elaboration scheme is that at this point *any* program analysis for ML-like languages (*i.e.* supporting products, sums, and first class functions) can be applied to discharge the dead-cast [26]: as long as the target is safe, the consistency theorems guarantee that the source is safe. In our case, we choose to instantiate the second phase with *refinement types* as they: (1) are especially well suited to handle higher-order polymorphic functions, like `minIndex` from Figure 3.1, (2) can easily express other correctness requirements, *e.g.* array bounds safety, thereby allowing us to establish not just type safety but richer correctness properties, and, (3) are automatically inferred via the abstract interpretation framework of Liquid Typing [91]. Next, we recall how refinement typing works to show how dead-cast checking can be carried out, and then present the end-to-end soundness guarantees established by composing the two phases.

Refined Typechecking Rules $\boxed{G \vdash w :: T}$

$$\frac{G \vdash w :: T_1 \qquad G \vdash T_1 \sqsubseteq T_2}{G \vdash w :: T_2}\ \text{[R-Sub]} \qquad \frac{}{G \vdash n :: b}\ \text{[R-Cst]} \qquad \frac{x\colon T \in G}{G \vdash x :: \mathsf{sngl}\,(T, x)}\ \text{[R-Var]}$$

$$\frac{G \vdash w :: \mathsf{Bool} \qquad G;\, w \vdash w_1 :: T \qquad G;\, \neg w \vdash w_2 :: T}{G \vdash \texttt{if}\ w\ \texttt{then}\ w_1\ \texttt{else}\ w_2 :: T}\ \text{[R-If]} \qquad \frac{G,\, x\colon T_x \vdash w :: T}{G \vdash (x) \Rightarrow w :: T_x \to T}\ \text{[R-Lam]}$$

$$\frac{G \vdash w_1 :: T_x \to T \qquad G \vdash w_2 :: T_x}{G \vdash w_1\,(w_2) :: [w_2/x]\,(T)}\ \text{[R-App]} \qquad \frac{\forall k \in \{1, 2\}\,.\, G \vdash w_k :: T_k}{G \vdash (w_1, w_2) :: T_1 \times T_2}\ \text{[R-Pair]}$$

$$\frac{G \vdash w :: T_1 \times T_2}{G \vdash \texttt{proj}_k w :: T_k}\ \text{[R-Proj]} \qquad \frac{G \vdash w :: T_k}{G \vdash \texttt{inj}_k\ w :: T_1 + T_2}\ \text{[R-Inj]}$$

$$\frac{G \vdash w :: T_1 + T_2 \qquad G,\, x_1\colon T_1 \vdash w_1 :: T \qquad G,\, x_2\colon T_2 \vdash w_2 :: T}{G \vdash \texttt{case}\ w\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 :: T}\ \text{[R-Case]}$$

Refinement Subtyping $\boxed{G \vdash T_1 \sqsubseteq T_2}$

$$\frac{\mathsf{Valid}(\llbracket G \rrbracket \wedge \llbracket P \rrbracket \Rightarrow \llbracket P' \rrbracket)}{G \vdash \{v\colon b \mid P\} \sqsubseteq \{v\colon b \mid P'\}}\ \text{[⊑-Base]} \qquad \frac{G \vdash T'_x \sqsubseteq T_x \qquad G,\, x\colon T'_x \vdash T \sqsubseteq T'}{G \vdash (x\colon T_x) \Rightarrow T \sqsubseteq (x\colon T'_x) \Rightarrow T'}\ \text{[⊑-Fun]}$$

Figure 3.9. Refined Type Checking for λ_{tgt}

3.4.1 Refinement Type Checking

We present a brief overview of refinement typing as the target language falls under the scope of existing refinement type systems [66], which can, after accounting for dead-casts, be reused *as is* for the second phase. Similarly, we limit the presentation to *checking*; *inference* follows directly from Liquid Type inference [91]. Figure 3.9 summarizes the refinement system. The type checking judgment is

$$G \vdash w :: T$$

where *type environment* G is a sequence of bindings of variables x to refinement types T and *guard predicates*, which encode control flow information gathered by conditional checks. As is standard [66] each primitive constant n has a refined type b, and a variable x with type T is typed as $\mathsf{sngl}\,(T, x)$ which is $\{v\colon b \mid v = x\}$ if T is a basic type b and T otherwise.

Checking dead**-casts.** The refinement system verifies dead-casts by treating them as special function calls, *i.e.* discharging them via the application rule R-APP. Formally, $\mathsf{dead}\downarrow^{\tau_1}_{\tau_2}(w)$ is treated as call to:

$$\mathsf{dead}\downarrow^{\tau_1}_{\tau_2} :: \mathsf{Bot}([\tau_1]) \rightarrow \mathsf{Bot}([\tau_2])$$

The notation $[\cdot]$ denotes the elaboration of λ_{src} types to λ_{tgt} types [31]:

$$\begin{aligned}
[\mathsf{b}] &\doteq \mathsf{b} \\
[\tau_1 \wedge \tau_2] &\doteq [\tau_1] \times [\tau_2] \\
[\tau_1 \vee \tau_2] &\doteq [\tau_1] + [\tau_2] \\
[\tau_1 \rightarrow \tau_2] &\doteq [\tau_1] \rightarrow [\tau_2]
\end{aligned}$$

The meta-function $\mathsf{Bot}(\mathsf{T}) \doteq \mathsf{Tx}(\mathsf{T}, \mathit{false})$ where:

$$\begin{aligned}
\mathsf{Tx}(\mathsf{b}, \mathsf{r}) &\doteq \{\mathsf{v}\colon \mathsf{b} \mid \mathsf{r}\} \\
\mathsf{Tx}(\mathsf{T}_1 + \mathsf{T}_2, \mathsf{r}) &\doteq \mathsf{Tx}(\mathsf{T}_1, \mathsf{r}) + \mathsf{Tx}(\mathsf{T}_2, \mathsf{r}) \\
\mathsf{Tx}(\mathsf{T}_1 \rightarrow \mathsf{T}_2, \mathsf{r}) &\doteq \mathsf{Tx}(\mathsf{T}_1, \neg\mathsf{r}) \rightarrow \mathsf{Tx}(\mathsf{T}_2, \mathsf{r}) \\
\mathsf{Tx}(\mathsf{T}_1 \times \mathsf{T}_2, \mathsf{r}) &\doteq \mathsf{Tx}(\mathsf{T}_1, \mathsf{r}) \times \mathsf{Tx}(\mathsf{T}_2, \mathsf{r})
\end{aligned}$$

Returning to Rule R-APP for dead-casts and inverting, expression w gets assigned a refinement type T. For simplicity we assume this is a base type b. Due to R-SUB we get the subtyping constraint

$$\mathsf{G} \vdash \{\mathsf{v}\colon \mathsf{b} \mid \mathsf{P}\} \sqsubseteq \{\mathsf{v}\colon \mathsf{b} \mid \mathit{false}\}$$

which generates the VC

$$\mathsf{Valid}([\![\mathsf{G}]\!] \wedge [\![\mathsf{P}]\!] \Rightarrow [\![\mathit{false}]\!])$$

This holds if the environment combined with the refinement in the left-hand side is inconsistent, which means that the gathered flow conditions are infeasible, hence dead-code [66]. Thus, the refinements statically ensure that the specially marked dead values are *never created at run-time*. As only dead terms cause execution to get stuck, the refinement verification phase ensures that the source is indeed type safe.

Conditional Checking. R-IF and R-CASE check each branch of a conditional or case splitting statement, by enhancing the environment with a guard (w or $\neg w$) or the right binding ($x\colon T_1$ or $x\colon T_2$), that encode the boolean test performed at the condition, or the structural check at the pattern matching, respectively. Crucially, this allows the use of "tests" inside the code to statically verify dead-casts and other correctness properties. The other rules are standard and are described in the refinement type literature.

Correspondence of Elaboration and Refinement Typing. The following result establishes the fact that the type τ assigned to a source expression e by elaboration and the type T assigned by refinement type checking to the elaborated expression w are connected with the relation: $[\tau] = \|T\|$, where $\|T\|$ is merely a (recursive) elimination of all refinements appearing in T. The notation $[\Gamma] = \|G\|$ means that for each binding $x\colon \tau \in \Gamma$ there exists $x\colon T \in G$, such that $[\tau] = \|T\|$, and vice versa.

Lemma 3.6 (Correspondence)**.** *If* $\Gamma \vdash e : \tau \hookrightarrow w$, $G \vdash w :: T$ *and* $[\Gamma] = \|G\|$, *then* $[\tau] = \|T\|$.

Proof. By induction on pairs of derivations: $\Gamma \vdash e : \tau \hookrightarrow w$ and $G \vdash w :: T$. Details of this proof can be found in the appendix (Lemma B.11). □

The target language satisfies a progress and preservation theorem [66]:

Theorem 3.7 (Refinement Type Safety)**.** *If* $\cdot \vdash w \ : \ T$ *then either* w *is a value or there exists* w' *such that* $w \longrightarrow w'$ *and* $\cdot \vdash w' \ : \ T$.

Proof. Given by Rondon et al. [91] for a similar language. □

3.4.2 Two-Phase Type Safety

We say that a source term e is *well two-typed* if there exists a source type τ, target term w and target (refinement) type T such that: (1) $\cdot \vdash e : \tau \hookrightarrow w$, and, (2) $\cdot \vdash w :: T$. That is, e is well two-typed if it elaborates to a refinement typed target. The Consistency Theorems 3.4 and 3.5, along with the Safety Theorem 3.7, yield end-to-end soundness: well two-typed terms do not get stuck, and step to well two-typed terms.

Theorem 3.8 (Two-Phase Soundness)**.** *If* e *is well two-typed then, either* e *is a value, or there exists* e' *such that:*

(1) ***(Progress)*** $e \longrightarrow e'$

(2) ***(Preservation)*** e' *is well two-typed.*

Proof. By induction on pairs of derivations: $\Gamma \vdash e : \tau \hookrightarrow w$ and $G \vdash w :: T$. Details are to be found in the appendix (Theorem B.14). □

3.5 Related Work

Several relevant research directions regarding type systems for functional and imperative dynamic languages were introduced in Section 1.3.1. This section attempts to compare Two-Phased Typing with related work in the areas of intersection and union types, program logics and refinement type systems.

Intersection and Union Types. Central to our elaboration phase are intersection and union types. Pierce [80] indicates the connection between unions and intersections with sums and products, that is the basis of Dunfield's elaboration scheme [31] on which we build. However, Dunfield studies *static* source languages that use *explicit* overloading via a merge operator [89]. In contrast, we target *dynamic* source languages with implicit value based overloading, and hence must account for "ill-typed" terms via dead-casts discharged via the second phase refinement check. Castagna *et al.* [16] describe a λ&-calculus, where functions are overloaded by combining several different branches of code. The branch to be executed is determined at run-time by using the arguments' typing information. This technique resembles the code duplication that happens in our approach, but overload resolution (*i.e.* deciding which branch is executed) is determined at runtime whereas we do so statically.

Furr et al. [44] present DRuby, a tool for type inference for Ruby scripts combining several practices from earlier work on soft typing, gradual typing and contracts [36]. DRuby uses intersection types to *represent* summaries for overloaded functions. However, these systems do not handle value-based overloading (like TypeScript, DRuby allows overloaded specifications for external functions).

Refinement Types. DML [113] is an early refinement type system composing ML's types with a decidable constraint system. *Hybrid type checking* [66] uses arbitrary refinements over basic types. A static type system verifies basic specifications and more complex ones are defered to dynamically checked contracts, since the specification logic is statically undecidable. In these cases, the source language is well typed (ignoring refinements), and lacks intersections

and unions. Our second phase can use Liquid Types [91] to infer refinements using predicate abstraction.

Program Logics for Dynamic Languages. The intuition of expressing subtyping relations as logical implication constraints and using SMT solvers to discharge these constraints allows for a more extensive variety of typing idioms. Bierman *et al.* [11] investigate semantic subtyping in a first order language with refinements and type-test expressions.

In *nested refinement types* [21], the typing relation itself is a predicate in the refinement logic and a feature-rich language of predicates accounts for heavily dynamic idioms, like run-time type tests, value-indexed dictionaries, polymorphism and higher order functions. While program logics allow the use of arbitrary tests to establish typing, the circular dependency between values and basic types leads to two significant problems in theory and practice. First, the circular dependency complicates the *metatheory* which makes it hard to add extra (basic) typing features (*e.g.* polymorphism, classes) to the language. Second, the circular dependency complicates the *inference* of types and refinements, leading to significant annotation overheads which make the system difficult to use in practice. In contrast, two-phase typing allows arbitrary type tests while enabling the trivial composition of soundness proofs and inference algorithms.

Kent et al. [64] present *Occurrence Typing Modulo Theories*, a type system combining occurrence typing, a technique for checking dynamic languages that was discussed in Section 1.3.1, with dependent refinement types. Their technique allows the integration of arbitrary solver-backed reasoning about logical propositions from external theories, leading to an expressive overall system. As with previous approaches, however, this technique is limited with respect to inferring refinements compared to the approach we follow in this work.

Acknowledgements

This chapter contains material adapted from the following publication: **Trust, but Verify: Two-Phase Typing for Dynamic Languages** appearing in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP'15).* Prague, Czech Republic, July 2015. Vekris, Panagiotis; Cosman, Benjamin; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Refinement Types for TypeScript

Modern *scripting* languages – like JavaScript, Python, and Ruby – have popularized the use of higher-order constructs that were once solely in the *functional* realm. This trend towards abstraction and reuse poses two related problems for static analysis: *modularity* and *extensibility*. First, how should analysis precisely track the flow of values across higher-order functions and containers or *modularly* account for external code like closures or library calls? Second, how can analyses be easily *extended* to new, domain specific properties, ideally by developers, while they are designing and implementing the code? (As opposed to by experts who can at best develop custom analyses run *ex post facto* and are of little use *during* development.)

Refinement types hold the promise of a precise, modular and extensible analysis for programs with higher-order functions and containers. Unfortunately, attempts to apply refinement typing to scripts have proven to be impractical due to the interaction of the machinery that accounts for imperative updates and higher-order functions [20] (Section 4.5).

In this chapter, we introduce Refined TypeScript (RSC): a novel, *lightweight* refinement type system for TypeScript, a typed superset of JavaScript. Our design of RSC addresses three intertwined problems by carefully integrating and extending existing ideas from the literature. First, RSC accounts for *mutation* by using ideas from Immutability Generic Java [116] to track which fields may be mutated, and to allow refinements to depend on immutable fields, and by using SSA-form to recover path and flow-sensitivity. Second, RSC accounts for *dynamic typing* by using the Two-Phase Typing technique described in Chapter 3, where dynamic behaviors are specified via union and intersection types, and verified by reduction to refinement typing. Third, the above are carefully designed to permit refinement *inference* via the Liquid Types [91] framework to render refinement typing practical on real-world programs. Concretely, we make

the following contributions:

- We develop a core calculus that permits locally flow-sensitive reasoning via SSA translation and formalizes the interaction of mutability and refinements via declarative refinement type checking that we prove sound (Section 4.2).
- We extend the core language to TypeScript by describing how we account for its various *dynamic* and *imperative* features; in particular we show how RSC accounts for type reflection via intersection types, encodes interface hierarchies via refinements and handles object initialization (Section 4.3).
- We implement `rsc`, a refinement type checker for TypeScript, and evaluate it on a suite of real-world programs from the Octane benchmarks, Transducers, D3 and the TypeScript compiler [1]. We show that RSC's refinement typing is *modular* enough to analyze higher-order functions, collections and external code, and *extensible* enough to verify a variety of properties from classic array-bounds checking to program specific invariants needed to ensure safe reflection: critical invariants that are well beyond the scope of existing techniques for imperative scripting languages (Section 4.4).

4.1 Overview

In Section 3.1 we gave a high-level overview of refinement types and showed how they can be used to verify properties like within bounds array accesses. The form in which they were presented there is readily applicable to the setting of Refined TypeScript. For completeness we briefly review them and demonstrate their applications (Section 4.1.1). Then we move on to show how Refined TypeScript handles imperative, higher-order constructs (Section 4.1.2).

Types and Refinements. A basic refinement type is a basic type, *e.g.* `number`, refined

[1] Our implementation and benchmarks can be found at https://github.com/UCSD-PL/refscript.

with a logical formula from an SMT decidable logic [74]. For example, the types

$$\begin{aligned}
\texttt{type nat} &= \{\nu\texttt{: number} \mid 0 \leq \nu\} \\
\texttt{type pos} &= \{\nu\texttt{: number} \mid 0 < \nu\} \\
\texttt{type natN}\langle\texttt{n}\rangle &= \{\nu\texttt{: nat} \mid \nu = \texttt{n}\} \\
\texttt{type idx}\langle\langle\texttt{a}\rangle\rangle &= \{\nu\texttt{: nat} \mid \nu < \texttt{len(a)}\}
\end{aligned}$$

describe (the set of values corresponding to) *non-negative* numbers, *positive* numbers, numbers *equal to* some value `n`, and *valid indexes* for an array `a`, respectively. Here, `len` is an *uninterpreted function* that describes the size of the array `a`.

Summaries. Function types $(\texttt{x}_1 : \tau_1, \ldots, \texttt{x}_n : \tau_n) \Rightarrow \tau$, where arguments are named $\texttt{x}_i$ and have types τ_i and the output is a type τ, are used to specify the behavior of functions. In essence, the *input* types τ_i specify the function's preconditions, and the *output* type τ describes the postcondition. Each input type and the output type can *refer to* the arguments $\texttt{x}_i$, yielding precise function contracts. For example,

$$(\texttt{x: nat}) \Rightarrow \{\nu\texttt{: nat} \mid \texttt{x} < \nu\}$$

is a function type that describes functions that *require* a non-negative input, and *ensure* that the output exceeds the input.

Higher-Order Summaries. This approach generalizes directly to precise descriptions for *higher-order* functions. Take for example the code in Figure 4.1. This code is the same as the one in Figure 3.1. It has been duplicated here for convenience. The function `reduce` can be specified as $\mathsf{T}_{\texttt{reduce}}$:

$$\forall\alpha, \beta\,.\,(\texttt{a: } \alpha\texttt{[]}, \texttt{f: } (\beta, \alpha, \texttt{idx}\langle\langle\texttt{a}\rangle\rangle) \Rightarrow \beta, \texttt{x: } \beta) \Rightarrow \beta \tag{4.1}$$

This type is a precise *summary* for the higher-order behavior of `reduce`: it describes the relationship between the input array `a`, the step ("callback") function `f`, and the initial value of the accumulator, and stipulates that the output satisfies the same *properties* β as the input `x`. Furthermore, it critically specifies that the callback `f` is only invoked on valid indices for the

```
function reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}

function minIndex(a) {
  if (a.length <= 0) return -1;
  function step(min, cur, i) {
    return cur < a[min] ? i : min;
  }
  return reduce(a, step, 0);
}
```

Figure 4.1. Computing the Min-Valued Index with `reduce`

array `a` being reduced.

4.1.1 Applications

Next, we show how refinement types let programmers *specify* and statically *verify* a variety of properties — array safety, reflection (value-based overloading), and downcasts — potential sources of runtime problems that cannot be prevented via existing techniques.

Array Bounds

Specification. We specify safety by defining suitable refinement types for array creation and access. For example, we view read `a[i]`, write `a[i] = e` and length access `a.length` as calls `get(a,i)`, `set(a,i,e)` and `length(a)` where

$$\begin{aligned} \texttt{get} &:: \forall\alpha.(\texttt{a}\colon \alpha[\,], \texttt{i}\colon \texttt{idx}\langle\langle \texttt{a}\rangle\rangle) \Rightarrow \alpha \\ \texttt{set} &:: \forall\alpha.(\texttt{a}\colon \alpha[\,], \texttt{i}\colon \texttt{idx}\langle\langle \texttt{a}\rangle\rangle, \texttt{e}\colon \alpha) \Rightarrow \texttt{void} \\ \texttt{length} &:: \forall\alpha.(\texttt{a}\colon \alpha[\,]) \Rightarrow \texttt{natN}\langle \texttt{a}\rangle \end{aligned}$$

Verification. Refinement typing ensures that the *actual* parameters supplied at each *call* to `get` and `set` are subtypes of the *expected* values specified in the signatures, and thus verifies that all accesses are safe. As an example, consider the function that returns the "head" element

of an array:

```
function head<T>(arr: NEArray<T>) {
  return arr[0];
}
```

The input type requires that `arr` be *non-empty*:

$$\texttt{type NEArray}\langle\alpha\rangle = \{\,\texttt{v: }\alpha\texttt{[] } \mid \texttt{ } 0 < \texttt{len}(\texttt{v})\,\}$$

We convert `arr[0]` to `get(arr,0)` which is checked under environment $\Gamma_{\texttt{head}}$ defined as

$$\texttt{arr: }\{\,\texttt{v: T[] } \mid \texttt{ } 0 < \texttt{len}(\texttt{v})\,\}$$

yielding the subtyping obligation

$$\Gamma_{\texttt{head}} \vdash \{\texttt{v} = 0\} \sqsubseteq \texttt{idx}\langle\langle\texttt{arr}\rangle\rangle$$

which reduces to the logical *verification condition* (VC)

$$0 < \texttt{len}(\texttt{arr}) \Rightarrow (\texttt{v} = 0 \Rightarrow 0 \leq \texttt{v} < \texttt{len}(\texttt{arr}))$$

The VC is proved *valid* by an SMT solver [74], verifying subtyping, and hence, the array access' safety.

Path Sensitivity. We obtain path sensitivity by adding branch conditions into the typing environment. Consider the function:

```
function head0(a: number[]): number {
  if (0 < a.length) return head(a);
  return 0;
}
```

Recall that `head` should only be invoked with *non-empty* arrays. The call to `head` above occurs under $\Gamma_{\texttt{head0}}$ defined as:

$$\texttt{a: number[], } 0 < \texttt{len}(\texttt{a})$$

i.e. which has the binder for the formal `a`, and the guard predicate established by the branch

condition. Thus, the call to `head` yields the obligation

$$\Gamma_{\texttt{head0}} \vdash \{\nu = \texttt{a}\} \sqsubseteq \texttt{NEArray}\langle\texttt{number}\rangle$$

yielding the valid VC

$$0 < \texttt{len}(\texttt{a}) \Rightarrow (\nu = \texttt{a} \Rightarrow 0 < \texttt{len}(\nu))$$

Polymorphic, Higher-Order Functions. Next, let us *assume* that `reduce` from Figure 3.1 has the type $\mathsf{T}_{\texttt{reduce}}$ and see how to verify the array safety of `minIndex`. The challenge here is to precisely track which values can flow into `min` (used to index into `a`), which is tricky since those values are actually produced inside `reduce`.

Types make it easy to track such flows: we need only determine the *instantiation* of the polymorphic type variables of `reduce` at this call site inside `minIndex`. The type of the `f` parameter in the instantiated type corresponds to a signature for the closure `step` which will let us verify the closure's implementation. Here, `rsc` automatically instantiates (by building complex logical predicates from simple terms that have been predefined in a prelude)

$$\alpha \mapsto \texttt{number} \qquad\qquad \beta \mapsto \texttt{idx}\langle\langle\texttt{a}\rangle\rangle \tag{4.2}$$

Let us reassure ourselves that this instantiation is valid, by checking that `step` and `0` satisfy the instantiated type. If we substitute (4.2) into $\mathsf{T}_{\texttt{reduce}}$ we obtain the following types for `step` and `0`, *i.e.* `reduce`'s second and third arguments:

$$\texttt{step} :: (\texttt{idx}\langle\langle\texttt{a}\rangle\rangle, \texttt{number}, \texttt{idx}\langle\langle\texttt{a}\rangle\rangle) \Rightarrow \texttt{idx}\langle\langle\texttt{a}\rangle\rangle \qquad\qquad \texttt{0} :: \texttt{idx}\langle\langle\texttt{a}\rangle\rangle$$

The initial value `0` is indeed a valid $\texttt{idx}\langle\langle\texttt{a}\rangle\rangle$ thanks to the `a.length` check at the start of the function. To check `step`, assume that its inputs have the above types:

$$\texttt{min} :: \texttt{idx}\langle\langle\texttt{a}\rangle\rangle \qquad\qquad \texttt{curr} :: \texttt{number} \qquad\qquad \texttt{i} :: \texttt{idx}\langle\langle\texttt{a}\rangle\rangle$$

The body is safe as the index `i` is trivially a subtype of the required $\texttt{idx}\langle\langle\texttt{a}\rangle\rangle$, and the output is one of `min` or `i` and hence, of type $\texttt{idx}\langle\langle\texttt{a}\rangle\rangle$ as required.

Overloading

Dynamic languages extensively use *value-based overloading* to simplify library interfaces. For example, a library may export

```
function $reduce(a, f, x) {
  if (arguments.length === 3) return reduce(a,f,x);
  return reduce(a.slice(1),f,a[0]);
}
```

The function `$reduce` has *two* distinct types depending on its parameters' *values*, rendering it impossible to statically type without path-sensitivity.

Intersection Types. Refinements let us statically *verify* value-based overloading via *Two-Phase Typing* (Chapter 3). First, we specify overloading as an intersection type. For example, `$reduce` gets the following signature, which is just the conjunction of the two overloaded behaviors:

$$\forall\alpha\,.\,(\texttt{a}\colon \alpha\texttt{[]}^{+},\ \texttt{f}\colon (\alpha,\ \alpha,\ \texttt{idx}\langle\langle\texttt{a}\rangle\rangle) \Rightarrow \alpha) \Rightarrow \alpha \tag{4.3}$$

$$\forall\alpha,\beta\,.\,(\texttt{a}\colon \alpha\texttt{[]},\ \texttt{f}\colon (\beta,\ \alpha,\ \texttt{idx}\langle\langle\texttt{a}\rangle\rangle) \Rightarrow \beta,\ \texttt{x}\colon \beta) \Rightarrow \beta \tag{4.4}$$

The type $\alpha\texttt{[]}^{+}$ in the first conjunct indicates that the first argument needs to be a non-empty array, so that the call to `slice` and the access of `a[0]` both succeed.

Dead Code Assertions. Second, we check each conjunct separately, replacing ill-typed terms in each context with `assert(false)`. This requires the refinement type checker to prove that the corresponding expressions are *dead code*, as `assert` requires its argument to always be `true`:

$$\texttt{assert} \ :: \ \forall\alpha\,.\,(\texttt{b}\colon \{\,\nu\colon \texttt{boolean} \mid \nu = \texttt{true}\,\}) \Rightarrow \alpha$$

To check `$reduce`, we specialize it per overload context as can be seen in Figure 4.2 In each case, the "ill-typed" term (for the corresponding input context) is replaced with the call `assert(false)`. Refinement typing easily verifies the `asserts`, as they respectively occur under the *inconsistent* environments

$$\Gamma_1 \ \doteq \ \texttt{arguments}\colon \{\texttt{len}(\nu) = 2\},\ \texttt{len}(\texttt{arguments}) = 3$$

$$\Gamma_2 \ \doteq \ \texttt{arguments}\colon \{\texttt{len}(\nu) = 3\},\ \texttt{len}(\texttt{arguments}) \neq 3$$

```
function $reduce1(a,f) {
  if (arguments.length === 3) return assert(false);
  return reduce(a.slice(1), f, a[0]);
}

function $reduce2(a,f,x) {
  if (arguments.length === 3) return reduce(a,f,x);
  return assert(false);
}
```

Figure 4.2. Specialization of `$reduce` Function

which bind `arguments` to an array-like object corresponding to the arguments passed to that function, and include the branch condition under which the call to `assert` occurs.

4.1.2 Analysis

Next, we outline how `rsc` uses refinement types to analyze programs with closures, polymorphism, assignments, classes and mutation.

Polymorphic Instantiation

`rsc` uses the framework of Liquid Typing [91] to *automatically synthesize* the instantiations of (4.2). In a nutshell, `rsc`

(a) creates *templates* for unknown refinement type instantiations,

(b) performs type checking over the templates to generate *subtyping constraints* over the templates that capture value-flow in the program,

(c) solves the constraints via a *fixpoint* computation (abstract interpretation).

Step 1: Templates. Recall that `reduce` has the polymorphic type $T_{\texttt{reduce}}$. At the call-site in `minIndex`, the type variables `A`, `B` are instantiated with the *known* base-type `number`. Thus, `rsc` creates fresh templates for the (instantiated) α, β:

$$\alpha \mapsto \{\, v\colon \texttt{number} \mid \kappa_A \,\} \qquad\qquad \beta \mapsto \{\, v\colon \texttt{number} \mid \kappa_B \,\}$$

where the *refinement variables* κ_A and κ_B represent the *unknown refinements*. We substitute the

above in the signature for `reduce` to obtain a *context-sensitive* template

$$(\texttt{a}\colon \kappa_A\,\texttt{[]},\ \texttt{f}\colon (\kappa_B,\ \kappa_A,\ \texttt{idx}\langle\langle \texttt{a}\rangle\rangle) \Rightarrow \kappa_B,\ \texttt{x}\colon \kappa_B) \Rightarrow \kappa_B \tag{4.5}$$

Step 2: Constraints. Next, `rsc` generates *subtyping* constraints over the templates. Intuitively, the templates describe the *sets* of values that each static entity (*e.g.* variable) can evaluate to at runtime. The subtyping constraints capture the *value-flow* relationships *e.g.* at assignments, calls and returns, to ensure that the template solutions – and hence inferred refinements – soundly over-approximate the set of runtime values of each corresponding static entity.

We generate constraints by performing type checking over the templates. As `a`, `0`, and `step` are passed in as arguments, we check that they respectively have the types $\kappa_A\,\texttt{[]}$, κ_B and $(\kappa_B,\ \kappa_A,\ \texttt{idx}\langle\langle \texttt{a}\rangle\rangle) \Rightarrow \kappa_B$. Checking `a` and `0` yields the subtyping constraints

$$\Gamma \vdash \texttt{number[]} \sqsubseteq \kappa_A\,\texttt{[]}$$
$$\Gamma \vdash \{\nu = 0\} \sqsubseteq \kappa_B$$

where $\Gamma \doteq \texttt{a}\colon \texttt{number[]},\ 0 < \texttt{len(a)}$ from the *else-guard* that holds at the call to `reduce`. We check `step` by checking its body under the environment $\Gamma_{\texttt{step}}$ that binds the input parameters to their respective types

$$\Gamma_{\texttt{step}} \doteq \texttt{min}\colon \kappa_B,\ \texttt{cur}\colon \kappa_a,\ \texttt{i}\colon \texttt{idx}\langle\langle \texttt{a}\rangle\rangle$$

As `min` is used to index into the array `a` we get

$$\Gamma_{\texttt{step}} \vdash \kappa_B \sqsubseteq \texttt{idx}\langle\langle \texttt{a}\rangle\rangle$$

As `i` and `min` flow to the output type κ_B, we get

$$\Gamma_{\texttt{step}} \vdash \texttt{idx}\langle\langle \texttt{a}\rangle\rangle \sqsubseteq \kappa_B$$
$$\Gamma_{\texttt{step}} \vdash \kappa_B \sqsubseteq \kappa_B$$

Step 3: Fixpoint. The above subtyping constraints over the κ variables are reduced via the standard rules for co- and contra-variant subtyping, into *Horn implications* over the κs. `rsc`

solves the Horn implications via (predicate) abstract interpretation [91] to obtain the solution $\kappa_A \mapsto$ `true` and $\kappa_B \mapsto 0 \leq \nu < \texttt{len(a)}$ which is exactly the instantiation in (4.2) that satisfies the subtyping constraints, and proves `minIndex` is array-safe.

Assignments

Next, let us see how the signature for `reduce` in Figure 4.1 is verified by `rsc`. Unlike in the functional setting, where refinements have previously been studied, here, we must deal with imperative features like assignments and `for`-loops.

SSA Transformation. We solve this problem in three steps. First, we convert the code into SSA form, to introduce new binders at each assignment. Second, we generate fresh templates that represent the unknown types (*i.e.* set of values) for each Φ-variable. Third, we generate and solve the subtyping constraints to infer the types for the Φ-variables, and hence, the "loop-invariants" needed for verification.

Let us see how this process lets us verify `reduce` from Figure 4.1. First, we convert the body to SSA form (Section 4.2.3):

```
function reduce(a, f, x) {
  var r0 = x, i0 = 0;
  while [i2 ≐ ϕ(i0,i1), r2 ≐ ϕ(r0,r1)] (i2 < a.length) {
    r1 = f(r2, a[i2], i2);
    i1 = i2 + 1;
  }
  return r2;
}
```

where `i2` and `r2` are the Φ-variables for `i` and `r` respectively. Second, we generate templates for the Φ-variables:

$$\texttt{i2}: \{\, \nu : \texttt{number} \mid \kappa_{i2} \,\} \qquad\qquad \texttt{r2}: \{\, \nu : \texttt{B} \mid \kappa_{r2} \,\} \tag{4.6}$$

We need not generate templates for the SSA variables `i0`, `r0`, `i1` and `r1` as their types are those of the expressions they are assigned. Third, we generate subtyping constraints as before; the

Φ-assignment generates *additional* constraints

$$\Gamma_0 \vdash \{\nu = \texttt{i0}\} \sqsubseteq \kappa_{i2} \qquad \Gamma_1 \vdash \{\nu = \texttt{i1}\} \sqsubseteq \kappa_{i2}$$
$$\Gamma_0 \vdash \{\nu = \texttt{r0}\} \sqsubseteq \kappa_{r2} \qquad \Gamma_1 \vdash \{\nu = \texttt{r1}\} \sqsubseteq \kappa_{r2}$$

where Γ_0 is the environment at the "exit" of the basic block where `i0` and `r0` are defined:

$$\Gamma_0 \doteq \texttt{a}\colon \texttt{number[]},\ \texttt{x}\colon \beta,\ \texttt{i0}\colon \texttt{natN}\langle 0\rangle,\ \texttt{r0}\colon \{\nu\colon \beta \mid \nu = \texttt{x}\}$$

Similarly, the environment Γ_1 includes bindings for variables `i1` and `r1`. In addition, code executing the loop body has passed the conditional check, so our *path-sensitive* environment is strengthened by the corresponding guard:

$$\Gamma_1 \doteq \Gamma_0,\ \texttt{i1}\colon \texttt{natN}\langle \texttt{i2} + 1\rangle,\ \texttt{r1}\colon \beta,\ \texttt{i2} < \texttt{len}(\texttt{a})$$

Finally, the above constraints are solved to

$$\kappa_{i2} \mapsto\ 0 \leq \nu < \texttt{len}(\texttt{a}) \qquad \kappa_{r2} \mapsto\ \texttt{true}$$

which verifies that the "callback" `f` is indeed called with values of type $\texttt{idx}\langle\langle \texttt{a}\rangle\rangle$, as it is only called with `i2`: $\texttt{idx}\langle\langle \texttt{a}\rangle\rangle$, obtained by plugging the solution into the template in (4.6).

Mutability

In the imperative, object-oriented setting (common in dynamic scripting languages), we must account for *class* and *object* invariants and their preservation in the presence of field mutation. For example, consider the code in Figure 4.3, modified from the Octane Navier-Stokes benchmark.

Class Invariants. Class `Field` implements a *2-dimensional* vector, "unrolled" into a single array `dens`, whose size is the product of the `width` and `height` fields. We specify this invariant by requiring that width and height be strictly positive (*i.e.* `pos`) and that `dens` be a `grid` with dimensions specified by `this.w` and `this.h`. An advantage of SMT-based refinement typing is that modern SMT solvers support non-linear reasoning, which lets `rsc` specify and verify program specific invariants outside the scope of generic bound checkers.

```
type ArrayN⟨α, n⟩ = {v: α[] | len(v) = n}
type grid⟨w, h⟩ = ArrayN⟨number, (w + 2) * (h + 2)⟩
type okW = natLE⟨this.w⟩
type okH = natLE⟨this.h⟩

class Field {
  immutable w: pos;
  immutable h: pos;
  dens       : grid⟨this.w, this.h⟩;

  constructor(w: pos, h: pos, d: grid⟨w, h⟩) {
    this.h = h; this.w = w; this.dens = d;
  }
  setDensity(x: okW, y: okH, d: number) {
    var rowS = this.w + 2;
    var i    = x+1 + (y+1) * rowS;
    this.dens[i] = d;
  }
  getDensity(x: okW, y: okH): number {
    var rowS = this.w + 2;
    var i    = x+1 + (y+1) * rowS;
    return this.dens[i];
  }
  reset(d: grid⟨w, h⟩) {
    this.dens = d;
  }
}
```

Figure 4.3. Example Adapted from D3: Two-Dimensional Arrays

Mutable and Immutable Fields. The above invariants are only meaningful and sound if fields `w` and `h` cannot be modified after object creation. We specify this via the `immutable` qualifier, which is used by `rsc` to then (1) *prevent* updates to the field outside the `constructor`, and (2) *allow* refinements of fields (*e.g.* `dens`) to soundly refer to the values of those immutable fields.

Constructors. We can create *instances* of `Field`, by using `new Field(...)` which invokes the `constructor` with the supplied parameters. `rsc` ensures that at the *end* of the constructor, the created object actually satisfies all specified class invariants *i.e.* field refinements. Of course, this only holds if the parameters passed to the constructor satisfy certain preconditions, specified

via the input types. Consequently, rsc accepts the first call, but rejects the second:

```
var z = new Field(3,7,new Array(45)); // OK
var q = new Field(3,7,new Array(44)); // BAD
```

Methods. rsc uses class invariants to verify setDensity and getDensity, that are checked *assuming* that the fields of this enjoy the class invariants, and method inputs satisfy their given types. The resulting VCs are valid and hence, check that the methods are array-safe. Of course, clients must supply appropriate arguments to the methods. Thus, rsc accepts the first call, but rejects the second as the x co-ordinate 5 exceeds the actual width (*i.e.* z.w), namely 3:

```
z.setDensity(2, 5, -5)    // OK
z.getDensity(5, 2);       // BAD
```

Mutation. The dens field is *not* immutable and hence, may be updated outside of the constructor. However, rsc requires that the class invariants still hold, and this is achieved by ensuring that the *new* value assigned to the field also satisfies the given refinement. Thus, the reset method requires inputs of a specific size, and updates dens accordingly. Hence:

```
var z = new Field(3,7,new Array(45));
z.reset(new Array(45));  // OK
z.reset(new Array(5));   // BAD
```

4.2 Formal System

Next, we formalize the ideas outlined in Section 4.1. We introduce our formal core I_{rsc} (Section 4.2.1): an imperative, mutable, object-oriented subset of Refined TypeScript, that resembles the core of Safe TypeScript [87]. To ease refinement reasoning, we SSA-transform (Section 4.2.3) I_{rsc} to a functional, yet still mutable, intermediate language λ_{rsc} (Section 4.2.2) that closely follows the design of CFJ [76] (the language used to formalize X10), which in turn is based on Featherweight Java [57]. We then formalize our static semantics in terms of λ_{rsc} (Section 4.2.4), prove them sound and connect them to those of I_{rsc} (Section 4.2.5).

4.2.1 Source Language (I_{rsc})

The syntax of this language is given below. Meta-variable e ranges over expressions, which can be variables x, constants n, property accesses $e.f$, method calls $e.m(\overline{e})$, object creations **new** $C(\overline{e})$, and cast operations <T>e. Here T is a source-level basic type (*i.e.* unrefined).

Variables	x	$\in$	$\mathcal{X}$
Constants	n	$\in$	*Consts*
Expressions	e	$::=$	$x \mid n \mid \texttt{this} \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid$
			$\texttt{new}\ C(\overline{e}) \mid \texttt{<T>}e$
Statements	s	$::=$	$e \mid \texttt{var}\ x = e \mid x = e \mid s; s \mid \texttt{if}\,(e)\,\{s\}\,\texttt{else}\,\{s\}$
Field Def.	F	$::=$	$\cdot \mid f \mid F_1, F_2$
Method Def.	M	$::=$	$\cdot \mid m(\overline{x}) : \{s; \texttt{return}\ e\} \mid M_1, M_2$
Class Def.	K	$::=$	$\cdot \mid \texttt{class}\ C\ \texttt{extends}\ D\ \{F, M\} \mid K_1, K_2$
Program	P	$::=$	$K\ s$
Field Sig.	$\widehat{F}$	$::=$	$\cdot \mid \circ f\colon T \mid \square f\colon T \mid \widehat{F}_1, \widehat{F}_2$
Method Sig.	$\widehat{M}$	$::=$	$\cdot \mid m(\overline{x}\colon \overline{T})\colon T \mid \widehat{M}_1, \widehat{M}_2$
Class Sig.	$\widehat{K}$	$::=$	$C \triangleleft D :: P \left\{\widehat{F}, \widehat{M}\right\}$

Figure 4.4. Syntax of I_{rsc}

Statements s include expressions, variable declarations, field updates, assignments, concatenations, conditionals and empty statements. Method definitions include a method name, parameters and a body, *i.e.* a statement immediately followed by a returned expression. Methods are annotated with method signatures that include input ant output types. Classes include methods and fields. We distinguish between immutable and mutable class members, using $\circ\, f\colon T$ and $\square\, f\colon T$, respectively. Finally, class signatures, that annotate classes, are associated with an invariant predicate P.

The core system does not formalize (a) method overloading, which is orthogonal to the current contribution and was investigated in Chapter 3, or (b) method overriding, which means that method names are distinct from the ones defined in parent classes.

4.2.2 Intermediate Language (λ_{rsc})

To maintain precision for stack-allocated variables, we transform I_{rsc} programs into equivalent (in a sense that we will make precise in the sequel) programs in a functional language λ_{rsc} through SSA renaming. In λ_{rsc}, statements are replaced by let-bindings and new variables are introduced for each reassigned variable in I_{rsc} code. Thus, λ_{rsc} has the syntax shown in

Expressions	w	::=	$z \mid n \mid \textbf{this} \mid w.f \mid w.m(\overline{w}) \mid \textbf{new}\ C(\overline{w}) \mid$
			$w\ \textbf{as}\ T \mid w_1.f \leftarrow w_2 \mid u\langle w \rangle$
SSA Context	u	::=	$\langle\,\rangle \mid \textbf{let}\ z = w\ \textbf{in}\ u \mid \textbf{if}\,[\,\overline{\phi}\,]\,w\ \textbf{then}\ u_1\ \textbf{else}\ u_2$
Phi Variable	ϕ	::=	$(z; z_1; z_2)$
Field Def.	$\mathcal{F}$	::=	$\cdot \mid f \mid \mathcal{F}_1, \mathcal{F}_2$
Method Def.	$\mathcal{M}$	::=	$\cdot \mid \textbf{def}\ m(\overline{z}) = w \mid \mathcal{M}_1, \mathcal{M}_2$
Class Def.	$\mathcal{K}$	::=	$\cdot \mid \textbf{class}\ C\ \textbf{extends}\ D\ \{\mathcal{F}, \mathcal{M}\} \mid \mathcal{K}_1, \mathcal{K}_2$
Program	$\mathcal{P}$	::=	$\mathcal{K}, w$

Figure 4.5. Syntax of λ_{rsc}

Figure 4.5.

The majority of the expression forms e are unsurprising. An exception is the form of the *SSA context* u, which corresponds to the translation of a statement s and contains a *hole* $\langle\,\rangle$ that will hold the translation of the continuation of s. Form $u\langle e \rangle$ fills the hole of u with expression e.

4.2.3 Static Single Assignment (SSA) Transformation

Figure 4.6 describes the SSA transformation, that uses *translation environments* δ to map I_{rsc} x to λ_{rsc} variables z. The translation of expressions e to w is routine; as expected, S-VAR maps a variable x to its bindings z in δ. The translating judgment for statements s has the form

$$\delta \vdash s \hookrightarrow u \dashv \delta'$$

The output environment δ' is used for the translation of the expression that will fill the hole in u.

The most interesting case is the conditional statement (Rule S-ITE). The condition expression and each branch are translated separately. To compute the variables that get updated in either branch (Φ-variables), we combine the produced translation states δ_1 and δ_2 as $\delta_1 \bowtie \delta_2$ defined as

$$\{\,(x; x_1; x_2) \mid x \mapsto x_1 \in \delta_1,\, x \mapsto x_2 \in \delta_2,\, x_1 \neq x_2\,\}$$

Fresh Φ-variables $\overline{z}$ populate the output environment δ' and annotate the produced structure, along with the versions of the Φ-variables at the end of each branch ($\overline{z}_1$ and $\overline{z}_2$).

Assignment statements introduce a new SSA variable and bind it to the updated source-

Expression Transformation Rules (selected) $\boxed{\delta \vdash e \hookrightarrow w}$

$$\frac{}{\delta \vdash x \hookrightarrow \delta(x)}\ \text{[S-VAR]} \qquad \frac{}{\delta \vdash n \hookrightarrow n}\ \text{[S-CONST]} \qquad \frac{}{\delta \vdash \texttt{this} \hookrightarrow \textbf{this}}\ \text{[S-THIS]}$$

$$\frac{\delta \vdash e \hookrightarrow w}{\delta \vdash e.f \hookrightarrow w.f}\ \text{[S-FLDRD]} \qquad \frac{\delta \vdash e \hookrightarrow w \qquad \delta \vdash e' \hookrightarrow w'}{\delta \vdash e.f = e' \hookrightarrow w.f \leftarrow w'}\ \text{[S-FLDWR]}$$

$$\frac{\delta \vdash e \hookrightarrow w \qquad \delta \vdash e_i \hookrightarrow \overline{w}_i}{\delta \vdash e.m(\overline{e}_i) \hookrightarrow w.m(\overline{w}_i)}\ \text{[S-CALL]}$$

Statement Transformation Rules (selected) $\boxed{\delta \vdash s \hookrightarrow u \dashv \delta'}$

$$\frac{\delta \vdash e \hookrightarrow w \qquad \delta[x \mapsto z] = \delta' \qquad z \text{ fresh}}{\delta \vdash \texttt{let } x = e \hookrightarrow \textbf{let } z = w \textbf{ in } \langle\,\rangle \dashv \delta'}\ \text{[S-VARDECL]}$$

$$\frac{\begin{array}{c}\delta \vdash e \hookrightarrow w \qquad \delta \vdash s_1 \hookrightarrow u_1 \dashv \delta_1 \qquad \delta \vdash s_2 \hookrightarrow u_2 \dashv \delta_2 \\ \delta_1 \bowtie \delta_2 = \overline{x} \mapsto (\overline{z}_1, \overline{z}_2) \qquad \delta[\overline{x} \mapsto \overline{z}] = \delta \qquad (\overline{z}; \overline{z}_1; \overline{z}_2) = \phi \qquad \overline{z} \text{ fresh}\end{array}}{\delta \vdash \texttt{if } (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \hookrightarrow \textbf{if } [\,\phi\,]\, w \textbf{ then } u_1 \textbf{ else } u_2 \dashv \delta'}\ \text{[S-ITE]}$$

$$\frac{\delta \vdash e \hookrightarrow w \qquad \delta[x \mapsto z'] = \delta' \qquad z' \text{ fresh}}{\delta \vdash x = e \hookrightarrow \textbf{let } z' = w \textbf{ in } \langle\,\rangle \dashv \delta'}\ \text{[S-ASGN]}$$

$$\frac{\delta \vdash s_1 \hookrightarrow u_1 \dashv \delta_1 \qquad \delta_1 \vdash s_2 \hookrightarrow u_2 \dashv \delta_2}{\delta \vdash s_1\,;\, s_2 \hookrightarrow u_1\langle u_2\rangle \dashv \delta_2}\ \text{[S-SEQ]} \qquad \frac{}{\delta \vdash \texttt{skip} \hookrightarrow \langle\,\rangle \dashv \delta}\ \text{[S-SKIP]}$$

$$\frac{\delta \vdash s \hookrightarrow u \dashv \delta' \qquad \delta' \vdash e \hookrightarrow w}{\delta \vdash s; \texttt{return } e \hookrightarrow u\langle w\rangle}\ \text{[S-BODY]}$$

Method Transformation Rule $\boxed{M \hookrightarrow \mathcal{M}}$

$$\frac{\overline{x} \mapsto \overline{z} \vdash \{s; \texttt{return } e\} \hookrightarrow w \qquad \overline{z} \text{ fresh}}{m(\overline{x}) : \{s; \texttt{return } e\} \hookrightarrow \textbf{def } m(\overline{z}) = w}\ \text{[S-METH-DEF]}$$

Figure 4.6. SSA Transformation in RSC

level variable (Rule S-ASGN). Statement sequencing is emulated with nesting SSA contexts (Rule S-SEQ); empty statements introduce a hole (rule S-SKIP); and, finally, method declarations fill in the hole introduced by the method statement with the translation of the return expression (Rule S-METH-DEF).

Consistency. To validate our transformation, we provide a consistency result that guarantees that stepping in the target language preserves the transformation relation, after the program in the source language has made an appropriate number of steps.

We define a *runtime configuration* R for $\mathrm{I}_{\mathsf{rsc}}$ (resp. $\mathcal{R}$ for λ_{rsc}) for a *program* P (resp. $\mathcal{P}$) as:

$$\mathsf{P} \doteq \mathsf{K}, e \qquad \mathcal{P} \doteq \mathcal{K}, w$$

$$\mathsf{R} \doteq \mathsf{S}, e \qquad \mathcal{R} \doteq \mathcal{S}, w$$

$$\mathsf{S} \doteq \langle \mathsf{K}; \mathsf{L}; \mathsf{X}; \mathsf{H} \rangle \qquad \mathcal{S} \doteq \mathcal{K}, \mathcal{H}$$

Similar to Safe TypeScript [87], a *runtime state* S consists of *class signatures* K, a *call stack* X, a *local store* L of the current stack frame and a *heap* H. The runtime state $\mathcal{S}$ for λ_{rsc} only consists of signatures $\mathcal{K}$ and a heap $\mathcal{H}$. SSA *consistency* is established via a weak forward simulation theorem that connects the dynamic semantics of the two languages, expressed through the reduction rules

$$\mathsf{R} \longrightarrow \mathsf{R}' \qquad \mathcal{R} \longrightarrow \mathcal{R}'$$

Rules for $\mathrm{I}_{\mathsf{rsc}}$ are adapted from Safe TypeScript and the rules for λ_{rsc} are straightforward, so we leave the details to the appendix (Section C.2.1). Figure C.2 presents some interesting cases:

(a) To emulate `tsc`'s type erasure, Rule R-CAST of $\mathrm{I}_{\mathsf{rsc}}$ trivially steps a cast operation to the enclosed expression. The corresponding Rule R-CAST of λ_{rsc}, on the other hand, checks that the content of the cast location satisfies the cast type (Corollary 4.2.1 deems this check redundant).

(b) In Rule R-LIF of λ_{rsc}, expression e is produced assuming Φ-variables $\overline{x}$, so as soon as the branch has been determined, $\overline{x}$ are substituted for $\overline{x}_1$ or $\overline{x}_2$ (depending on the branch) in e. This formulation allows us to perform all SSA-related book-keeping in a single step, which is key to preserving the *invariant* that λ_{rsc} steps faster than $\mathrm{I}_{\mathsf{rsc}}$.

We also extend our SSA transformation judgment to runtime configurations, leveraging

Ref. Types	T	$::=$	$\exists x{:}\, T_1 \,.\, T_2 \mid \{ v{:}\, N \mid P \}$
Base Types	N	$::=$	$C \mid b$
Predicates	P	$::=$	$P_1 \wedge P_2 \mid \neg P \mid t$
Terms	t	$::=$	$x \mid n \mid v \mid$ **this** $\mid t.f \mid f(\bar{t}) \mid b(\bar{t})$

Figure 4.7. Type Language in RSC

the SSA environments that have been statically computed for each program entity. A *global SSA environment* Δ is used to map each AST node (e, s, *etc.*) to an SSA environment δ:

$$\Delta \;::=\; \cdot \mid e \mapsto \delta \mid s \mapsto \delta \mid \ldots \mid \Delta_1, \Delta_2$$

We assume that the compile-time SSA translation yields this environment as a side-effect (*e.g.* $\delta \vdash e \hookrightarrow w$ produces $e \mapsto \delta$) and the top-level program transformation judgment returns the net effect:

$$P \hookrightarrow \mathcal{P} \triangleright \Delta$$

Hence, the SSA transformation judgment for configurations becomes:

$$S, e \overset{\Delta}{\hookrightarrow} \mathcal{S}, w$$

We can now state our simulation theorem as:

Theorem 4.1 (Forward Simulation)**.** *If* $R \overset{\Delta}{\hookrightarrow} \mathcal{R}$*, then:*

(a) if $\mathcal{R}$ *is terminal, then there exists* R' *s.t.* $R \longrightarrow^* R'$ *and* $R' \overset{\Delta}{\hookrightarrow} \mathcal{R}$*.*

(b) if $\mathcal{R} \longrightarrow \mathcal{R}'$*, then there exists* R' *s.t.* $R \longrightarrow^* R'$ *and* $R' \overset{\Delta}{\hookrightarrow} \mathcal{R}'$*.*

4.2.4 Static Semantics

We proceed by describing refinement checking for λ_{rsc}.

Types. Type annotations on the source language are propagated unaltered through the translation phase. Our type language shown in Figure 4.7 resembles that of existing refinement type systems [66, 91, 76]. A *refinement type* T may be an existential type or have the form $\{ v{:}\, N \mid P \}$, where N is a class name C or a primitive type b, and P is a logical predicate (over

some decidable logic) which describes the properties that values of the type must satisfy. Type specifications (*e.g.* method types) are existential-free, while inferred types may be existentially quantified [65].

Logical Predicates. Predicates P are logical formulas over terms t. These terms can be variables z, primitive constants n, the reserved value variable ν, the reserved variable **this** to denote the containing object, field accesses $t.f$, uninterpreted function applications $f(\bar{t})$ and applications of terms on built-in operators b, such as $=$, $<$, $+$, etc.

Structural Constraints. Following CFJ, we reuse the notion of an Object Constraint System, to encode constraints related to the object-oriented nature of the program. Most of the rules carry over to our system. A key extension in our setting is we partition C has I (that encodes inclusion of an element I in a class C) into two cases: C hasMut I and C hasImm I, to account for elements that may be mutated.

These elements can only be fields (*i.e.* there is no mutation on methods).

Environments and Well-formedness. A type environment Γ contains *type bindings* $x : T$ and *guard predicates* P that encode path sensitivity. Γ is *well-formed* if all of its bindings are well-formed. A refinement type is well-formed in an environment Γ if all symbols (simple or qualified) in its logical predicate (i) are bound in Γ, and (ii) correspond to *immutable* fields of objects. We omit the rest of the well-formedness rules as they are standard in refinement type systems. Besides well-formedness, our system's main judgment forms are those for subtyping and refinement typing [66].

Subtyping. We defined subtyping by the judgment:

$$\Gamma \vdash T_1 \ \leq \ T_2$$

The rules are standard among refinement type systems with existential types. For example, the rule for subtyping between two refinement types

$$\Gamma \vdash \{\nu : N \mid P\} \leq \{\nu : N \mid P'\}$$

reduces to a *verification condition*

$$\mathsf{Valid}([\![\Gamma]\!] \Rightarrow ([\![P]\!] \Rightarrow [\![P']\!]))$$

Expression Typing Rules $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma(z) = T}{\Gamma \vdash z : \mathsf{sngl}\,(T,\, z)}\ [\text{T-Var}] \qquad \frac{}{\Gamma \vdash n : b_n}\ [\text{T-Cst}] \qquad \frac{\Gamma \vdash u \triangleright \overline{z}\colon \overline{T} \qquad \Gamma,\, \overline{z}\colon \overline{T} \vdash e : T}{\Gamma \vdash u\langle e \rangle : \exists\, \overline{z}\colon \overline{T}\,.\, T}\ [\text{T-Ctx}]$$

$$\frac{\Gamma \vdash e : T \qquad y \text{ fresh} \qquad \Gamma,\, y\colon T \vdash y \text{ hasImm } f_i\colon T_i}{\Gamma \vdash e.f_i : \exists\, y\colon T\,.\, \mathsf{sngl}\,(T_i,\, y.f_i)}\ [\text{T-Fld-I}]$$

$$\frac{\Gamma \vdash e : T \qquad \Gamma,\, y\colon T \vdash y \text{ hasMut } g_i : T_i \qquad y \text{ fresh}}{\Gamma \vdash e.g_i : \exists\, y\colon T\,.\, T_i}\ [\text{T-Fld-M}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T,\, \overline{e} : \overline{T} \\ \Gamma,\, y\colon T \vdash y \text{ has } (m(\overline{y}\colon \overline{T}_1)\colon T_2) \qquad \Gamma,\, y\colon T,\, \overline{y}\colon \overline{T}_1 \vdash \overline{T} \le \overline{T}_1 \qquad y,\, \overline{y} \text{ fresh}\end{array}}{\Gamma \vdash e.m(\overline{e}) : \exists\, y\colon T\,.\, \exists\, \overline{y}\colon \overline{T}_1\,.\, T_2}\ [\text{T-Mth-Call}]$$

$$\frac{\Gamma \vdash e_1 : T_1,\, e_2 : T_2 \qquad \Gamma,\, y_1 : \lfloor T_1 \rfloor \vdash y_1 \text{ hasMut } f\colon T_2',\, T_2 \le T_2' \qquad y_1 \text{ fresh}}{\Gamma \vdash e_1.f \leftarrow e_2 : T_2}\ [\text{T-DotAsgn}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{e} : (\overline{T}_\circ,\, \overline{T}_\square) \qquad \vdash \mathsf{class}(C) \qquad \Gamma,\, y\colon C \vdash \mathsf{fields}(y) = \circ\, \overline{f}\colon \overline{T}'_\circ,\, \square\, \overline{g}\colon \overline{T}'_\square \\ \Gamma,\, y\colon C,\, \overline{y}_\circ\colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right) \vdash \overline{T}_\circ \le \overline{T}'_\circ,\, \overline{T}_\square \le \overline{T}'_\square,\, \mathsf{inv}(C,\, y) \qquad y, \overline{y}_\circ \text{ fresh}\end{array}}{\Gamma \vdash \textbf{new } C(\overline{e}) : \exists\, \overline{y}_\circ\colon \overline{T}_\circ\,.\,\{\nu\colon C \mid \nu.\overline{f} = \overline{y}_\circ \wedge \mathsf{inv}(C,\, \nu)\}}\ [\text{T-New}]$$

$$\frac{\Gamma \vdash e : T' \qquad \Gamma \vdash T \qquad \Gamma \vdash T' \lesssim T}{\Gamma \vdash e \textbf{ as } T : T}\ [\text{T-Cast}]$$

SSA Context Typing Rules $\boxed{\Gamma \vdash u \triangleright \Gamma'}$

$$\frac{}{\Gamma \vdash \langle\,\rangle \triangleright \cdot}\ [\text{T-CtxEmp}] \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{let } z = e \textbf{ in } \langle\,\rangle \triangleright z\colon T}\ [\text{T-LetIn}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T,\, T \le \mathsf{Bool} \qquad \Gamma,\, z\colon T,\, z \vdash u_1 \triangleright \Gamma_1 \qquad \Gamma,\, z\colon T,\, \neg z \vdash u_2 \triangleright \Gamma_2 \\ \phi \equiv (\overline{z};\, \overline{z}_1;\, \overline{z}_2) \qquad \Gamma,\, \Gamma_1 \vdash \Gamma_1(\overline{z}_1) \le \overline{T} \qquad \Gamma,\, \Gamma_2 \vdash \Gamma_2(\overline{z}_2) \le \overline{T} \qquad \Gamma \vdash \overline{T} \qquad \overline{T} \text{ fresh}\end{array}}{\Gamma \vdash \textbf{if } [\,\phi\,]\, e \textbf{ then } u_1 \textbf{ else } u_2 \triangleright \overline{z}\colon \overline{T}}\ [\text{T-If}]$$

Figure 4.8. Static Typing Rules for λ_{rsc}

where $[\![\Gamma]\!]$ is the embedding of environment Γ into our logic accounting for both guard predicates and variable bindings:

$$[\![\Gamma]\!] \doteq \bigwedge \{ P \mid P \in \Gamma \} \wedge \bigwedge \{ [x/\nu](P), \mid x : \{ \nu : N \mid P \} \in \Gamma \}$$

Here, we assume existential types are simplified to non-existential bindings when they enter the environment.

Details regarding structural and well-formedness constraints, and subtyping rules are included in Section C.1.4 of the appendix.

Refinement Typing Rules. Figure 4.8 contains rules of the two forms of our typing judgements:

$$\Gamma \vdash e : T \qquad\qquad \Gamma \vdash u \triangleright \Gamma'$$

The first assigns a type T to an expression e under an environment Γ, and the second checks the body of an SSA context u under Γ and returns the environment Γ' of the variables introduced in u that are available when checking its hole (Rule T-CTX). Below, we discuss the novel rules:

[**T-FLD-I**] Immutable object parts can be assigned a more precise type, by leveraging the preservation of their *identity*. This notion, known as *self-strengthening* [65, 76], is defined with the aid of the *strengthening* operator $\oplus$:

$$\begin{aligned} \{ \nu : N \mid P \} \oplus P' &\doteq \{ \nu : N \mid P \wedge P' \} \\ (\exists x : T_1 \,.\, T_2) \oplus P &\doteq \exists x : T_1 \,.\, (T_2 \oplus P) \\ \mathsf{sngl}(T, t) &\doteq T \oplus (\nu = t) \end{aligned}$$

[**T-FLD-M**] Here we avoid such strengthening, as the value of field g_i is mutable, so cannot appear in refinements.

[**T-NEW**] Similarly, only immutable fields are referenced in the refinement of the inferred type at object construction.

[**T-MTH-CALL**] Extracting the method signature using the has operator has already performed the necessary substitutions to account for the specific receiver object.

[**T-CAST**] Cast operations are checked *statically* obviating the need for a dynamic check. This rule uses the notion of *compatibility subtyping* ($\lesssim$), which is defined as:

Definition 4.2.1 (Compatibility Subtype)**.** *A type* T_1 *is a compatibility subtype of a type* T_2 *if transforming* T_1 *to match the base of* T_2 *results in a subtype of* τ_2*. Formally,*

$$\Gamma \vdash \mathsf{T}_1 \lesssim \mathsf{T}_2$$

iff

$$\langle \mathsf{T}_1 \xrightarrow{\Gamma} \lfloor \mathsf{T}_2 \rfloor \rangle = \mathsf{T}_1' \neq \mathsf{fail}$$

with $\Gamma \vdash \mathsf{T}_1' \leq \mathsf{T}_2$.

Here, the operation $\lfloor \mathsf{T} \rfloor$ extracts the base type of T, and $\langle \mathsf{T} \xrightarrow{\Gamma} \mathsf{D} \rangle$ succeeds when under environment Γ we can statically prove D's invariants, starting from the invariants contained in T. We use the predicate $\mathsf{inv}(\mathsf{D}, \nu)$ (as in CFJ) to denote the conjunction of the class invariants of D and its supertypes (with the necessary substitutions of **this** by ν). We assume that part of these invariants is a predicate that states inclusion in the specific class (*instanceof*(ν, D)). Therefore, we can prove that T can safely be cast to D. For the output of this operation it holds that: $\lfloor \langle \mathsf{T} \xrightarrow{\Gamma} \mathsf{D} \rangle \rfloor = \mathsf{D}$, which enables the use of traditional subtyping. Formally:

$$\langle \{\, \nu \colon \_ \mid \mathsf{P} \,\} \xrightarrow{\Gamma} \mathsf{D} \rangle \doteq \begin{cases} \mathsf{D} \Cap \mathsf{P} & \text{if } (\llbracket \Gamma \rrbracket \wedge \llbracket \mathsf{P} \rrbracket) \implies \mathsf{inv}(\mathsf{D}, \nu) \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

$$\langle \exists\, x \colon \mathsf{T}_1 \,.\, \mathsf{T}_2 \xrightarrow{\Gamma} \mathsf{D} \rangle \doteq \exists\, x \colon \mathsf{T}_1 \,.\, \langle \mathsf{T}_2 \xrightarrow{\Gamma, x \colon \mathsf{T}_1} \mathsf{D} \rangle$$

[**T-DOTASGN**] Only *mutable* fields may be reassigned.

[**T-LETIF**] To type conditional structures, we first infer a type for the condition and then check each of the branches u_1 and u_2, assuming that the condition is true or false, respectively, to achieve path sensitivity. Each branch assigns types to the Φ-variables which compose Γ_1 and Γ_2, and the propagated types for these variables are fresh types operating as upper bounds to their respective bindings in Γ_1 and Γ_2.

4.2.5 Type Safety

To state our safety results, we extend our type checking judgment to runtime locations ℓ with the use of a *heap typing* Σ, binding locations to types, and add a location typing rule:

$$\frac{\Sigma(\ell) = \mathsf{T}}{\Gamma;\, \Sigma \vdash \ell \colon \mathsf{T}}\ \text{[T-Loc]}$$

We establish type safety for λ_{rsc} in the form of a subject reduction (preservation) and a progress theorem that connect the static and dynamic semantics of λ_{rsc}. These theorems employ the notions of heap and signature well-formedness: $\Sigma \vdash \mathcal{H}$ and $\vdash \mathcal{K}$.

Theorem 4.2 (Subject Reduction)**.** *If*

(i) $\Gamma;\, \Sigma \vdash e \colon \mathsf{T}$

(ii) $\mathcal{S};\, e \longrightarrow \mathcal{S}';\, e'$

(iii) $\Sigma \vdash \mathcal{S}.\mathcal{H}$

then there exist T' *and* $\Sigma' \supseteq \Sigma$ *s.t.*

(a) $\Gamma;\, \Sigma' \vdash e' \colon \mathsf{T}'$

(b) $\Gamma \vdash \mathsf{T}' \lesssim \mathsf{T}$

(c) $\Sigma' \vdash \mathcal{S}'.\mathcal{H}$

Theorem 4.3 (Progress)**.** *If*

(i) $\Gamma;\, \Sigma \vdash e \colon \mathsf{T}$

(ii) $\vdash \mathcal{K}$

(iii) $\Sigma \vdash \mathcal{H}$

then either e *is a value, or there exist* e', $\mathcal{H}'$ *and* $\Sigma' \supseteq \Sigma$ *s.t.*

(a) $\Sigma' \vdash \mathcal{H}'$

(b) $\mathcal{K},\, \mathcal{H};\, e \longrightarrow \mathcal{K},\, \mathcal{H}';\, e'$

The proofs can be found in Section C.2.2 of the appendix. As a corollary of the Progress Theorem we get that cast operators are guaranteed to succeed, hence they can safely be erased.

Corollary 4.2.1 (Safe Casts)**.** *Cast operations can safely be erased when compiling to executable code.*

With the use of our Simulation Theorem and extending our checking judgment for terms in λ_{rsc} to runtime configurations ($\vdash \mathcal{R}$), we can state a soundness result for $\mathrm{I}_{\mathsf{rsc}}$:

Theorem 4.4. *($\mathrm{I}_{\mathsf{rsc}}$ Type Safety) If* $\mathtt{R} \overset{\Delta}{\hookrightarrow} \mathcal{R}$ *and* $\vdash \mathcal{R}$ *then either* $\mathtt{R}$ *is a terminal form, or there exists* $\mathtt{R}'$ *s.t.* $\mathtt{R} \longrightarrow \mathtt{R}'$, $\mathtt{R}' \overset{\Delta}{\hookrightarrow} \mathcal{R}'$ *and* $\vdash \mathcal{R}'$.

4.3 Scaling to TypeScript

TypeScript extends JavaScript with modules, classes and a lightweight type system that enables IDE support for auto-completion and refactoring.

TypeScript deliberately eschews soundness [10] for backwards compatibility with existing JavaScript code. In this section, we show how to use refinement types to *regain safety*, by presenting the highlights of Refined TypeScript (and our tool `rsc`), that scales the core calculus from Section 4.2 up to TypeScript by extending the support for *types* (Section 4.3.1), *reflection* (Section 4.3.2), *interface hierarchies* (Section 4.3.3), and *imperative* programming (Section 4.3.4).

4.3.1 Types

First, we discuss how `rsc` handles core TypeScript features like object literals, interfaces and primitive types.

Object Literal Types. TypeScript supports object literals, *i.e.* anonymous objects with field and method bindings. `rsc` types object members in the same way as class members: method signatures need to be explicitly provided, while field types and mutability modifiers are inferred based on use, *e.g.* in:

```
var point = { x: 1, y: 2 };
point.x = 2;
```

the field `x` is updated and hence, `rsc` infers that `x` is mutable.

Interfaces. TypeScript supports named object types in the form of interfaces, and treats them in the same way as their *structurally* equivalent class types. For example, the interface

```
interface PointI {
  number x, y;
}
```

is equivalent to a class `PointC` defined as

```
class PointC {
  number x, y;
}
```

In `rsc` these two types are *not* equivalent, as objects of type `PointI` do not necessarily have `PointC` as their constructor:

```
var pI = { x: 1, y: 2 };
var pC = new PointC(1,2);
pI instanceof PointC;   // returns false
pC instanceof PointC;   // returns true
```

However,

$$\cdot \vdash \texttt{PointC} \leq \texttt{PointI}$$

i.e. instances of the *class* may be used to implement the *interface*.

Primitive Types. We extend `rsc`'s support for primitive types to model the corresponding types in TypeScript. TypeScript has `undefined` and `null` types to represent the eponymous values, and treats these types as the "bottom" of the type hierarchy, effectively allowing those values to inhabit *every* type via subtyping. `rsc` also includes these two types, but *does not* treat them as "bottom" types. Instead `rsc` handles them as distinct primitive types inhabited solely by `undefined` and `null`, respectively, that can take part in unions. Consequently, the following code is accepted by TypeScript but *rejected* by `rsc`:

```
var x = undefined;
var y = x + 1;
```

Unsound Features. TypeScript's system is unsound due to

1. treating `undefined` and `null` as inhabitants of all types,
2. co-variant input subtyping,
3. allowing unchecked overloads, and
4. allowing a special "dynamic" `any` type to be ascribed to any term.

`rsc` ensures soundness by

1. performing checks when non-null (non-undefined) types are required (*e.g.* during field accesses),
2. using the correct variance for functions and constructors,
3. checking overloads via two-phase typing (Chapter 3), and,
4. *eliminating* the `any` type.

Many uses of `any` (indeed, *all* uses, in our benchmarks Section 4.4) can be replaced with a combination of union or intersection types or downcasting, all of which are soundly checked via path-sensitive refinements. In future work, we wish to support the full language, namely allow *dynamically* checked uses of `any` by incorporating orthogonal dynamic techniques from the contracts literature. We envisage a *dynamic cast* operation

$$\texttt{cast}_{\mathsf{T}} \;::\; (\texttt{x}\colon \texttt{any}) \Rightarrow \{\, \nu\colon \mathsf{T} \mid \nu = \texttt{x} \,\}$$

It is straightforward to implement $\texttt{cast}_{\mathsf{T}}$ for first-order types T as a dynamic check that traverses the value, testing that its components satisfy the refinements [92]. Wrapper-based techniques from the contracts/gradual typing literature should then let us support higher-order types.

4.3.2 Reflection

JavaScript programs make extensive use of reflection via "dynamic" type tests. `rsc` statically accounts for these by encoding type-tags in refinements. The following tests if `x` is a `number` before performing an arithmetic operation on it:

```
var r = 1;
if (typeof x === "number") {
  r += x;
}
```

We account for this idiomatic use of `typeof` by *statically* tracking the "type" tag of values inside refinements using uninterpreted functions (akin to the size of arrays). So a type τ is refined with the predicate $\texttt{ttag} = \texttt{tag}(\tau)$ where `tag` was defined in Figure 3.6. For example, values `v` of type `boolean`, `number`, `string`, *etc.* `ttag(v) = "boolean"`, `ttag(v) = "number"`,

`ttag(v) = "string"`, *etc.*. Furthermore, `typeof` has type

$$\texttt{typeof} \;::\; \forall\alpha.(\texttt{z}\colon \alpha) \Rightarrow \{\,\texttt{v}\colon \texttt{string} \mid \texttt{v} = \texttt{ttag}(\texttt{z})\,\}$$

so the output type of `typeof x` and the path-sensitive guard under which the assignment `r = x + 1` occurs, ensures that at the assignment `x` can be statically proven to be a `number`.

The above technique coupled with two-phase typing (Chapter 3) allows `rsc` to statically verify reflective, value-overloaded functions that are ubiquitous in TypeScript (Section 3.1.1).

4.3.3 Interface Hierarchies

JavaScript programs frequently build up object hierarchies that represent *unions* of different kinds of values, and then use value tests to determine which kind of value is being operated on. In TypeScript this is encoded by building up a hierarchy of interfaces, and then performing *downcasts* based on *value* tests[2].

Implementing Hierarchies with Bit-vectors. Figure 4.9 describes a slice of the hierarchy of types used by the TypeScript compiler (`tsc`) v1.0.1.0. `tsc` uses bit-vector valued flags to encode membership in a particular interface type, *i.e.* discriminate between the different entities. (*Older* versions of `tsc` used a class-based approach, where inclusion could be tested via `instanceof` tests.) For example, the enumeration `TypeFlags` above maps semantic entities to bit-vector values used as masks that determine inclusion in a sub-interface of `Type`. Suppose `t` of type `Type`. The invariant here is that if `t.flags` masked with `0x00000800` is non-zero, then `t` can be safely treated as an `InterfaceType` object, or an `ObjectType` object, since the relevant flag emerges from the bit-wise disjunction of the `Interface` flag with some other flags.

Specifying Hierarchies with Refinements. `rsc` allows developers to *create* and *use* `Type` objects with the above invariant by specifying a predicate `typeInv` (Figure 4.10) and then refining `TypeFlags` with the predicate [3]:

$$\texttt{type}\ \texttt{TypeFlags} = \{\,\texttt{v}\colon \texttt{TypeFlags} \mid \texttt{typeInv}\langle\langle\texttt{v}\rangle\rangle\,\}$$

Intuitively, the refined type says that when `v` (that is the `flags` field) is a bit-vector with the first

[2] `rsc` handles other type tests, *e.g.* `instanceof`, via an extension of the technique used for `typeof` tests.

[3] Modern SMT solvers easily handle formulas over bit-vectors, including operations that shift, mask bit-vectors, and compare them for equality.

```
interface Type {
  immutable flags: TypeFlags;
  id: number;
  symbol?: Symbol;
  ...
}

interface ObjectType extends Type { ...  }

interface InterfaceType extends ObjectType {
  baseTypes: ObjectType[];
  declaredProperties: Symbol[];
  ...
}

enum TypeFlags {
  Any       = 0x00000001,
  String    = 0x00000002,
  Number    = 0x00000004,
  Class     = 0x00000400,
  Interface = 0x00000800,
  Reference = 0x00001000,
  Object    = Class | Interface | Reference
  ...
}
```

Figure 4.9. Type Hierarchies in the `tsc` Compiler

position set to `1` the corresponding object satisfies the `AnyType` interface, etc.

Verifying Downcasts. `rsc` *verifies* the code that uses ad hoc hierarchies such as the above by proving the TypeScript *downcast* operations (that allow objects to be used at particular instances) safe. For example, consider the following code that *tests* if `t` implements the `ObjectType` interface before performing a downcast from type `Type` to `ObjectType` that permits the access of the latter's fields:

```
function getPropertiesOfType(t: Type): Symbol[] {
  if (t.flags & TypeFlags.Object) {
    var o = <ObjectType>t;
    [...]
  }
}
```

$$
\begin{aligned}
\texttt{isMask}\langle\langle \nu, m, \tau \rangle\rangle &\doteq \texttt{mask}(\nu, m) \implies \texttt{implements}(\texttt{this}, \tau) \\
\texttt{typeInv}\langle\langle \nu \rangle\rangle &\doteq \texttt{isMask}\langle\langle \nu, \texttt{0x00000001}, \texttt{Anytype} \rangle\rangle \\
&\wedge\ \texttt{isMask}\langle\langle \nu, \texttt{0x00000002}, \texttt{StringType} \rangle\rangle \\
&\wedge\ \texttt{isMask}\langle\langle \nu, \texttt{0x00003C00}, \texttt{ObjType} \rangle\rangle
\end{aligned}
$$

Figure 4.10. Type Invariant Predicate Definition

`tsc` erases casts, thereby missing possible runtime errors. The same code *without* the if-test, or with a *wrong* test would pass the `tsc` type checker. `rsc`, on the other hand, checks casts *statically*. In particular, `<ObjectType>t` is treated as a call to a function with signature

$$\forall \alpha . (\texttt{x}: \{ \nu : \alpha \mid \texttt{implements}(\nu, \texttt{ObjectType}) \}) \Rightarrow \{ \nu : \texttt{ObjectType} \mid \nu = \texttt{x} \}$$

The if-test ensures that the *immutable* field `t.flags` masked with `0x00003C00` is non-zero, satisfying the third line in the type definition of `typeInv`, which in turn implies that `t` in fact implements the `ObjectType` interface.

4.3.4 Imperative Features

Immutability Guarantees. Our system uses ideas from Immutability Generic Java [116] (IGJ) to provide statically checked immutability guarantees. In IGJ a type reference is of the form `C<M,`$\overline{\texttt{T}}$`>`, where *immutability* argument `M` works as proxy for the immutability modifiers of the contained fields (unless overridden). It can be one of: `Immutable` (or `IM`), when neither this reference nor any other reference can mutate the referenced object; `Mutable` (or `MU`), when this and potentially other references can mutate the object; and `ReadOnly` (or `RO`), when this reference cannot mutate the object, but some other reference may. Similar reasoning holds for method annotations. IGJ provides *deep immutability*, since a class's immutability parameter is (by default) reused for its fields; however, this is not a firm restriction imposed by refinement type checking.

Arrays. TypeScript's definitions file provides a detailed specification for the `Array` interface. In Figure 4.11 we extend this definition to account for the mutating nature of certain array operations.

Mutating operations (`push`, `pop`, field updates) are only allowed on mutable arrays, and the type of `a.length` encodes the exact length of an immutable array `a`, and just a natural number

```
interface Array<M extends ReadOnly,T> {
   Mutable   pop(): T;
   Mutable   push(x:T): number;
   Immutable get length(): {ν: nat | ν = len(this)}
   ReadOnly  get length(): nat;
  ...
}
```

Figure 4.11. Array Interface with Mutability Annotations in Refined TypeScript

otherwise. For example, assume the following code:

```
for (var i = 0; i < a.length; i++) {
  var x = a[i];
  ..
}
```

To prove the access `a[i]` safe we need to establish `0` $\leq$ `i` and `i < a.length`. To guarantee that the length of `a` is constant, `a` needs to be immutable, so `rsc` will flag an error unless `a: Array<IM,T>`.

Object Initialization. Our formal core (Section 4.2) treats constructor bodies in a very limiting way: object construction is merely an assignment of the constructor arguments to the fields of the newly created object. In `rsc` we relax this restriction in two ways: (a) We allow class and field invariants to be violated *within* the body of the constructor, but checked for at the exit. (b) We permit the common idiom of certain fields being initialized *outside* the constructor, via an additional mutability variant that encodes reference *uniqueness*. In both cases, we still restrict constructor code so that it does not *leak* references of the constructed object (`this`) or *read* any of its fields, as they might still be in an uninitialized state.

(a) Internal Initialization: Constructors. Type invariants do not hold while the object is being "cooked" within the constructor. To safely account for this idiom, `rsc` defers the checking of class invariants (*i.e.* the types of fields) by replacing: (a) occurrences of `this.`f_i `=` e_i, with $\widehat{f_i} = e_i$, where $\widehat{f_i}$ are *local* variables, and (b) all return points with a call `ctor_init(`$\widehat{f_i}$`,...)`, where the signature for `ctor_init` is:

$$(\overline{x}\colon \overline{T}) \Rightarrow \texttt{void}$$

Thus, `rsc` treats field initialization in a field- and path-sensitive way (through the usual SSA conversion), and establishes the class invariants via a single atomic step at the constructor's exit

```
function createType(flags:TypeFlags):Type<IM> {
  const r: Type<UQ> = new Type(checker, flags);
  r.id = typeCount++;
  return r;
}
```

Figure 4.12. Initialization Outside the Constructor in Refined TypeScript

(return).

(b) External Initialization: Unique References. Sometimes we want to allow immutable fields to be initialized outside the constructor. Consider the code in Figure 4.12 (adapted from `tsc`).

Field `id` is expected to be *immutable*. However, its initialization happens after `Type`'s constructor has returned. Fixing the type of `r` to `Type<IM>` right after construction would disallow the assignment of the `id` field on the following line. So, instead, we introduce `Unique` (or `UQ`), a new mutability type that denotes that the current reference is the *only* reference to a specific object, and hence, allows mutations to its fields. This idea is related to the main idea bind Recency Types [53]. When `createType` returns, we can finally fix the mutability parameter of `r` to `IM`. We could also return `Type<UQ>`, extending the *cooking* phase of the current object and allowing further initialization by the caller. `UQ` references obey stricter rules to avoid leaking of unique references:

- they cannot be reassigned,
- they generally cannot be referenced, unless this occurs at a context that guarantees that no aliases will be produced, *e.g.* the context of `e1` in `e1.f = e2`, or the context of a returned expression, and
- they cannot be cast to types of a different mutability (*e.g.* `<C<IM>>x`), as this would allow the same reference to be subsequently aliased.

Section 4.5 discusses more expressive initialization approaches.

4.4 Evaluation

To evaluate `rsc`, we have used it to analyze a suite of JavaScript and TypeScript programs, to answer two questions: (1) What kinds of properties can be statically verified for real-world

code? (2) What kinds of annotations or overhead does verification impose? Next, we describe the properties, benchmarks and discuss the results.

Safety Properties. We verify with `rsc` the following:

- ***Property Accesses.*** `rsc` verifies that each field (`x.f`) or method lookup (`x.m(...)`) succeeds. Recall that `undefined` and `null` are not considered to inhabit the types to which the fields or methods belong.

- ***Array Bounds.*** `rsc` verifies that each array read (`x[i]`) or write (`x[i] = e`) occurs within the bounds of the array (`x`).

- ***Overloads.*** `rsc` verifies that functions with overloaded (*i.e.* intersection) types correctly implement the intersections in a path-sensitive manner as described in (Section 4.1.1).

- ***Downcasts.*** `rsc` verifies that at each TypeScript (down)cast of the form `<T>e`, the expression `e` is indeed an instance of `T`. This requires tracking program-specific invariants, *e.g.* bit-vector invariants that encode hierarchies (Section 4.3.3).

4.4.1 Benchmarks

We ported a number of existing JavaScript or TypeScript programs to `rsc`. We selected benchmarks that make heavy use of language constructs relevant to the safety properties described above. These include parts of the Octane test suite, developed by Google as a JavaScript performance benchmark [46] and already ported to TypeScript by Rastogi *et al.* [87], the TypeScript compiler [71], and the D3 [13] and Transducers [24] libraries:

- `navier-stokes`, which simulates two-dimensional fluid motion over time; `richards`, which simulates a process scheduler with several types of processes passing information packets; `splay`, which implements the *splay tree* data structure; and `raytrace`, which implements a raytracer that renders scenes involving multiple lights and objects; all from the Octane suite,

- `transducers`: a library that implements composable data transformations, a JavaScript port of Hickey's Clojure library, which is extremely dynamic in that some functions have 12 (value-based) overloads,

Table 4.1. Benchmark Results for `rsc` (Annotations). **LOC** is the number of non-comment lines of source (computed via `cloc` v1.62). The number of `rsc` specifications given as JML style comments is partitioned into **T** trivial annotations *i.e.* `tsc` type signatures, **M** mutability annotations, and **R** refinement annotations, *i.e.* those which actually mention invariants. **Time** is the number of seconds taken to analyze each file.

Benchmark	LOC	T	M	R	Time (s)
`navier-stokes`	366	3	18	39	473
`splay`	206	18	2	0	6
`richards`	304	61	5	17	7
`raytrace`	576	68	14	2	15
`transducers`	588	138	13	11	12
`d3-arrays`	189	36	4	10	37
`tsc-checker`	293	10	48	12	62
TOTAL	2522	334	104	91	

- `d3-arrays`: the array manipulating routines from the D3 [13] library, which makes heavy use of higher-order functions as well as value-based overloading,
- `tsc-checker`, which includes parts of the TypeScript compiler (v1.0.1.0), abbreviated as `tsc`. We check 15 functions from `compiler/core.ts` and 14 functions from `compiler/checker.ts` (for which we needed to import 779 lines of type definitions from `compiler/types.ts`). These code segments were selected among tens of thousands of lines of code comprising the compiler codebase, because they exemplified interesting properties, like the bit-vector based type hierarchies explained in Section 4.3.3.

Results. Figure 4.1 quantitatively summarizes the results of our evaluation. Overall, we had to add about 1 line of annotation per 5 lines of code (529 for 2522 LOC). The vast majority (334/529 or 63%) of the annotations are *trivial*, *i.e.* are TypeScript-like types of the form $(\texttt{x}\colon \texttt{nat}) \Rightarrow \texttt{nat}$; 20% (104/529) are trivial but have *mutability* information, and only 17% (91/529) mention refinements, *i.e.* are definitions like $\texttt{type nat} = \{\nu\colon \texttt{number} \mid 0 \leq \nu\}$ or dependent signatures like

$$\forall\alpha.(\texttt{a}\colon \alpha\texttt{[]}, \texttt{n}\colon \texttt{idx}\langle\langle\texttt{a}\rangle\rangle) \Rightarrow \alpha$$

These numbers show `rsc` has annotation overhead comparable to TypeScript, as in 83% of the cases the annotations are either identical to TypeScript annotations or to TypeScript annotations with some mutability modifiers. Of course, in the remaining 17% of the cases, the signatures are

more complex than the (non-refined) TypeScript version.

Code Changes. We had to modify the source in various small (but important) ways to facilitate verification. The total number of changes is summarized in Figure 4.2. The *trivial* changes include the addition of type annotations (accounted for above) and simple transformations to work around the current limitations of our front-end, *e.g.* converting `x++` to `x=x+1`. The *important* classes of changes are the following:

- ***Control-Flow:*** Some programs had to be restructured to work around `rsc`'s currently limited support for certain control flow structures (*e.g.* `break`). We also modified some loops to use explicit termination conditions.

- ***Classes and Constructors:*** As `rsc` does not yet support *default* constructor arguments, we changed relevant `new` calls in Octane to supply them explicitly, and refactored `navier-stokes` to use traditional OO style classes and constructors instead of JavaScript records with function fields.

- ***Non-null Checks:*** In `splay` we added 5 explicit non-null checks for mutable objects as proving those required precise heap analysis that is outside `rsc`'s scope.

- ***Ghost Functions:*** `navier-stokes` has more than a hundred (static) array access sites, most of which compute indices via non-linear arithmetic (*i.e.* via computed indices of the form `arr[r*s + c]`); SMT support for non-linear integer arithmetic is brittle (and accounts for the anomalous time for `navier-stokes`). We factored axioms about non-linear arithmetic into *ghost functions* whose types were proven once via non-linear SMT queries, and which were then explicitly called at use sites to instantiate the axioms (thereby bypassing non-linear analysis). An example of such a function is:

$$\texttt{mulThm} \;::\; (\texttt{a}\colon \texttt{nat}, \texttt{b}\colon \{\,\texttt{v}\colon \texttt{number} \mid \texttt{v} \geq 2\,\}) \Rightarrow \{\,\texttt{v}\colon \texttt{boolean} \mid \texttt{a} + \texttt{a} \leq \texttt{a} * \texttt{b}\,\}$$

  which, when *instantiated* via a call `mulThm(x, y)` establishes the fact that (at the call-site), `x + x <= x * y`. The reported performance assumes the use of ghost functions. In cases where they were not used `rsc` would time out.

Table 4.2. Changes Made on `rsc` Benchmarks. **LOC**: number of non-comment lines of source (computed via `cloc` v1.62). The *number of lines changed* is counted as either **ImpDiff**: *important* changes, such as restructuring the original JavaScript code to account for limited support for control flow constructs, replacing records with classes and constructors, and adding ghost functions; or **AllDiff**: the above plus *trivial* changes due to the addition of plain or refined type annotations (Figure 4.1), and simple edits to work around current limitations of our front-end.

Benchmark	LOC	ImpDiff	AllDiff
`navier-stokes`	366	79	160
`splay`	206	58	64
`richards`	304	52	108
`raytrace`	576	93	145
`transducers`	588	170	418
`d3-arrays`	189	8	110
`tsc-checker`	293	9	47
TOTAL	2522	469	1052

4.4.2 Transducers (A Case Study)

We now delve deeper into one of our benchmarks: the Transducers library. At its heart this library is about reducing collections, in other words performing folds. A Transformer is anything that implements three functions: `init` to begin computation, `step` to consume one element from an input collection, and `result` to perform any post-processing. One could imagine rewriting `reduce` from Figure 4.1 by building a Transformer where `init` returns `x`, `step` invokes `f`, and `result` is the identity[4]. The Transformers provided by the library are composable - their constructors take, as a final argument, another Transformer, and then all calls to the outer Transformer's functions invoke the corresponding one of the inner Transformer. This gives rise to the concept of a Transducer, a function of type $(\texttt{Transformer}) \Rightarrow \texttt{Transformer}$ and this library's namesake.

The main reason this library interests us is because some of its functions are massively overloaded. Consider, for example, the `reduce` function it defines in Figure 4.13. As discussed above, `reduce` needs a Transformer and a collection. There are two opportunities for overloading here. First of all, the main ways that a Transformer is more general than a simple step function is that it can be stateful and that it defines the `result` post-processing step. Most of the time the user does not need these features, in which case the Transformer is just a wrapper around a step

[4]For simplicity of discussion we will henceforth ignore `init` and initialization in general, as well as some other details.

```
function reduce<A, B>(xf: (B, A) => B           , col: A[])   : B;
function reduce<B>   (xf: (B, string) => B      , col: string): B;
function reduce<A, B>(xf: Transformer<A, B>     , col: A[])   : B;
function reduce<B>   (xf: Transformer<string, B>, string)     : B;
function reduce(xf, col) {
  xf = (typeof xf == "function") ? wrap(xf) : xf;
  if (isString(col)) {
    return stringReduce(xf, col);
  }
  if (isArray(col)) {
    return arrayReduce(xf, col);
  }
}
```

Figure 4.13. Sample Adapted from Transducers Benchmark

function. Thus for convenience, the user is allowed to pass in either a full-fledged Transformer or a step function which will automatically get wrapped into one. Secondly, the collection being reduced can be a stunning array of options: an array, a string (*i.e.* a collection of characters, which are themselves just strings), an arbitrary object (*i.e.*, in JavaScript, a collection of key-value pairs), an iterator (an object that defines a `next` function that iterates through the collection), or an iterable (an object that defines an `iterator` function that returns an iterator). Each of these collections needs to be dispatched to a type-specific reduce function that knows how to iterate over that kind of collection. In each overload, the type of the collection must match the type of the Transformer or step function. Thus our `reduce` begins as shown in Figure 4.13. Considering all five possible types of collections and the option between a step function or a Transformer, `reduce` has ten distinct overloads!

4.4.3 Unhandled Cases

This section outlines and explains some pitfalls of `rsc`.

Complex Constructor Patterns. Due to our limited internal initialization scheme, certain common constructor patterns are not supported by `rsc`. For example, the code in Figure 4.14. Currently, `rsc` does not allow method invocations on the object under construction in the constructor, as it cannot track the (value of the) updates happening in the method `setF`. Note that this case is supported by IGJ. Section (Section 4.5) includes approaches that could lift this restriction.

```
1  class A<M extends RO> {
2    f: nat;
3    constructor() {
4      this.setF(1);
5    }
6    setF(x: number) {
7      this.f = x;
8    }
9  }
```

Figure 4.14. Complex Constructor Pattern Example

```
10 function distinct<T>(a: T[]): T[] {
11   var res: T[] = [];
12   for (var i = 0, n = a.length; i < n; i++) {
13     var current = a[i];
14     for (var j = 0; j < res.length; j++) {
15       if (res[j] === current) break;
16     }
17     if (j === res.length)
18       res.push(current);
19   }
20   return res;
21 }
```

Figure 4.15. Function Computing Distinct Elements of an Array

Recovering Unique References. `rsc` cannot recover the `Unique` state for objects after they have been converted to `Mutable` (or other state), as it lacks a fine-grained alias tracking mechanism. Assume, for example the function `distinct` in Figure 4.15 from the TypeScript compiler v1.0.1.0. Array `res` is defined in line 11 so it is initially typed as `Array<UQ,T>`. At lines 14 – 17 it is iterated over, so to prove the access in line 15 safe, we need to treat `res` as an immutable array. However, in line 18 an element is pushed on `res`, which requires `res` to be mutable. Our system cannot handle the interleaving of these two kinds of operations that (in addition) appear in a tight loop (lines 12 – 20). However, Section 4.5 includes approaches that could allow support for such cases.

Annotations per Function Overload . A weakness of `rsc`, that stems from the use of Two-Phase Typing (Chapter 3) in handling intersection types, is cases where type checking requires annotations under a specific signature overload. Consider for example the code of

```
22 function reduce<A>   (a: A[]+, f: (A, A, idx⟨⟨a⟩⟩) => A): A;
23 function reduce<A, B>(a: A[], f: (B, A, idx⟨⟨a⟩⟩) => B): B;
24 function reduce(a, f, x) {
25   var r, s;
26   if (arguments.length === 3) {
27     r = x;
28     s = 0;
29   } else {
30     r = a[0];
31     s = 1;
32   }
33   for (var i = s; i < a.length; i++)
34     r = f(r, a[i], i);
35   return r;
36 }
```

Figure 4.16. Alternative `reduce` Function

Figure 4.16, which is a variation of the `reduce` function presented in Section 4.1. Checking the function body for the second overload (line 23) is problematic: without an annotation on `r`, its type at the end of the conditional will be `B|(A|undefined)` (`r` collects values from `x` and `a[0]`, at lines 27 and 30), instead of the intended `B`. This causes an error when `r` is passed to function `f` at line 34, expected to have type `B`, which cannot be overcome even with refinement checking, since this code is no longer guarded by the check on the length of `arguments` (line 26). A solution would be for the user to annotate the type of `r` as `B` at its definition in line 25, but only for the specific (second) overload. The assignment in line 30 will be invalid, but this is acceptable since that branch is provably (by the refinement checking phase of Section 3.4) dead. This option, however, is currently not available.

4.5 Related Work

Program Logics for Imperative Programs. Instead of developing a system that segregates base types from refinements like we described in this chapter, one can encode types as formulas in a logic, and use SMT solvers for all the analysis (subtyping). DMinor explores this idea in a first-order functional language with type tests [11]. The idea can be scaled to higher-order languages by embedding (*nesting*) the typing relation inside the logic [20]. DJS combines nested refinements with alias types [95], a restricted separation logic, to account for aliasing and

flow-sensitive heap updates to obtain a static type system for a large portion of JavaScript [20]. DJS proved to be extremely difficult to use. First, the programmer had to spend a lot of effort on manual heap related annotations; a task that became especially cumbersome in the presence of higher-order functions. Second, nested refinements precluded the possibility of refinement inference, further increasing the burden on the user. In contrast, mutability modifiers have proven to be lightweight [116] and two-phase typing lets `rsc` use liquid refinement inference [91], yielding a system that is more practical for real-world programs.

Extended Static Checking [40] uses Floyd-Hoare style first-order contracts (pre-, post-conditions and loop invariants) to generate verification conditions discharged by an SMT solver. Refinement types can be viewed as a generalization of Floyd-Hoare logics that uses types to compositionally account for polymorphic higher-order functions and containers that are ubiquitous in modern languages like TypeScript.

X10 [76] is a language that extends an object-oriented type system with *constraints* on the immutable state of classes. Compared to X10, in `rsc`: (a) we make mutability parametric [116], and extend the refinement system accordingly, (b) we crucially obtain flow-sensitivity via SSA transformation, and path-sensitivity by incorporating branch conditions, (c) we account for reflection by encoding tags in refinements and two-phase typing [108], and (d) our design ensures that we can use liquid type inference [91] to automatically synthesize refinements.

Object and Reference Immutability. `rsc` builds on existing methods for statically enforcing immutability. In particular, we build on Immutability Generic Java which encodes object and reference immutability using Java generics [116]. Subsequent work extends these ideas to allow (1) richer *ownership* patterns for creating immutable cyclic structures [117], (2) *unique* references, and ways to recover immutability after violating uniqueness, without requiring alias analysis [47].

Reference immutability has recently been combined with rely-guarantee logics (originally used to reason about thread interference), to allow refinement type reasoning. Gordon *et al.* [48] treat references to shared objects like threads in rely-guarantee logics, and so multiple aliases to an object are allowed only if the guarantee condition of each alias implies the rely condition for all other aliases. Their approach allows refinement types over mutable data, but resolving their proof obligations depends on theorem-proving, which hinders automation. Militão *et al.* present Rely-Guarantee Protocols [72] that can model complex aliasing interactions, and, compared to

Gordon's work, allow temporary inconsistencies, can recover from shared state via ownership tracking, and resort to more lightweight proving mechanisms.

The above extensions are orthogonal to `rsc`; in the future, it would be interesting to see if they offer practical ways for accounting for (im)mutability in TypeScript programs.

Object Initialization. A key challenge in ensuring immutability is accounting for the construction phase where fields are *initialized*. We limit our attention to *lightweight* approaches *i.e.* those that do not require tracking aliases, capabilities or separation logic [95, 45]. Haack and Poll [51] describe a flexible initialization schema that uses secret tokens, known only to *stack-local* regions, to initialize all members of cyclic structures. Once initialization is complete the tokens are converted to global ones. Their analysis is able to infer the points where new tokens need to be introduced and committed. The *Masked Types* [85] approach tracks, within the type system, the set of fields that remain to be initialized. X10's *hardhat* flow-analysis based approach to initialization [118] and *Freedom Before Commitment* [100] are the most permissive of the lightweight methods, allowing, unlike `rsc`, method dispatches or field accesses in constructors.

Acknowledgements

This chapter contains material adapted from the publication **Refinement types for TypeScript** appearing in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Santa Barbara, CA, USA, June 2016. Vekris, Panagiotis; Cosman, Benjamin; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusions and Future Work

In this dissertation we have presented two main techniques for type checking JavaScript code. In the first one, Flow (Chapter 2) we focus on type reasoning, whereas in the second one, Refined TypeScript (Chapter 4) on value and relational reasoning. In the latter we bridge the gap between type and logic analysis with the novel type checking technique of Two-Phase Typing (Chapter 3). In both works we show that precise static type checking at scale is possible through flow- and path-sensitivity and taking dynamic type tests into account in our analyses. In this chapter we address some of the limitations and explore some future directions in each of these lines of work.

5.1 Flow

Flow's analysis is context-insensitive, and also not well-suited for libraries with reflection lacking type annotations. Types originating from multiple contexts get merged at the boundaries of exported functions, leading to imprecise type inference. Typical remedies for this situation, is to either infer the dynamic type `any` or require type annotations. On the other hand, many libraries provide annotations without checked implementations, so we can type check the vast majority of code that uses these libraries. Better techniques for checking libraries [60] can complement Flow.

Like many other type systems for dynamically typed languages, Flow has the `any` type, with which type checking can be completely bypassed. Unlike gradual type systems, though, there is no runtime enforcement of types when they interact with any. For sound gradual typing, the subtyping rules can be augmented to mark all type constructors as either trusted or untrusted.

Even without `any`, some aspects of JavaScript force us into choosing unsoundness where it is objectively justified. We can lay down the conditions for soundness, but not enforce them. For

example, arrays in JavaScript can have "holes": it is possible to add an element out of bounds, in which case any intermediate positions are filled with `undefined`. Likewise, records in JavaScript can also be accessed as dictionaries, so it is possible to read and write a named property by passing a computed string. Short of complicated numeric and string analysis, soundness would demand that we lose type information on array dereferences and dictionary reads, but this is too restrictive in practice. Instead we hope that developers who care about soundness will not create arrays with holes (*e.g.* by always using `Array.push` to add elements), or will check for `undefined` on dereferences when needed; and the properties that are named and those that are accessed via computed strings are disjoint.

5.2 Refinement Types for TypeScript

Refined TypeScript brings SMT-based modular and extensible analysis to dynamic, imperative, class-based languages by harmoniously integrating several techniques. First, we restrict refinements to immutable variables and fields (cf. X10 [101]). Second, we make mutability parametric (cf. IGJ [116]) and recover path- and flow-sensitivity via SSA. Third, we account for reflection and value overloading via two-phase typing (Chapter 3). Our design ensures that we can use liquid type inference [91] to automatically synthesize refinements. Finally, we have shown how RSC can verify a variety of properties with a modest annotation overhead similar to TypeScript.

Our experience points to several avenues for future work. There is plenty of room for improvement by using a more sophisticated system to establish immutability which is a prerequisite for refinement reasoning. In addition, the initialization scheme supported at this point is fairly limited. Incorporating a more advanced approach would allow more programs to be verified with our tool. Section 4.5 has a more detailed discussion on the techniques that could be integrated with our checker. Note that the modular way in which base type reasoning is segregated from refinement reasoning allows for new approaches to be easily recruited without having to drastically change refinement reasoning as well.

Refined TypeScript assumes that each program term has already been assigned a base type. Assigning these base types still imposes a burden to the developer. Hence employing base type inference in our toolchain would be a reasonable direction. As we saw in Chapter 2, however, this task itself is not trivial, especially when this system needs to be aware of object immutability

invariants. One possible future direction is using Flow as the provider for base types, upon which refinement reasoning will be done later. Recent advances in property variance [1] could be of help in establishing guarantees that would enable refinement reasoning.

Finally, just like Flow the dynamic type `any` is a source of unsoundness. In addition to the issues that were discussed in the previous section, recovering type information from dynamic contexts will now also need to account for immutability guarantees. Recent work by Lehmann and Tanter [68] and Jafery and Dunfield [58] explore the interaction of logical refinements and gradual typing but in a functional setting. It would be interesting to investigate the extensions that would have to be made to support this idea in the setting of a mutable object-oriented language like JavaScript.

[1] https://flow.org/blog/2016/10/04/Property-Variance/

Appendix A

Flow: Precise Type Inference for JavaScript

A.1 Types

In this section we include a discussion on ground types which is the model that the types described in Section 2.3 are based on. We then provide some more context on the notion of polarity that was alluded to during the discussion about constraint propagation (Section 2.4.2). Finally, we define notions related to type subsumption as they are going to be useful for the statement of lemmas and theorems moving forward.

A.1.1 Ground Types

At the basis of the type language described in Section 2.3.2 is the notion of ground types. The formulation of our ground type language follows the one presented by Pottier [83]. Here we will focus on the changes we made to adapt that formulation to our system's needs. Ground types in our system are *regular trees*. The formal definition is similar to Pottier [83, Definition 1.1] but our ground signature Σ_g contains the terminals $\mathfrak{b}$ and $\rightarrow$ for types and the terminals $\perp$ and the set of program variables $\mathcal{X}$ for effects. Also $\rightarrow$ has arity 3 to also account for the function's effect, whose position is co-variant.

Ground Substitutions. We connect the notion of types that were introduced in Section 2.3.2 with ground types using the notion of ground substitutions ρ.

Definition A.1.1 (Ground Substitution)**.** *A* ground substitution ρ *(we will also refer to it as* solution*) is a total mapping from type variables to ground types.*

Ground substitution can be applied to types by recursively applying the substitution the parts of the type replacing type variables with their ground type equivalent.

As regular trees, ground types can be infinite structures, whereas the types we introduced in the main part are finite, but crucially include type variables. This means that a finite, yet recursively defined, type may correspond (through a substitution) to an infinite ground type.

Ground Subtyping. Because of their infinite nature defining a subtyping relation on ground types requires some special treatment. Here, we define an ordering on ground types

by quantifying over paths in the regular trees that represent types. The symbol $\leq_k$ denotes subtyping up to level k. The definition is similar to Pottier [83, Definition 1.5]. For the case of effects, $\leq_0$ is reflexive and $\bot$ is the minimum element. The definition of the subtyping relation $\leq$ over ground types follows Pottier [83, Definition 1.4] and as in Pottier [83, Proposition 1.3], $\tau \leq \tau'$ is equivalent to:

$$\forall k \geq 0 \,.\, \tau \leq_k \tau'$$

Equipping our ground alphabet with $\bot$ and $\top$ for types, and $\top$ for effects (we have omitted them from our formulation to avoid clutter), and using the subtyping relation, our ground types can form a lattice. The proof follows Pottier [83, Proposition 1.3].

Effect and Environment Subtyping. The subtyping relation is extended to ground effects as well (we also refer to them as concrete effects). Concrete effects can be interpreted as sets of variables and so effect subtyping corresponds to the subset relation.

In the following we assume that applying a substitution ρ to an environment Γ of the constraint generation system returns a pair containing two environments:

- a concrete *flow-sensitive environment* Δ binding variables to ground types in a flow-sensitive manner (*i.e.* types that correspond to the base of the entries in Γ), and
- a *general environment* G binding variables to ground types corresponding to the general type of each entry in Γ.

We write this as:

$$\rho\,(\Gamma) = \Delta \wr G$$

We also use indexes as subscripts to retrieve the first or second part of the above pair:

$$\rho\,(\Gamma)_1 = \Delta \qquad\qquad \rho\,(\Gamma)_2 = G$$

The subtyping relation is extended to Δ and G in a point-wise manner.

A.1.2 Constraint Satisfaction

The following definitions relate ground substitutions with constraint sets.

Definition A.1.2 (Constraint Satisfaction)**.** *We say that a ground substitution ρ satisfies a constraint c, and we write $\rho \vdash c$, if the corresponding subtyping relation(s) in the right hand side of the definitions below hold(s):*

$$\begin{array}{lcl}
\rho \vdash \tau \leq \alpha & \doteq & \rho\,(\tau) \leq \rho\,(\alpha) \\
\rho \vdash \tau \leq \mathsf{Call}(\tau') & \doteq & \rho\,(\tau) \leq \rho\,(\tau') \\
\rho \vdash \tau \leq \mathsf{Pred}(P, \tau') & \doteq & \rho\,(\tau :: P) \leq \rho\,(\tau') \\
\rho \vdash \tau \leq \mathsf{Get}\,(f, \tau') & \doteq & \rho\,(\tau) \leq \rho\,(\{f : \tau'\}) \\
\rho \vdash \tau \leq \mathsf{Set}\,(f, \tau') & \doteq & \rho\,(\tau) \leq \rho\,(\{f : \tau'\})
\end{array}$$

$$\begin{array}{lcl} \rho \vdash \epsilon \leq \phi & \doteq & \rho(\epsilon) \leq \rho(\phi) \\ \rho \vdash \epsilon \leq \mathsf{Havoc}(\Gamma) & \doteq & \forall x \in \rho(\epsilon)\ .\ G(x) \leq \Delta(x) \\ & & \textit{where } \rho(\Gamma) = \Delta \fatsemi G \end{array}$$

Definition A.1.3 (Constraint Set Satisfaction under Substitution)**.** *We say that a ground substitution* ρ *satisfies a constraint set* C*, and we write* $\rho \vdash C$*, if for all* c *of* C *it holds that* $\rho \vdash c$*.*

The following proposition connects constraint set consistency that was discussed in Section 2.4.3 with constraint satisfiability under ground substitution defined above.

Proposition A.1.1 (Constraint Set Satisfaction)**.** *A (saturated) constraint set* C *is* satisfiable*, iff there exists ground substitution* ρ *s.t.* $\rho \vdash C$*.*

A.1.3 Polarities

In Section 2.4.2, we introduced Rule CP-P-TRANS that contained the notion of a "positive type hole". To define this formally we first introduce *polar* types, which can be positive or negative. A *positive* type τ^+ is a type used to to describe outputs, whereas a *negative* type τ^- describes inputs. Similar definitions hold for effects (ϵ^+ and ϵ^-). Formally:

$$\begin{array}{lcl} \tau^+ & ::= & b \mid \tau_1^- \xrightarrow{\epsilon^+} \tau_2^+ \mid \{f_1 : \tau_1^+, \ldots, f_n : \tau_n^+\} \mid \alpha \mid \tau_1^+ \sqcup \tau_2^+ \\ \tau^- & ::= & b \mid \tau_1^+ \xrightarrow{\epsilon^-} \tau_2^- \mid \{f_1 : \tau_1^-, \ldots, f_n : \tau_n^-\} \mid \alpha \\ \epsilon^+ & ::= & \bot \mid x \mid \phi \mid \epsilon_1^+ \sqcup \epsilon_2^+ \\ \epsilon^- & ::= & \bot \mid x \mid \phi \end{array}$$

With this in mind we now define a *type context* t as a type that contains a hole $\langle\rangle$ in one of its leafs. Type contexts also come in two flavors:

$$\begin{array}{lcl} t^+ & ::= & b \mid t_1^- \xrightarrow{\epsilon^+} t_2^+ \mid \{f_1 : t_1^+, \ldots, f_n : t_n^+\} \mid \alpha \mid t_1^+ \sqcup t_2^+ \mid \langle\rangle \\ t^- & ::= & b \mid t_1^+ \xrightarrow{\epsilon^-} t_2^- \mid \{f_1 : t_1^-, \ldots, f_n : t_n^-\} \mid \alpha \end{array}$$

The critical part in the above definition is that negative contexts t^- do not contain joins at their top-levels.

A.2 Declarative Type System

In Figures A.1, A.2 and A.3 we define a declarative type system that assigns types to expressions and statements of FLOWCORE. The typing judgments for expressions and statements are:

$$\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta' \qquad\qquad \Delta \fatsemi G \Vdash s : \epsilon \dashv\!\Vdash \Delta'$$

Expression Typing $\boxed{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\vdash \Delta'}$

$$\frac{}{\Delta \fatsemi G \Vdash n : b_n \fatsemi \bot \fatsemi \varnothing \dashv\!\vdash \Delta}\ \text{[T-Const]} \qquad \frac{\Delta(x) = \tau}{\Delta \fatsemi G \Vdash x : \tau \fatsemi \bot \fatsemi x \mapsto \mathsf{truthy} \dashv\!\vdash \Delta}\ \text{[T-Var]}$$

$$\frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\vdash \Delta'}{\Delta \fatsemi G \Vdash x = e : \tau \fatsemi \epsilon \sqcup x \fatsemi \psi \backslash x \dashv\!\vdash \Delta'[x \mapsto \tau]}\ \text{[T-Assign]}$$

$$\frac{\mathsf{erase}_G(\Delta), x : \tau, \mathsf{locals}(s) \fatsemi G \Vdash \{s; \texttt{return}\ e\} : \tau' \fatsemi \epsilon \dashv\!\vdash \Delta'}{\Delta \fatsemi G \Vdash (x) \Rightarrow \{s; \texttt{return}\ e\} : \tau \xrightarrow{\epsilon} \tau' \fatsemi \bot \fatsemi \varnothing \dashv\!\vdash \Delta}\ \text{[T-Fun]}$$

$$\frac{\begin{array}{c}\Delta \fatsemi G \Vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv\!\vdash \Delta_1 \\ \Delta_1 \fatsemi G \Vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\!\vdash \Delta_2 \qquad \tau_1 \leq \tau_2 \xrightarrow{\epsilon} \tau \qquad \Delta' = \mathsf{erase}^{\epsilon}_{G}(\Delta_2)\end{array}}{\Delta \fatsemi G \Vdash e_1(e_2) : \tau \fatsemi \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon \fatsemi \varnothing \dashv\!\vdash \Delta'}\ \text{[T-Call]}$$

Figure A.1. Expression Typing in FlowCore (Variables and Functions)

Here types τ are identical in structure to the types introduced for the inference system but are concrete, *i.e.* there contain no type variables. As mentioned earlier, environments Δ bind variables x to types τ (instead of type entries containing both a precise and a general type). The most general type for each variable is included in environment G– a flow-insensitive structure that gathers the most general type (globally) for each variable across the entire program. Thanks to α-renaming each defined variable to a unique name, there is no ambiguity among variable identifiers.

Effects ϵ are also concrete in this declarative system. This means that they can now be directly interpreted as sets of variables (since no effect variables are presents).

We use the shorthand $\mathsf{erase}_G(\Delta)$ to denote the erasure of an environment Δ with the types of G. This operation effectively creates a new environment binding all variables in Δ to their bound types in G. We also introduce the variant $\mathsf{erase}^{\epsilon}_{G}(\Delta)$, where ϵ is a concrete effect, to denote the environment $\Delta[x \mapsto G(x) \mid x \in \epsilon]$.

Environment join ($\sqcup$) and environment refinement ($::$) have similar definitions as in their constraint generation counterparts of Figure 2.8, and so are omitted here.

A.3 Runtime Typing

Stating a progress and preservation theorem requires us to extend the notion of well-typed expressions and statements to runtime configurations.

Expression Typing $\boxed{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta'}$

$$\frac{\begin{array}{c}\Delta \fatsemi G \Vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv\!\Vdash \Delta_1 \qquad \Delta_1 :: \psi_1 \fatsemi G \Vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\!\Vdash \Delta_2 \\ \tau = \tau_1 :: \mathsf{falsy} \sqcup \tau_2 \qquad \epsilon = \epsilon_1 \sqcup \epsilon_2 \qquad \psi = (\psi_1 \backslash \epsilon_2) \wedge \psi_2 \qquad \Delta' = (\Delta_1 :: \neg\psi_1) \sqcup \Delta_2\end{array}}{\Delta \fatsemi G \Vdash e_1 \,\&\&\, e_2 : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta'}\ \text{[T-And]}$$

$$\frac{\begin{array}{c}\Delta \fatsemi G \Vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv\!\Vdash \Delta_1 \qquad \Delta_1 :: \neg\psi_1 \fatsemi G \Vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\!\Vdash \Delta_2 \\ \tau = \tau_1 :: \mathsf{truthy} \sqcup \tau_2 \qquad \epsilon = \epsilon_1 \sqcup \epsilon_2 \qquad \psi = (\psi_1 \backslash \epsilon_2) \vee \psi_2 \qquad \Delta' = (\Delta_1 :: \psi_1) \sqcup \Delta_2\end{array}}{\Delta \fatsemi G \Vdash e_1 \,||\, e_2 : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta'}\ \text{[T-Or]}$$

$$\frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta'}{\Delta \fatsemi G \Vdash \,!\, e : \texttt{boolean} \fatsemi \epsilon \fatsemi \neg\psi \dashv\!\Vdash \Delta'}\ \text{[T-Not]} \qquad \frac{}{\Delta \fatsemi G \Vdash p(x) : \texttt{boolean} \fatsemi \bot \fatsemi x \mapsto p \dashv\!\Vdash \Delta}\ \text{[T-Pred]}$$

$$\frac{\Delta \equiv \Delta_0 \qquad \forall i \in [1, n] \,.\, \Delta_{i-1} \fatsemi G \Vdash e_i : \tau_i \fatsemi \epsilon_i \fatsemi \psi_i \dashv\!\Vdash \Delta_i \qquad \forall i \in [1, n] \,.\, \tau_i \leq \tau'_i}{\Delta \fatsemi G \Vdash \{f_1 : e_1, \ldots, f_n : e_n\} : \{f_1 : \tau'_1, \ldots, f_n : \tau'_n\} \fatsemi \bigsqcup \epsilon_i \fatsemi \varnothing \dashv\!\Vdash \Delta_n}\ \text{[T-Rec]}$$

$$\frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta' \qquad \tau \leq \{f : \tau'\}}{\Delta \fatsemi G \Vdash e.f : \tau' \fatsemi \epsilon \fatsemi \varnothing \dashv\!\Vdash \Delta'}\ \text{[T-FldRd]}$$

$$\frac{\begin{array}{c}\Delta \fatsemi G \Vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv\!\Vdash \Delta_1 \\ \tau_1 \leq \{f : \tau_f\} \qquad \Delta_1 \fatsemi G \Vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\!\Vdash \Delta_2 \qquad \tau_2 \leq \tau_f\end{array}}{\Delta \fatsemi G \Vdash e_1.f = e_2 : \tau_2 \fatsemi \epsilon_1 \sqcup \epsilon_2 \fatsemi \psi_2 \dashv\!\Vdash \Delta_2}\ \text{[T-FldWr]}$$

Figure A.2. Expression Typing in FlowCore (Logical Operators and Records)

A.3.1 Term Typing

Expressions & Statements. First we extend typing to runtime expressions. The judgment form is similar to the one for static expressions with the difference that we have to include locations ℓ in the set of typeable expressions. To do that we equip our judgment with an additional argument, the *heap typing* Σ, defined as:

$$\Sigma \;::=\; \cdot \mid \Sigma, \ell : \tau$$

The expression typing judgment becomes:

$$\Delta \fatsemi G \Vdash_\Sigma e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta'$$

Statement Typing $\boxed{\Delta \fatsemi G \Vdash s : \epsilon \dashv\!\shortmid \Delta'}$

$$\frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta'}{\Delta \fatsemi G \Vdash e : \epsilon \dashv\!\shortmid \Delta'}\ [\text{T-Exp}] \qquad \frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta'}{\Delta \fatsemi G \Vdash \texttt{var}\ x = e : \epsilon \sqcup x \dashv\!\shortmid \Delta'[x \mapsto \tau]}\ [\text{T-VarDecl}]$$

$$\frac{\Delta \fatsemi G \Vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta' \qquad \Delta' :: \psi \fatsemi G \Vdash s_1 : \epsilon_1 \dashv\!\shortmid \Delta_1' \qquad \Delta' :: \neg\psi \fatsemi G \Vdash s_2 : \epsilon_2 \dashv\!\shortmid \Delta_2'}{\Delta \fatsemi G \Vdash \texttt{if}\,(e)\,\{s_1\}\,\texttt{else}\,\{s_2\} : \epsilon \sqcup \epsilon_1 \sqcup \epsilon_2 \dashv\!\shortmid \Delta_1' \sqcup \Delta_2'}\ [\text{T-If}]$$

$$\frac{\Delta \fatsemi G \Vdash s_1 : \epsilon_1 \dashv\!\shortmid \Delta_1 \qquad \Delta_1 \fatsemi G \Vdash s_2 : \epsilon_2 \dashv\!\shortmid \Delta_2}{\Delta \fatsemi G \Vdash s_1\,;\, s_2 : \epsilon_1 \sqcup \epsilon_2 \dashv\!\shortmid \Delta_2}\ [\text{T-Seq}]$$

Figure A.3. Statement Typing in FlowCore

Extending the rules for expression typing in Figures A.1 and A.2 to runtime expressions is straightforward. An important addition is the rule for location ℓ typing:

$$\frac{\Sigma(\ell) = \tau}{\Delta \fatsemi G \Vdash_\Sigma \ell : \tau \fatsemi \bot \fatsemi \varnothing \dashv\!\shortmid \Delta}\ [\text{T-Loc}]$$

Similarly the form of typing runtime statements is extended to:

$$\Delta \fatsemi G \Vdash_\Sigma s : \epsilon \dashv\!\shortmid \Delta'$$

Evaluation Contexts. A more interesting situation arises when we try to extend the judgment to evaluation contexts E. The main issue here is that the object under judgment contains a "hole" where another expression is expected to appear. To address this we include a "hole" in the type structure of the return type to host the type of the term that is expected to fill in the hole of the evaluation context. The linked effect and predicate are handled in a similar fashion:

$$\Delta \fatsemi G \Vdash_\Sigma E : \tau'\langle\tau\rangle \fatsemi \epsilon'\langle\epsilon\rangle \fatsemi \psi'\langle\psi\rangle \dashv\!\shortmid \Delta'$$

Figure A.4 contains a selection of rules for this judgment.

A natural extension of the definition of evaluation context typing is the following lemma that accounts for substituting an expression in the hole of an evaluation context.

Lemma A.1 (Evaluation Context Typing)**.** *If*

(i) $\Delta \fatsemi G \Vdash_\Sigma e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta'$

(ii) $\Delta' \fatsemi G \Vdash_\Sigma E : \tau'\langle\tau\rangle \fatsemi \epsilon'\langle\epsilon\rangle \fatsemi \psi'\langle\psi\rangle \dashv\!\shortmid \Delta''$

Evaluation Context Typing Rules (selected) $\boxed{\Delta \fatsemi G \Vdash_\Sigma E : \tau'\langle\tau\rangle \fatsemi \epsilon'\langle\epsilon\rangle \fatsemi \psi'\langle\psi\rangle \dashv\!\Vdash \Delta'}$

$$\frac{}{\Delta \fatsemi G \Vdash_\Sigma \langle\rangle : \langle\rangle \fatsemi \langle\rangle \fatsemi \langle\rangle \dashv\!\Vdash \Delta}\ [\textsc{Ectx-Hole}]$$

$$\frac{\begin{array}{c}\Delta \fatsemi G \Vdash_\Sigma E : \tau_1'\langle\tau_1\rangle \fatsemi \epsilon_1'\langle\epsilon_1\rangle \fatsemi \psi_1'\langle\psi_1\rangle \dashv\!\Vdash \Delta_1 \\ \Delta_1 \fatsemi G \Vdash e : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\!\Vdash \Delta_2 \qquad \tau_1' \le \tau_2 \xrightarrow{\epsilon} \tau \qquad \Delta' = \mathsf{erase}^{\epsilon}_{G}(\Delta_2)\end{array}}{\Delta \fatsemi G \Vdash_\Sigma E(e) : \tau\langle\tau_1\rangle \fatsemi \epsilon_1'\langle\epsilon_1\rangle \sqcup \epsilon_2 \sqcup \epsilon \fatsemi \varnothing\langle\psi_1\rangle \dashv\!\Vdash \Delta'}\ [\textsc{Ectx-Call}]$$

Figure A.4. Evaluation Context Typing in FLOWCORE

then

$$\Delta \fatsemi G \Vdash_\Sigma E\langle e\rangle : \tau' \fatsemi \epsilon' \fatsemi \psi' \dashv\!\Vdash \Delta''$$

Proof. By induction on the second given derivation. □

When inverting typing relations, we often need to decompose the typing of filled evaluation contexts $E\langle e\rangle$. The following lemma deconstructs the typing of such an expression to the typing of a bare evaluation context E and a typing of the filling expression e.

Lemma A.2 (Decomposing Evaluation Context Typing)**.** *If*

$$\Delta \fatsemi G \Vdash_\Sigma E\langle e\rangle : \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta''$$

then there exist τ', ϵ', ψ' *and* Δ' *s.t.*

(a) $\Delta \fatsemi G \Vdash_\Sigma e : \tau' \fatsemi \epsilon' \fatsemi \psi' \dashv\!\Vdash \Delta'$

(b) $\Delta \fatsemi G \Vdash_\Sigma E : \tau\langle\tau'\rangle \fatsemi \epsilon\langle\epsilon'\rangle \fatsemi \psi\langle\psi'\rangle \dashv\!\Vdash \Delta''$

Proof. By examining all possible cases of typing evaluation contexts E*, we will always type the expression* e *in the "hole" first and then the evaluation context* E*.* □

A.3.2 Configuration Typing

A runtime configuration in FLOWCORE contains the runtime state, that itself comprises a heap H, a stack X and a store L, and a program term. Typing configurations amounts to typing their subparts. Before we move on to that we define two auxiliary functions.

Auxiliary Functions. The first one is the *environment composition* $M \circ N$. This operation works in the usual way. The range of environment N needs to be compatible with the domain of M, otherwise the result is undefined:

$$(M \circ N)(x) = \begin{cases} M(N(x)) & \text{if } X \in \mathsf{dom}(N) \text{ and } N(x) \in \mathsf{dom}(M) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Stack Typing Rules $\boxed{G \Vdash_\Sigma X\colon \tau'\langle\tau\rangle}$

$$\frac{}{G \Vdash_\Sigma \cdot\colon \tau\langle\tau\rangle}\ [\text{RT-STACK-E}]$$

$$\frac{\Delta = \Sigma \circ L \qquad \Delta \fatsemi G \Vdash E\colon \tau'\langle\tau\rangle \dashv\Vdash \Delta' \qquad \Sigma' = \Delta' \circ L^{-1} \oplus \Sigma \qquad G \Vdash_{\Sigma'} X\colon \tau''\langle\tau'\rangle}{G \Vdash_\Sigma X, L.E\colon \tau''\langle\tau\rangle}\ [\text{RT-STACK-C}]$$

Figure A.5. Runtime Stack Typing in FLOWCORE

The second operator is the *environment override* $M \oplus N$. This operator produces an environment whose domain is the union of the domains of the two arguments. For each one of its arguments the override first attempts to return a binding by looking it up in environment M; if this fails it tries N; and finally returns undefined if it fails there as well.

$$(M \oplus N)(x) = \begin{cases} M(x) & \text{if } x \in \mathsf{dom}(M) \\ N(x) & \text{if } x \in \mathsf{dom}(N) \setminus \mathsf{dom}(M) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Stack. The form of the stack is reminiscent of the evaluation context, so the judgment we use here has the following form:

$$G \Vdash_\Sigma X\colon \tau\langle\tau'\rangle$$

Figure A.5 contains the rules for this judgment. The interesting rule here is Rule RT-STACK-C, that types a stack $X, L.E$. Following the flow of execution the rule first checks the frame E that is on the top of the stack and then proceeds with the remaining stack X. What is interesting here is the construction of the environment used for checking X. Assume Δ' the output environment after checking E. This environment contains the most recent updates of all the variables that were assigned to in E. Our goal here is to construct an accurate heap typing Σ' that corresponds to the state of the heap at the end of E. This heap typing will subsequently be used to check X. To do that, for every variable x such that $x\colon \ell \in L$, *i.e.* in scope at the beginning of E, we require its type to be looked up in Δ'. This amounts to $\Delta' \circ L^{-1}$. The rest will just be looked up in the incoming Σ.

Heap. Figure A.6 shows the rules for checking a heap H against a heap typing Σ. The most interesting case here is that of record typing by Rule RT-HEAP-REC. This rule infers a type for each value v_i stored at some field of the record and then unifies this type with the type of each field specified in the store typing Σ.

Configuration. Finally, Figure A.7 shows the typing rules for runtime configurations where the terms are either expressions, function bodies or statements. These largely follow the same principles as the typing for stacks that we saw earlier.

Heap Typing Rules $\boxed{G \Vdash_\Sigma H}$

$$\frac{}{G \Vdash_\Sigma \cdot}\ [\text{RT-HEAP-E}] \qquad \frac{\ell' \in \mathsf{dom}(H) \qquad G \Vdash_\Sigma H \qquad \Sigma(\ell) = \Sigma(\ell')}{G \Vdash_\Sigma H, \ell \mapsto \ell'}\ [\text{RT-HEAP-LOC}]$$

$$\frac{G \Vdash_\Sigma H \qquad \Sigma(\ell) = b_n}{G \Vdash_\Sigma H, \ell \mapsto n}\ [\text{RT-HEAP-CONST}]$$

$$\frac{\begin{array}{c} G \Vdash_\Sigma H \\ \Sigma(\ell) = \tau \qquad \Delta = \Sigma \circ L \qquad \Delta \fatsemi G \Vdash (x) \Rightarrow \{s; \texttt{return}\ e\} : \tau \fatsemi \bot \fatsemi \varnothing \dashv\vdash \Delta \end{array}}{G \Vdash_\Sigma H, \ell \mapsto \langle L, (x) \Rightarrow \{s; \texttt{return}\ e\}\rangle}\ [\text{RT-HEAP-FUN}]$$

$$\frac{\begin{array}{c} G \Vdash_\Sigma H \qquad \Sigma(\ell) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \\ \forall i \in [1, n]\,.\, \Delta \fatsemi G \Vdash_\Sigma v_i : \tau_i' \fatsemi \bot \fatsemi \varnothing \dashv\vdash \Delta \qquad \forall i \in [1, n]\,.\, \tau_i' \leq \tau_i \end{array}}{G \Vdash_\Sigma H, \ell \mapsto \{f_1 : v_1, \ldots, f_n : v_n\}}\ [\text{RT-HEAP-REC}]$$

Figure A.6. Heap Typing in FLOWCORE

Runtime Configuration Typing $\boxed{G \Vdash_\Sigma S; M : \tau}$ $\boxed{G \Vdash_\Sigma S; s}$

$$\frac{\begin{array}{c} G \Vdash_\Sigma H \\ \Delta = \Sigma \circ L \qquad \Delta \fatsemi G \Vdash_\Sigma M : \tau \fatsemi \epsilon \dashv\vdash \Delta' \qquad \Sigma' = \Delta' \circ L^{-1} \oplus \Sigma \qquad G \Vdash_{\Sigma'} X : \tau'\langle\tau\rangle \end{array}}{G \Vdash_\Sigma \langle H; X; L\rangle; M : \tau'}\ [\text{RT-CONF-B}]$$

$$\frac{\begin{array}{c} G \Vdash_\Sigma H \\ \Delta = \Sigma \circ L \qquad \Delta \fatsemi G \Vdash_\Sigma s : \epsilon \dashv\vdash \Delta' \qquad \Sigma' = \Delta' \circ L^{-1} \oplus \Sigma \qquad G \Vdash_{\Sigma'} X : \tau'\langle\tau\rangle \end{array}}{G \Vdash_\Sigma \langle H; X; L\rangle; s}\ [\text{RT-CONF-S}]$$

Figure A.7. Runtime Configuration Typing in FLOWCORE

A.4 Proofs

This section contains a statement and proof of soundness of the inference type system of Section 2.4 with respect to the declarative system of Section A.2, followed by our type safety result for the declarative system and by extension the entire type system.

A.4.1 Type Inference Soundness

The following lemma captures the intuition behind the "havoc" mechanism, as the erasure of the part of the widened environment that is affected by the reaching effect.

Lemma A.3 (Havoc)**.** *If*

(i) $\mathsf{widen}(\Gamma) = \Gamma' \triangleright C$

(ii) $C' \supseteq C \cup \{\, \phi \leq \mathsf{Havoc}\,(\Gamma')\,\}$

(iii) $\rho \vdash C'$

then

$$\Delta' = \mathsf{erase}_G^{\rho(\phi)}\,(\Delta)$$

where $\rho\,(\Gamma) = \Delta \fatsemi G$ *and* $\rho\,(\Gamma') = \Delta' \fatsemi G$.

Proof. Let $\rho\,(\phi) = \epsilon$. For every variable $x \in \epsilon$, it also holds that $x \leq \mathsf{Havoc}\,(\Gamma') \in C'$, since C' is saturated. Let $\Gamma'(x) = \tau^{\alpha}$. By Rule CP-HAVOC on the binding for x, it holds that $\alpha \leq \tau \in C'$. Due to (iii), $\rho\,(\alpha) \leq \rho\,(\tau)$. Which is also written as $G(x) \leq \Delta(x)$. But by definition of G it holds that $\Delta(x) \leq G(x)$, so it must be that $\Delta(x) = G(x)$. Generalizing for all variables in $\rho\,(\phi)$ we prove the wanted. □

Lemma A.4 (Type Inference Soundness)**.** *If*

(i) $\Gamma \vdash e\colon \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C$

(ii) $\rho \vdash C$

then

$$\rho\,(\Gamma) \Vdash e\colon \rho\,(\tau) \fatsemi \rho\,(\epsilon) \fatsemi \psi \dashv\!\vert\, \rho\,(\Gamma')_1$$

Proof. By induction on the derivation of (i):

- CG-CALL:

$$\Gamma \vdash e_1\,(e_2)\colon \alpha \fatsemi \epsilon \fatsemi \varnothing \dashv \Gamma_3 \triangleright C \tag{A.4.1}$$

By inverting Rule CG-CALL on (A.4.1):

$$\Gamma \vdash e_1\colon \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv \Gamma_1 \triangleright C_1 \tag{A.4.2}$$

$$\Gamma_1 \vdash e_2\colon \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv \Gamma_2 \triangleright C_2 \tag{A.4.3}$$

$$\mathsf{widen}(\Gamma_2) = \Gamma_3 \triangleright C_w \tag{A.4.4}$$

$$\epsilon_1 \sqcup \epsilon_2 \sqcup \phi = \epsilon \tag{A.4.5}$$

$$C_1 \cup C_2 \cup C_w \cup \left\{\, \phi \leq \mathsf{Havoc}\,(\Gamma_3)\,,\ \tau_1 \leq \mathsf{Call}(\tau_2 \xrightarrow{\phi} \alpha) \,\right\} = C \tag{A.4.6}$$

where α, ϕ fresh.

Since by (A.4.6) it is $C \supseteq C_1$ and $C \supseteq C_2$, using (ii) it holds that:

$$\rho \vdash C_1 \tag{A.4.7}$$
$$\rho \vdash C_2 \tag{A.4.8}$$

By induction hypothesis using (A.4.2), (A.4.7), (A.4.3) and (A.4.8):

$$\rho(\Gamma) \Vdash e_1 : \rho(\tau_1) \fatsemi \rho(\epsilon_1) \fatsemi \psi_1 \dashv\Vdash \rho(\Gamma_1)_1 \tag{A.4.9}$$
$$\rho(\Gamma_1) \Vdash e_2 : \rho(\tau_2) \fatsemi \rho(\epsilon_2) \fatsemi \psi_2 \dashv\Vdash \rho(\Gamma_2)_1 \tag{A.4.10}$$

By (A.4.6), using Definition A.1.2:

$$\rho(\tau_1) \leq \rho\left(\tau_2 \xrightarrow{\phi} \alpha\right) \equiv \rho(\tau_2) \xrightarrow{\rho(\phi)} \rho(\alpha) \tag{A.4.11}$$

By Lemma A.3 on (A.4.4), (A.4.6) and (ii):

$$\Delta_3 = \mathsf{erase}_G^{\rho(\phi)}(\Delta_2) \tag{A.4.12}$$

where $\rho(\Gamma_2) = \Delta_2 \fatsemi G$ and $\rho(\Gamma_3) = \Delta_3 \fatsemi G$.

By Rule T-CALL on (A.4.9), (A.4.10), (A.4.11) and (A.4.12)

$$\rho(\Gamma) \Vdash e_1(e_2) : \rho(\alpha) \fatsemi \rho(\epsilon_1) \sqcup \rho(\epsilon_2) \sqcup \rho(\phi) \fatsemi \varnothing \dashv\Vdash \rho(\Gamma_3)_1 \tag{A.4.13}$$
$$\therefore \quad \rho(\Gamma) \Vdash e_1(e_2) : \rho(\alpha) \fatsemi \rho(\epsilon_1 \sqcup \epsilon_2 \sqcup \phi) \fatsemi \varnothing \dashv\Vdash \rho(\Gamma_3)_1 \tag{A.4.14}$$

- CG-ASSIGN:

$$\Gamma \vdash x = e : \tau \fatsemi \epsilon \sqcup x \fatsemi \psi \backslash x \dashv \Gamma'[x \mapsto \tau^{\alpha}] \triangleright C \tag{A.4.15}$$

By inverting Rule CG-ASSIGN on (A.4.15):

$$\Gamma \vdash e : \tau \fatsemi \epsilon \fatsemi \psi \dashv \Gamma' \triangleright C_0 \tag{A.4.16}$$
$$\Gamma'(x) = \tau_0{}^{\alpha} \tag{A.4.17}$$
$$C = C_0 \cup \{\tau \leq \alpha\} \tag{A.4.18}$$

Since by (A.4.18) it is $C \supseteq C_0$, using (ii) it holds that:

$$\rho \vdash C_0 \tag{A.4.19}$$

By induction hypothesis using (A.4.16) and (A.4.19):

$$\rho\,(\Gamma) \Vdash e\colon \rho\,(\tau) \fatsemi \rho\,(\epsilon) \fatsemi \psi \dashv\!\shortmid \rho\left(\Gamma'\right)_1 \tag{A.4.20}$$

By applying Rule T-ASSIGN on (A.4.20):

$$\rho\,(\Gamma) \Vdash x = e\colon \rho\,(\tau) \fatsemi \rho\,(\epsilon) \sqcup x \fatsemi \psi\backslash x \dashv\!\shortmid \rho\left(\Gamma'\right)[x \mapsto \rho\,(\tau)] \tag{A.4.21}$$

$$\therefore \quad \rho\,(\Gamma) \Vdash x = e\colon \rho\,(\tau) \fatsemi \rho\,(\epsilon \sqcup x) \fatsemi \psi\backslash x \dashv\!\shortmid \rho\left(\Gamma'[x \mapsto \tau^{\alpha}]\right)_1 \tag{A.4.22}$$

The rest of the cases are handled similarly.

□

A.4.2 Type Safety

In this section we present the proofs of our safety result that connects the declarative type system of Section A.2 with the runtime semantics of Section 2.5. First we set up a number of auxiliary lemmas and then proceed with a Preservation Theorem (A.11) and a Progress Theorem (A.12) that are later combined to produce a Type Safety Theorem (A.14).

Lemma A.5 (Erased Environment Subtyping)**.** *If* $\mathsf{erase}^{\epsilon}_{G}\,(\Delta) = \Delta'$*, then* $\Delta \leq \Delta'$*.*

Proof. By definition of the erase *operator.* □

In the remaining we use the metavariable M to denote a term that is either an expression e or a function body `{s; return e}`.

Lemma A.6 (Heap Typing Weakening)**.** *For* $\Sigma' \supseteq \Sigma$*, if* $\Delta \fatsemi G \Vdash_{\Sigma} M\colon \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta'$*, then* $\Delta \fatsemi G \Vdash_{\Sigma'} M\colon \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta'$*.*

Proof. By induction on the given derivation. □

Lemma A.7 (Environment Weakening)**.** *For the following, let environments* Δ *and* Δ' *be defined over common domains.*

I. If $\Delta \fatsemi G \Vdash e\colon \tau \fatsemi \epsilon \fatsemi \psi \dashv\!\shortmid \Delta_1$ *then for* $\Delta' \leq \Delta$*:*

(a) $\Delta' \fatsemi G \Vdash e\colon \tau' \fatsemi \epsilon' \fatsemi \psi' \dashv\!\shortmid \Delta'_1$*,*

(b) $\tau' \leq \tau$*,* $\epsilon' \leq \epsilon$ *and* $\Delta'_1 \leq \Delta_1$*.*

II. If $\Delta \fatsemi G \Vdash E\colon \tau\langle\tau_1\rangle \fatsemi \epsilon\langle\epsilon_1\rangle \fatsemi \psi\langle\psi_1\rangle \dashv\!\shortmid \Delta_1$ *then for* $\Delta' \leq \Delta$*,* $\tau'_1 \leq \tau_1$ *and* $\epsilon'_1 \leq \epsilon_1$*:*

(a) $\Delta' \fatsemi G \Vdash E\colon \tau'\langle\tau'_1\rangle \fatsemi \epsilon'\langle\epsilon'_1\rangle \fatsemi \psi'\langle\psi'_1\rangle \dashv\!\shortmid \Delta'_1$

(b) $\tau' \leq \tau$*,* $\epsilon' \leq \epsilon$ *and* $\Delta'_1 \leq \Delta_1$*.*

Proof. By induction on the given derivation. □

Lemma A.8 (Heap Typing Weakening)**.** *If*

(i) $\Sigma' \leq \Sigma$

(ii) $\tau_1' \leq \tau_1$

(iii) $G \Vdash_\Sigma X \colon \tau\langle\tau_1\rangle$

then

(a) $G \Vdash_{\Sigma'} X \colon \tau'\langle\tau_1'\rangle$

(b) $\tau' \leq \tau$

Proof. By induction on the derivation (iii). □

Lemma A.9 (NonEffect)**.** *If*

$$\Delta \fatsemi G \Vdash e \colon \tau \fatsemi \epsilon \fatsemi \psi \dashv\Vdash \Delta'$$

then

$$\Delta' \big|_{\overline{\epsilon}} \leq \Delta \big|_{\overline{\epsilon}}$$

where $\overline{\epsilon}$ *is the set of program variables that do not belong to the concrete effect* ϵ*.*

Proof. By induction on the given derivation:

- T-VAR, T-CONST, T-FUN and T-PRED: It holds that

$$\Delta' \equiv \Delta \tag{A.9.1}$$

  so the wanted result holds trivially.

- T-ASSIGN:

$$\Delta \fatsemi G \Vdash \underbrace{x = e_0}_{e} \colon \tau \fatsemi \epsilon_0 \sqcup x \fatsemi \psi_0 \backslash x \dashv\Vdash \underbrace{\Delta_0[x_0 \mapsto \tau]}_{\Delta'} \tag{A.9.2}$$

  By inverting T-ASSIGN on (A.9.2):

$$\Delta \fatsemi G \Vdash e_0 \colon \tau \fatsemi \epsilon_0 \fatsemi \psi_0 \dashv\Vdash \Delta_0 \tag{A.9.3}$$

  By (A.9.2) for a variable y *s.t.* $y \notin \epsilon$, it also holds that:

$$y \neq x \tag{A.9.4}$$

$$y \notin \epsilon_0 \tag{A.9.5}$$

By induction hypothesis using (A.9.3) and (A.9.5):

$$\Delta_0(y) \leq \Delta(y) \tag{A.9.6}$$

By (A.9.4) it holds that $\Delta_0(y) = \Delta'(y)$, and so by (A.9.6):

$$\Delta'(y) \leq \Delta(y) \tag{A.9.7}$$

- T-CALL:

$$\Delta \fatsemi G \Vdash e_1(e_2) : \tau \fatsemi \underbrace{\epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_c}_{\epsilon} \fatsemi \varnothing \dashv\Vdash \Delta' \tag{A.9.8}$$

By inverting T-CALL on (A.9.8)

$$\Delta \fatsemi G \Vdash e_1 : \tau_1 \fatsemi \epsilon_1 \fatsemi \psi_1 \dashv\Vdash \Delta_1 \tag{A.9.9}$$
$$\Delta_1 \fatsemi G \Vdash e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\Vdash \Delta_2 \tag{A.9.10}$$
$$\tau_1 \leq \tau_2 \xrightarrow{\epsilon_c} \tau \tag{A.9.11}$$
$$\Delta' = \mathsf{erase}^{\epsilon_c}_{G}(\Delta_2) \tag{A.9.12}$$

For a variable $x \in \mathsf{dom}(\Delta')$ *s.t.* $x \notin \epsilon$, it also holds that:

$$x \notin \epsilon_1 \tag{A.9.13}$$
$$x \notin \epsilon_2 \tag{A.9.14}$$
$$x \notin \epsilon_c \tag{A.9.15}$$

By induction hypothesis on (A.9.9) and (A.9.13), and (A.9.10) and (A.9.14):

$$\Delta_1(x) \leq \Delta(x) \tag{A.9.16}$$
$$\Delta_2(x) \leq \Delta_1(x) \tag{A.9.17}$$

By definition of the erase operator on (A.9.12) for x *s.t.* (A.9.15):

$$\Delta'(x) = \Delta_2(x) \tag{A.9.18}$$

By (A.9.16), (A.9.17) and (A.9.18):

$$\Delta'(x) \leq \Delta(x) \tag{A.9.19}$$

- T-AND, T-OR, T-NOT, T-PRED, T-REC, T-FLDRD and T-FLDWR: *Similar to above.*

□

Lemma A.10 (Preservation of Typing by Expression Reduction)**.** *Typing is preserved over the reduction of an expression that preserves the state of the stack. That is, for an initial runtime state* $S \doteq \langle H, X, L \rangle$*, a target state* $S' \doteq \langle H', X, L' \rangle$ *if, under a heap typing* Σ*:*

(i) $G \Vdash_\Sigma H$

(ii) $\Delta \fatsemi G \Vdash_\Sigma e : \tau \fatsemi \epsilon \fatsemi \psi \dashv\Vdash \Delta_1$

(iii) $S;\, e \longrightarrow S';\, e'$

where $\Delta \doteq \Sigma \circ L$*, then there exist* Σ' *s.t.:*

(a) $G \Vdash_{\Sigma'} H'$

(b) $\Delta' \fatsemi G \Vdash_{\Sigma'} e' : \tau' \fatsemi \epsilon' \fatsemi \psi' \dashv\Vdash \Delta'_1$

(c) $\tau' \leq \tau$

(d) $\epsilon' \leq \epsilon$

(e) $\Delta'_1 \leq \Delta_1$

where $\Delta' \doteq \Sigma' \circ L'$*.*

Proof. By induction on the derivation of (ii):

- RT-PRED-VAR:

$$\langle H, X, L \rangle;\, \underbrace{p(x)}_{e} \longrightarrow \langle H, X, L \rangle;\, v \tag{A.10.1}$$

By Rule T-PRED, (ii) is of the form:

$$\Delta \fatsemi G \Vdash_\Sigma p(x) : \texttt{boolean} \fatsemi \bot \fatsemi \underbrace{x \mapsto p}_{\psi} \dashv\Vdash \Delta \tag{A.10.2}$$

We pick $\Sigma' \doteq \Sigma$.

Store and heap do not evolve, *i.e.* $L' = L$ and $H' = H$. So, by (i):

$$G \Vdash_{\Sigma'} H' \tag{A.10.3}$$

which proves (a).

By definition of Δ and Δ', it holds that $\Delta' = \Delta$.

By applying Rule T-CONST on v (`true` or `false`)

$$\Delta' \fatsemi G \Vdash_{\Sigma'} v : \texttt{boolean} \fatsemi \bot \fatsemi \varnothing \dashv\!\mid \Delta' \tag{A.10.4}$$

which proves (b), (c), (d) and (e).

- RT-ASGN with $v = n$:

$$\langle H, X, L\rangle; \underbrace{x = n}_{e} \longrightarrow \langle H', X, L\rangle; n \tag{A.10.5}$$

By inverting Rule RT-ASGN on (A.10.5):

$$H' = H[\, L(x) \mapsto n \,] \tag{A.10.6}$$

By Rule T-ASSIGN, (ii) is of the form:

$$\Delta \fatsemi G \Vdash_{\Sigma} x = n : b_n \fatsemi \epsilon \sqcup x \fatsemi \psi \backslash x \dashv\!\mid \Delta[\, x \mapsto b_n \,] \tag{A.10.7}$$

By inverting Rule T-ASSIGN on (A.10.7):

$$\Delta \fatsemi G \Vdash_{\Sigma} n : b_n \fatsemi \bot \fatsemi \varnothing \dashv\!\mid \Delta \tag{A.10.8}$$

So, $\epsilon = \bot$ and $\psi = \varnothing$.

Let ℓ *s.t.* $x \mapsto \ell \in L$.

We pick $\Sigma' \doteq \Sigma[\, \ell \mapsto b_n \,]$. By T-CONST on n under Δ':

$$\Delta' \fatsemi G \Vdash_{\Sigma'} n : b_n \fatsemi \bot \fatsemi \varnothing \dashv\!\mid \Delta' \tag{A.10.9}$$

Let Σ_0 and H_0 *s.t.* $\Sigma = \Sigma_0, \ell : \tau_\ell$ and $H = H_0, \ell \mapsto n$.

It holds that:

$$\Sigma' = \Sigma_0, \ell : b_n \tag{A.10.10}$$

By applying Rule RT-HEAP-CONST on (i) (on the part of H_0) and (A.10.10):

$$G \Vdash_{\Sigma'} H_0, \ell \mapsto n \tag{A.10.11}$$

which proves (a).

By (A.10.8) we prove (b), (c) and (d).

Since Δ' and Δ agree on all variables with the exception potentially of x, we limit the scope to x. By definition of Δ' it holds that:

$$\begin{aligned}\Delta'(x) &= (\Sigma' \circ L)(x)\\ &= \Sigma'(L(x))\\ &= \Sigma'(\ell)\\ &= b_n\\ &= \Delta(x)\end{aligned}$$

- RT-ECTX:

$$\langle H, X, L\rangle; \underbrace{E\langle e_0\rangle}_{e} \longrightarrow \langle H', X, L'\rangle; E\langle e_0'\rangle \tag{A.10.12}$$

By inverting Rule RT-ECTX on (A.10.12):

$$\langle H, X, L\rangle; e_0 \longrightarrow \langle H', X, L'\rangle; e_0' \tag{A.10.13}$$

By Lemma A.2 on (ii) for $e \equiv E\langle e_0\rangle$:

$$\Delta \fatsemi G \Vdash_\Sigma e_0 : \tau_0 \fatsemi \epsilon_0 \fatsemi \psi_0 \dashv\Vdash \Delta_0 \tag{A.10.14}$$

$$\Delta_0 \fatsemi G \Vdash E : \tau\langle\tau_0\rangle \fatsemi \epsilon\langle\epsilon_0\rangle \fatsemi \psi\langle\psi_0\rangle \dashv\Vdash \Delta_1 \tag{A.10.15}$$

By induction hypothesis using (i), (A.10.14) and (A.10.13) there exists Σ' *s.t.*:

$$G \Vdash_{\Sigma'} H' \tag{A.10.16}$$

$$\Delta' \fatsemi G \Vdash_{\Sigma'} e_0' : \tau_0' \fatsemi \epsilon_0' \fatsemi \psi_0' \dashv\Vdash \Delta_1' \tag{A.10.17}$$

$$\tau_0' \leq \tau_0 \tag{A.10.18}$$

$$\epsilon_0' \leq \epsilon_0 \tag{A.10.19}$$

$$\Delta_0' \leq \Delta_0 \tag{A.10.20}$$

By (A.10.16) we prove (a).

By Lemma A.7.II on (A.10.15), (A.10.20), (A.10.18) and (A.10.19):

$$\Delta_0' \fatsemi G \Vdash E : \tau'\langle\tau_0'\rangle \fatsemi \epsilon'\langle\epsilon_0'\rangle \fatsemi \psi'\langle\psi_0\rangle \dashv\Vdash \Delta_1' \tag{A.10.21}$$

$$\tau' \leq \tau \tag{A.10.22}$$

$$\epsilon' \leq \epsilon \tag{A.10.23}$$

$$\Delta_1' \leq \Delta_1 \tag{A.10.24}$$

By Lemma A.1 on (A.10.17) and (A.10.21):

$$\Delta' \fatsemi G \Vdash_{\Sigma'} E\langle e_0' \rangle : \tau' \fatsemi \epsilon' \fatsemi \psi' \dashv\Vdash \Delta_1' \tag{A.10.25}$$

By (A.10.25), (A.10.22), (A.10.23) and (A.10.24) we prove (b), (c), (d) and (e), respectively.

- RT-AND-TRU:

$$\langle H, X, L \rangle; \underbrace{v_1 \mathbin{\&\&} e_2}_{e} \longrightarrow \langle H, X, L \rangle; e_2 \tag{A.10.26}$$

By inverting Rule RT-AND-TRU on (A.10.26):

$$\mathsf{truthy}(v_1) \tag{A.10.27}$$

Due to (A.10.26) judgment (ii) is of the form:

$$\Delta \fatsemi G \Vdash_{\Sigma} v_1 \mathbin{\&\&} e_2 : \underbrace{\tau_1 :: \mathsf{falsy} \sqcup \tau_2}_{\tau} \fatsemi \underbrace{\epsilon_1 \sqcup \epsilon_2}_{\epsilon} \fatsemi \underbrace{(\psi_1 \backslash \epsilon_1) \wedge \psi_2}_{\psi} \dashv\Vdash \underbrace{\Delta_1 :: \neg\psi_1 \sqcup \Delta_2}_{\Delta_1} \tag{A.10.28}$$

By inverting Rule T-AND on (ii) and simplifying by using Rules RT-T-LOC and T-CONST:

$$\Delta \fatsemi G \Vdash_{\Sigma} v_1 : \tau_1 \fatsemi \bot \fatsemi \varnothing \dashv\Vdash \Delta \tag{A.10.29}$$

$$\Delta \fatsemi G \Vdash_{\Sigma} e_2 : \tau_2 \fatsemi \epsilon_2 \fatsemi \psi_2 \dashv\Vdash \Delta_2 \tag{A.10.30}$$

So, $\epsilon_1 = \bot$ and $\psi_1 = \varnothing$.

Store and heap do not evolve, *i.e.* $L' = L$ and $H' = H$.

We pick $\Sigma' \doteq \Sigma$ and so $\Delta' = \Delta$. So, by (i):

$$G \Vdash_{\Sigma'} H' \tag{A.10.31}$$

which proves (a).

By (A.10.30) we prove (b).

It holds that $\tau_2 \leq (\tau_1 :: \mathsf{falsy} \sqcup \tau_2) \overset{(A.10.28)}{\equiv} \tau$, which proves (c).

It holds that $\epsilon_2 \leq \epsilon_1 \sqcup \epsilon_2 \overset{(A.10.28)}{\equiv} \epsilon$, which proves (d).

Finally, it holds that $\Delta_2 \leq (\Delta_1 :: \neg\varnothing \sqcup \Delta_2) \overset{(A.10.28)}{\equiv} \Delta_1$, which proves (e).

The rest of the cases are handled similarly. □

Theorem A.11 (Subject Reduction)**.** *Typing is preserved over expression reduction. Formally, if*

(i) $G \Vdash_\Sigma S; e : \tau$

(ii) $S;\ e \longrightarrow S';\ M'$

then there exists Σ' *s.t.*

(a) $G \Vdash_{\Sigma'} S'; M' : \tau'$

(b) $\tau' \leq \tau$

Proof. Let

$$S \equiv \langle H;\ X;\ L \rangle \tag{A.11.1}$$

$$S' \equiv \langle H';\ X';\ L' \rangle \tag{A.11.2}$$

By inverting Rule RT-CONF-B on (i):

$$G \Vdash_\Sigma H \tag{A.11.3}$$

$$\Delta = \Sigma \circ L \tag{A.11.4}$$

$$\Delta \fatsemi G \Vdash_\Sigma e : \tau_e \fatsemi \epsilon_e \dashv\Vdash \Delta_e \tag{A.11.5}$$

$$\Sigma_X = \Delta_e \circ L^{-1} \oplus \Sigma \tag{A.11.6}$$

$$G \Vdash_{\Sigma_X} X : \tau\langle\tau_e\rangle \tag{A.11.7}$$

By induction on the derivation of (ii):

- RT-ASGN, RT-ARROW, RT-PRED-VAR, RT-AND-TRU, RT-AND-FLS, RT-OR-TRU, RT-OR-FLS, RT-NEG, RT-LET, RT-IF-TRU, RT-IF-FLS, and RT-SKIP do not evolve the stack, so can be proven by use of Lemma A.10.
- RT-CALL:

$$\langle H;\ X;\ L \rangle;\ \underbrace{E\langle \ell(v) \rangle}_{e} \longrightarrow \langle H';\ X';\ L' \rangle;\ \underbrace{\{s_0\,;\ \texttt{return}\ e_0\}}_{M'} \tag{A.11.8}$$

where ℓ' fresh. By inverting Rule RT-CALL on (A.11.8):

$$H(\ell) = \langle L_0,\ (x) \Rightarrow M' \rangle \tag{A.11.9}$$

$$\overline{x_i} = \mathsf{locals}(M') \tag{A.11.10}$$

$$H' = H,\ \ell' \mapsto v,\ \overline{\ell_i \mapsto \texttt{undefined}} \tag{A.11.11}$$

$$X' = X,\ L.E \tag{A.11.12}$$

$$L' = L_0,\ x \mapsto \ell',\ \overline{x_i \mapsto \ell_i} \tag{A.11.13}$$

Due to (A.11.8), judgment (A.11.5) is of the form:

$$\Delta \fatsemi G \Vdash_\Sigma E\langle \ell(v) \rangle : \tau_e \fatsemi \epsilon_e \dashv \Delta_e \tag{A.11.14}$$

By Lemma A.2 on (A.11.14):

$$\Delta \fatsemi G \Vdash_\Sigma \ell(v) : \tau_c \fatsemi \epsilon_c \fatsemi \psi_c \dashv \Delta_3 \tag{A.11.15}$$

$$\Delta_3 \fatsemi G \Vdash_\Sigma E : \tau_e\langle \tau_c \rangle \fatsemi \epsilon_e\langle \epsilon_c \rangle \fatsemi \psi\langle \psi_c \rangle \dashv \Delta_e \tag{A.11.16}$$

By inverting Rule T-CALL on (A.11.15):

$$\Delta \fatsemi G \Vdash_\Sigma \ell : \tau_\ell \fatsemi \bot \fatsemi \varnothing \dashv \Delta_1 \tag{A.11.17}$$

$$\Delta_1 \fatsemi G \Vdash_\Sigma v : \tau_v \fatsemi \bot \fatsemi \varnothing \dashv \Delta_2 \tag{A.11.18}$$

$$\tau_\ell \leq \tau_v \xrightarrow{\epsilon_c} \tau_c \tag{A.11.19}$$

$$\Delta_3 = \mathsf{erase}^{\epsilon_c}_G\ (\Delta_2) \tag{A.11.20}$$

By Rules RT-T-LOC and T-CONST on (A.11.17) and on (A.11.18):

$$\Delta = \Delta_1 = \Delta_2 \tag{A.11.21}$$

So (A.11.18) becomes:

$$\Delta \fatsemi G \Vdash_\Sigma v : \tau_v \fatsemi \bot \fatsemi \varnothing \dashv \Delta \tag{A.11.22}$$

By inverting Rule RT-HEAP-FUN on (A.11.3) using (A.11.9):

$$G \Vdash_\Sigma H_0 \tag{A.11.23}$$

$$\Sigma(\ell) = \tau_\ell \tag{A.11.24}$$

$$\Delta_0 = \Sigma \circ L_0 \tag{A.11.25}$$

$$\Delta_0 \fatsemi G \Vdash (x) \Rightarrow \{s_0\,;\, \texttt{return}\ e_0\} : \tau_\ell \fatsemi \bot \fatsemi \varnothing \dashv \Delta_0' \tag{A.11.26}$$

where

$$H \equiv H_0, \ell \mapsto \langle L_0, (x) \Rightarrow M' \rangle \tag{A.11.27}$$

By inverting Rule T-FUN on (A.11.26):

$$\underbrace{\mathsf{erase}_G(\Delta_0),\, x\colon \tau_x,\, \overline{x_i\colon \texttt{void}}}_{\Delta_{0.1}} \fatsemi G \Vdash_\Sigma \{s_0\,;\, \texttt{return}\ e_0\}\colon \tau_0 \fatsemi \epsilon_0 \dashv\!\mid \Delta_{0.2} \tag{A.11.28}$$

$$\tau_\ell \equiv \tau_x \xrightarrow{\epsilon_0} \tau_0 \tag{A.11.29}$$

where $\overline{x_i}$ are the local variables defined in s_0.

By (A.11.19) and (A.11.29):

$$\tau_x \xrightarrow{\epsilon_0} \tau_0 \leq \tau_v \xrightarrow{\epsilon_c} \tau_c \tag{A.11.30}$$

By subtyping decomposition on (A.11.30):

$$\tau_v \leq \tau_x \tag{A.11.31}$$

$$\tau_0 \leq \tau_c \tag{A.11.32}$$

$$\epsilon_0 \leq \epsilon_c \tag{A.11.33}$$

After the reduction step, we pick:

$$\Sigma' = \Sigma,\, \ell'\colon \tau_v,\, \overline{\ell_i\colon \texttt{void}} \tag{A.11.34}$$

The body $M' = \{s_0\,;\, \texttt{return}\ e_0\}$ is checked under the environment produced by store L_0 and heap typing Σ'. Σ' coincides with Σ on their common domain $\mathsf{dom}(L)$, so:

$$\Delta_0 = \Sigma' \circ L_0 = \Sigma \circ L_0 \tag{A.11.35}$$

We extend the store L_0 with a binding from x to ℓ', and from every variable declared in the body s_0 to `void`, resulting in the following environment:

$$\Delta' = \Delta_0,\, x\colon \tau_v,\, \overline{x_i\colon \texttt{void}} = \Sigma' \circ \underbrace{L_0,\, x\colon \ell',\, \overline{x_i\colon \ell_i}}_{L'} \tag{A.11.36}$$

By Lemma A.6 on (A.11.22):

$$\Delta \fatsemi G \Vdash_{\Sigma'} v\colon \tau_v \fatsemi \bot \fatsemi \varnothing \dashv\!\mid \Delta \tag{A.11.37}$$

By applying Rules RT-HEAP-LOC and RT-HEAP-CONST using (A.11.3), (A.11.34) and (A.11.37):

$$G \Vdash_{\Sigma'} H' \tag{A.11.38}$$

By definition of erase and (A.11.31):

$$\underbrace{\Delta_0,\, x\colon \tau_v,\, \overline{x_i\colon \texttt{void}}}_{\Delta'} \leq \underbrace{\mathsf{erase}_G(\Delta_0),\, x\colon \tau_x,\, \overline{x_i\colon \texttt{void}}}_{\Delta_{0.1}} \tag{A.11.39}$$

By Lemma A.7.I on (A.11.28) and (A.11.39), and using the extended heap typing Σ':

$$\Delta' \fatsemi G \Vdash_{\Sigma'} \{s_0\,;\, \texttt{return}\ e_0\}\colon \tau_0' \fatsemi \epsilon_0' \dashv\!\mid \Delta_2' \tag{A.11.40}$$

$$\tau_0' \leq \tau_0 \tag{A.11.41}$$

$$\Delta_2' \leq \Delta_{0.2} \tag{A.11.42}$$

$$\epsilon_0' \leq \epsilon_0 \tag{A.11.43}$$

Stack $X' = X$, L.E is checked under a heap typing:

$$\Sigma'_{X,\,L.E} \doteq \Delta_2' \circ L_0^{-1} \oplus \Sigma' \tag{A.11.44}$$

Evaluation context E is checked under an environment:

$$\Delta_3' = \Sigma'_{X,\,L.E} \circ L \tag{A.11.45}$$

Let $\mathcal{X}$ and $\mathcal{X}_0$ be the domains of L and L_0:

$$\mathcal{X} \doteq \mathsf{dom}(L) \tag{A.11.46}$$

$$\mathcal{X}_0 \doteq \mathsf{dom}(L_0) \tag{A.11.47}$$

Since $\mathsf{dom}(\Delta_3) = \mathcal{X}$, we can examine Δ_3 in two parts based on whether an element x in $\mathcal{X}$, also belongs to $\mathcal{X}_0$ or not:

$$\Delta_3 \equiv \Delta_3 \big|_{\mathcal{X}_0 \cap \mathcal{X}},\, \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0} \tag{A.11.48}$$

We similarly examine Δ_3' into two parts: (i) the closure environment Δ_2' at the end of the function body, and (ii) the part of the environment at the call-site that is not part of the closure environment and so retains the typing from before the function call:

$$\Delta_3' \equiv \Delta_2' \big|_{\mathcal{X}_0 \cap \mathcal{X}},\, \Delta \big|_{\mathcal{X} \setminus \mathcal{X}_0} \tag{A.11.49}$$

We examine the two non-overlapping domains separately:

♦ $\mathcal{X}_0 \cap \mathcal{X}$. By restricting (A.11.36) to $\mathcal{X}_0 \cap \mathcal{X}$:

$$\Delta' \big|_{\mathcal{X}_0 \cap \mathcal{X}} = \Delta_0,\, x\colon \tau_v,\, \overline{x_i\colon \texttt{void}}\Big|_{\mathcal{X}_0 \cap \mathcal{X}} \qquad = \Sigma' \circ \left(L_0,\, x\colon \ell',\, \overline{x_i\colon \ell_i}\Big|_{\mathcal{X}_0 \cap \mathcal{X}} \right) \tag{A.11.50}$$

$$\therefore \quad \Delta' \big|_{\mathcal{X}_0 \cap \mathcal{X}} = \Delta_0 \big|_{\mathcal{X}_0 \cap \mathcal{X}} \qquad = \Sigma' \circ L_0 \big|_{\mathcal{X}_0 \cap \mathcal{X}} \tag{A.11.51}$$

Note that due to α-renaming every variable is uniquely defined. Therefore, each variable is bound to the same location in a store that contains it. In particular, for L_0 and L it holds that:

$$L_0 \big|_{\mathcal{X}_0 \cap \mathcal{X}} = L \big|_{\mathcal{X}_0 \cap \mathcal{X}} \tag{A.11.52}$$

By restricting (A.11.4) to $\mathcal{X}_0 \cap \mathcal{X}$:

$$\Delta \big|_{\mathcal{X}_0 \cap \mathcal{X}} = \Sigma \circ L \big|_{\mathcal{X}_0 \cap \mathcal{X}} \tag{A.11.53}$$

By combining (A.11.21), (A.11.51), (A.11.52) and (A.11.53):

$$\Delta' \big|_{\mathcal{X}_0 \cap \mathcal{X}} = \Delta_2 \big|_{\mathcal{X}_0 \cap \mathcal{X}} \tag{A.11.54}$$

Effect ϵ_c is concrete so it can be interpreted as a set of variables. We split the set $\mathcal{X}_0 \cap \mathcal{X}$ in the following:

$$\mathcal{X}_0 \cap \mathcal{X} = \underbrace{\mathcal{X}_0 \cap \mathcal{X} \cap \epsilon_c}_{\mathcal{X}_\epsilon},\, \underbrace{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c}_{\mathcal{X}_{\overline{\epsilon}}} \tag{A.11.55}$$

We examine each part separately.

► $\mathcal{X}_\epsilon$. We first restrict (A.11.20) to domain ϵ_c (a concrete effect interpreted as a set):

$$\Delta_3 \big|_{\epsilon_c} = \mathsf{erase}_G^{\epsilon_c}\,(\Delta_2) \,\big|_{\epsilon_c} \tag{A.11.56}$$

By definition of erase, (A.11.56) can be written as:

$$\Delta_3 \big|_{\epsilon_c} = G \big|_{\epsilon_c} \tag{A.11.57}$$

By definition of G it holds that:

$$\Delta_2' \leq G \tag{A.11.58}$$

By (A.11.57) and (A.11.58):

$$\Delta_2' \big|_{\mathcal{X}_\epsilon} \leq \Delta_3 \big|_{\mathcal{X}_\epsilon} \tag{A.11.59}$$

- $\mathcal{X}_{\overline{\epsilon}}$. By definition of erase:

$$\mathsf{erase}_G^{\epsilon_c}(\Delta_2) \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} = \Delta_2 \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \tag{A.11.60}$$

since the binding for variables not in ϵ_c will not be affected by the erasure.
By (A.11.20) and (A.11.60):

$$\Delta_2 \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} = \Delta_3 \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \tag{A.11.61}$$

By (A.11.33) and (A.11.43) (interpreting concrete effects as sets):

$$\epsilon_0' \subseteq \epsilon_c \tag{A.11.62}$$

By Lemma A.9 on (A.11.40):

$$\Delta_2' \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_0'} \leq \Delta' \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_0'} \tag{A.11.63}$$

By (A.11.62) and (A.11.63):

$$\Delta_2' \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \leq \Delta' \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \tag{A.11.64}$$

By (A.11.64) and (A.11.54):

$$\Delta_2' \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \leq \Delta_2 \big|_{(\mathcal{X}_0 \cap \mathcal{X}) \setminus \epsilon_c} \tag{A.11.65}$$

By definition of $\mathcal{X}_{\overline{\epsilon}}$, (A.11.65) can be written as

$$\Delta_2' \big|_{\mathcal{X}_{\overline{\epsilon}}} \leq \Delta_2 \big|_{\mathcal{X}_{\overline{\epsilon}}} \tag{A.11.66}$$

- $\mathcal{X} \setminus \mathcal{X}_0$. We follow a similar reasoning to above restricting the difference $\mathcal{X} \setminus \mathcal{X}_0$ to variables contained in ϵ_c or not. We examine the cases:
  - Restrict to ϵ_c. By (A.11.20):

$$\Delta_3 \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \cap \epsilon_c} = \mathsf{erase}_G^{\epsilon_c}(\Delta_2) \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \cap \epsilon_c} \tag{A.11.67}$$

By definition of erase the above becomes:

$$\Delta_3 \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \cap \epsilon_c} = \Delta_2 \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \cap \epsilon_c} \tag{A.11.68}$$

▶ Restrict to $\overline{\epsilon_c}$. By (A.11.20):

$$\Delta_3 \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \setminus \epsilon_c} = \mathsf{erase}_G^{\epsilon_c}(\Delta_2) \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \setminus \epsilon_c} \tag{A.11.69}$$

By definition of erase the above becomes:

$$\Delta_3 \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \setminus \epsilon_c} = G \big|_{(\mathcal{X} \setminus \mathcal{X}_0) \setminus \epsilon_c} \tag{A.11.70}$$

In either case it holds that:

$$\Delta \big|_{\mathcal{X} \setminus \mathcal{X}_0} \leq \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0} \tag{A.11.71}$$

By (A.11.48) it holds that:

$$\left(\Delta_3 \big|_{\mathcal{X}_\epsilon}, \Delta_3 \big|_{\mathcal{X}_{\overline{\epsilon}}}, \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0}\right) \equiv \Delta_3 \tag{A.11.72}$$

By (A.11.59) and (A.11.72):

$$\left(\Delta_2' \big|_{\mathcal{X}_\epsilon}, \Delta_3 \big|_{\mathcal{X}_{\overline{\epsilon}}}, \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0}\right) \leq \Delta_3 \tag{A.11.73}$$

By Lemma A.5 on (A.11.73) and (A.11.19):

$$\left(\Delta_2' \big|_{\mathcal{X}_\epsilon}, \Delta_2 \big|_{\mathcal{X}_{\overline{\epsilon}}}, \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0}\right) \leq \Delta_3 \tag{A.11.74}$$

By (A.11.74) and (A.11.66):

$$\left(\Delta_2' \big|_{\mathcal{X}_\epsilon}, \Delta_2' \big|_{\mathcal{X}_{\overline{\epsilon}}}, \Delta_3 \big|_{\mathcal{X} \setminus \mathcal{X}_0}\right) \leq \Delta_3 \tag{A.11.75}$$

By (A.11.71) and (A.11.75):

$$\underbrace{\left(\Delta_2' \big|_{\mathcal{X}_\epsilon}, \Delta_2' \big|_{\mathcal{X}_{\overline{\epsilon}}}, \Delta \big|_{\mathcal{X} \setminus \mathcal{X}_0}\right)}_{\Delta_3'} \leq \Delta_3 \tag{A.11.76}$$

By Lemma A.7.II on (A.11.76) and (A.11.16):

$$\Delta_3' \fatsemi G \Vdash_\Sigma E : \tau_e' \langle \tau_c \rangle \fatsemi \epsilon_e' \langle \epsilon_c \rangle \fatsemi \psi' \langle \psi_c \rangle \dashv\!\shortmid \Delta_e' \tag{A.11.77}$$

$$\tau_e' \leq \tau_e \tag{A.11.78}$$

$$\epsilon_e' \leq \epsilon_e \tag{A.11.79}$$

$$\Delta_e' \leq \Delta_e \tag{A.11.80}$$

We check the remaining stack X under a heap typing (see also (A.11.44)):

$$\Sigma'_X \doteq \Delta'_e \circ L^{-1} \oplus \underbrace{\Delta'_2 \circ L_0^{-1} \oplus \Sigma'}_{\Sigma'_{X,\,L.E}} \tag{A.11.81}$$

Let $\mathcal{L}$ and $\mathcal{L}_0$ the ranges of L and L_0:

$$\mathcal{L} \doteq \mathsf{rng}\,(L) \tag{A.11.82}$$

$$\mathcal{L}_0 \doteq \mathsf{rng}\,(L_0) \tag{A.11.83}$$

We examine Σ'_X in the following subdomains that correspond to the three parts of the definition above:

- $\mathcal{L}$. By restricting (A.11.6) and (A.11.81), respectively, to $\mathcal{L}$ (the first part of the override will always be selected):

$$\Sigma_X\big|_{\mathcal{L}} = \Delta_e \circ L^{-1} \tag{A.11.84}$$

$$\Sigma'_X\big|_{\mathcal{L}} = \Delta'_e \circ L^{-1} \tag{A.11.85}$$

  By substituting (A.11.84) and (A.11.85) in (A.11.80):

$$\Sigma'_X\big|_{\mathcal{L}} \leq \Sigma_X\big|_{\mathcal{L}} \tag{A.11.86}$$

- $\mathcal{L}_0 \setminus \mathcal{L}$. By (A.11.6) and (A.11.25):

$$\Sigma_X\big|_{\mathcal{L}_0 \setminus \mathcal{L}} \overset{(A.11.6)}{=} \Sigma\big|_{\mathcal{L}_0 \setminus \mathcal{L}} \overset{(A.11.25)}{=} \Delta_0 \circ (L_0 \setminus L)^{-1} \tag{A.11.87}$$

  By restricting (A.11.81) to $\mathcal{L}_0 \setminus \mathcal{L}$ (the second part of the override will always be selected):

$$\Sigma'_X\big|_{\mathcal{L}_0 \setminus \mathcal{L}} = \Delta'_2 \circ (L_0 \setminus L)^{-1} \tag{A.11.88}$$

  By (A.11.65), (A.11.87) and (A.11.88):

$$\Sigma'_X\big|_{\mathcal{L}_0 \setminus \mathcal{L}} \leq \Sigma_X\big|_{\mathcal{L}_0 \setminus \mathcal{L}} \tag{A.11.89}$$

- $\overline{(\mathcal{L}_0 \cup \mathcal{L})}$. By (A.11.6) and (A.11.44):

$$\Sigma_X\big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} = \Sigma\big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} \tag{A.11.90}$$

$$\Sigma_X\big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} = \Sigma'\big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} \tag{A.11.91}$$

By (A.11.90), (A.11.91) and (A.11.34):

$$\Sigma'_X \big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} \leq \Sigma_X \big|_{\overline{(\mathcal{L}_0 \cup \mathcal{L})}} \tag{A.11.92}$$

By composing (A.11.86), (A.11.89) and (A.11.92):

$$\Sigma'_X \leq \Sigma_X \tag{A.11.93}$$

By Lemma A.8 on (A.11.93), (A.11.78) and (A.11.7):

$$G \Vdash_{\Sigma'_X} X\colon \tau'\langle\tau'_e\rangle \tag{A.11.94}$$

$$\tau' \leq \tau \tag{A.11.95}$$

By Rule RT-STACK-C on (A.11.45), (A.11.77), (A.11.81) and (A.11.94) we get the typing for $X' = X, L.E$:

$$G \Vdash_{\Sigma'_{X,L.E}} X, L.E\colon \tau'\langle\tau_c\rangle \tag{A.11.96}$$

By applying Rule RT-CONF-B on (A.11.38), (A.11.36), (A.11.40) and (A.11.96):

$$G \Vdash_{\Sigma'} \langle H'; X'; L'\rangle; \{s_0\,;\, \texttt{return}\ e_0\}\colon \tau' \tag{A.11.97}$$

which proves (a).

By (A.11.95) we prove (b).

- RT-RET

  This case is treated similarly.

□

Theorem A.12 (Progress – Expressions and Function Bodies)**.** *If*

$$G \Vdash_\Sigma S; M\colon \tau$$

then one of the following holds:

(a) M *is a value*

(b) there exist S' *and* M' *s.t.* $S; M \longrightarrow S'; M'$.

Proof. Let

$$S \equiv \langle H; X; L\rangle \tag{A.12.1}$$

We prove the desired by induction on the given derivation.

- RT-CONF-B:

$$G \Vdash_\Sigma S;\, e\colon \tau \tag{A.12.2}$$

By inverting Rule RT-CONF-E on (A.12.2):

$$G \Vdash_\Sigma H \tag{A.12.3}$$
$$\Delta = \Sigma \circ L \tag{A.12.4}$$
$$\Delta \fatsemi G \Vdash_\Sigma e\colon \tau_e \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta' \tag{A.12.5}$$
$$\Sigma' = \Delta' \circ L^{-1} \oplus \Sigma \tag{A.12.6}$$
$$G \Vdash_{\Sigma'} X\colon \tau\langle\alpha_X\rangle \tag{A.12.7}$$

By induction on the derivation of (A.12.5):

- T-CONST: This expression is already a value so (a) holds.
- T-CALL:

$$\Delta \fatsemi G \Vdash_\Sigma \underbrace{\ell(v)}_{e}\colon \tau_e \fatsemi \epsilon \fatsemi \psi \dashv\!\Vdash \Delta' \tag{A.12.8}$$

By inverting Rule T-CALL on (A.12.8):

$$\Delta \fatsemi G \Vdash_\Sigma \ell\colon \tau_\ell \fatsemi \bot \fatsemi \varnothing \dashv\!\Vdash \Delta_1 \tag{A.12.9}$$
$$\Delta_1 \fatsemi G \Vdash_\Sigma v\colon \tau_v \fatsemi \bot \fatsemi \varnothing \dashv\!\Vdash \Delta_2 \tag{A.12.10}$$
$$\tau_\ell \leq \tau_v \xrightarrow{\epsilon} \tau_e \tag{A.12.11}$$
$$\Delta' = \mathsf{erase}^{\epsilon}_{G}\,(\Delta_2) \tag{A.12.12}$$

By inverting Rule RT-T-LOC on (A.12.9):

$$\Delta_1 = \Delta \tag{A.12.13}$$
$$\Sigma(\ell) = \tau_\ell \tag{A.12.14}$$

By Rule T-CONST (or Rule RT-T-LOC) on (A.12.10):

$$\Delta_2 = \Delta \tag{A.12.15}$$

By (A.12.3) for location ℓ:

$$G \Vdash_\Sigma H_0,\, \ell \mapsto \dot{v} \tag{A.12.16}$$

For some heap H_0 and heap value $\dot{v}$.

Next we prove that $H(\ell) = \langle L_0, (x) \Rightarrow \{s_0\,;\, \texttt{return}\ e_0\}\rangle$ for some L_0, s_0 and e_0 by induction on the derivation of (A.12.16):

- RT-HEAP-LOC:

$$G \Vdash_\Sigma H_0,\ \ell \mapsto \ell' \tag{A.12.17}$$

  For some location ℓ' distinct from ℓ.
  By inverting RT-HEAP-LOC on (A.12.17):

$$G \Vdash_\Sigma H_0 \tag{A.12.18}$$

  Let $H_0 = H_0',\ \ell' \mapsto \dot{v}'$. (A.12.18) becomes:

$$G \Vdash_\Sigma H_0',\ \ell' \mapsto \dot{v}' \tag{A.12.19}$$

  By induction hypothesis using (A.12.19):

$$H_0(\ell') = \langle L_0', (x) \Rightarrow \{s_0'\,;\, \texttt{return}\ e_0'\}\rangle \tag{A.12.20}$$

- RT-HEAP-CONST:

$$G \Vdash_\Sigma H_0,\ \ell \mapsto n \tag{A.12.21}$$

  For some constant n. By inverting RT-HEAP-CONST on (A.12.21):

$$G \Vdash_\Sigma H_0 \tag{A.12.22}$$
$$\Sigma(\ell) = b_n \tag{A.12.23}$$

  The subtyping constraint (A.12.11) and (A.12.23) lead to a contradiction.

- RT-HEAP-FUN:

$$G \Vdash_\Sigma H_0,\ \ell \mapsto \langle L_0, (x) \Rightarrow \{s_0\,;\, \texttt{return}\ e_0\}\rangle \tag{A.12.24}$$

  which proves the desired result immediately.

- RT-HEAP-REC: *Similar to rule* RT-HEAP-CONST.

So there exist L_0, s_0 and e_0 *s.t.*:

$$H(\ell) = \langle L_0, (x) \Rightarrow \{s_0\,;\, \texttt{return}\ e_0\}\rangle \tag{A.12.25}$$

We pick:

$$H' = H,\ \ell_v \mapsto v,\ \overline{\ell_i \mapsto \texttt{undefined}} \tag{A.12.26}$$

$$X' = X,\ L.\langle\,\rangle \tag{A.12.27}$$

$$L' = L_0,\ x \mapsto \ell_v,\ \overline{x_i \mapsto \ell_i} \tag{A.12.28}$$

where $\overline{x_i}$ are the variables defined in the function body, and ℓ_v and ℓ_i are fresh locations.

By applying Rule RT-CALL using (A.12.25), (A.12.26), (A.12.27) and (A.12.28)

$$S;\ \ell(v) \longrightarrow \langle H';\ X';\ L'\rangle;\ \{s_0\,;\ \texttt{return}\ e_0\} \tag{A.12.29}$$

which proves (b).

- T-VAR, T-ASSIGN, T-FUN, T-AND, T-OR, T-NOT and T-PRED are straight-forward since they only impose very minimal preconditions for the respective transition to happen.

- RT-CONF-S Proved by applying Theorem A.13 on the statement part of the body.

□

Theorem A.13 (Progress – Statements)**.** *If*

(i) $\vdash_\Sigma S;\ s \triangleright C$

(ii) C *is consistent*

then one of the following holds:

(a) s *is a irreducible form*

(b) there exists S' *and* s' *s.t.* $S;\ s \longrightarrow S';\ s'$

Proof. The proof is by induction on the derivation of (i).

□

Theorem A.14 (Type Safety)**.** *A well-typed program is either in normal form or reduces to another well typed state.*

Proof. The proof follows by subsequent applications of Theorems A.12, A.13 and A.11. □

Appendix B

Trust, but Verify: Two-Phase Typing for Dynamic Languages

We now provide detailed versions of the proofs mentioned in Chapter 3. This part reuses the definitions of Sections 3.2, 3.3 and 3.4, and is structured in three main sections:

- Assumptions (Section B.1)
- Lemmas (Section B.2)
- Theorems (Section B.3)

Sections B.1 and B.2 build up to the main results:

- Consistency and Reverse Consistency Theorems (3.4, 3.5)
- Two-phase Safety Theorem (B.14)

For the remainder of the document we are going to use the plain version of the elaboration relation, *i.e.* without mode annotations:

$$\Gamma \vdash e : \tau \hookrightarrow w$$

The annotations on the judgment merely determine which rules are available at type checking. The majority of the proofs below involve induction over the elaboration derivation, which is fixed once type checking is complete, so the annotations can be safely ignored.

In certain lemmas the reader is referred to Dunfield's techniques from his work on the elaboration of intersection and union types [31]. The proofs there refer to a language similar but not exactly the same as ours. The main proof ideas, however, hold.

B.1 Assumptions

Assumption B.1.1 (Primitive Constant Application)**.** *If*

(i) $\cdot \vdash n : \tau \to \tau' \hookrightarrow n$,

(ii) $\cdot \vdash v : \tau \hookrightarrow v$,

(iii) $v \not\equiv \mathsf{dead}\downarrow^{\cdot}_{\tau}(\cdot)$,

then

(a) $n(v) \longrightarrow [\![n]\!](v)$

(b) $n(v) \longrightarrow [\![n]\!](v)$

(c) $\cdot \vdash [\![n]\!](v) : \tau' \hookrightarrow [\![n]\!](v)$

Assumption B.1.2 (Lambda Application)**.** *If*

(i) $\cdot \vdash (x) \Rightarrow e : \tau \to \tau' \hookrightarrow (x) \Rightarrow w$,

(ii) $\cdot \vdash v : \tau \hookrightarrow v$,

(iii) $v \not\equiv \mathsf{dead}\downarrow^{\cdot}_{\tau}(\cdot)$,

then

(a) $((x) \Rightarrow e)(v) \longrightarrow [v/x](e)$

(b) $((x) \Rightarrow w)(v) \longrightarrow [v/x](w)$

Assumption B.1.3 (Canonical Forms)**.**

(I) If $\Gamma \vdash (x) \Rightarrow e : \tau \to \tau' \hookrightarrow v$ *then*

- *(a)* $v \equiv (x) \Rightarrow w$ *for some* w*, or*
- *(b)* $v \equiv \mathsf{dead}\downarrow^{\cdot}_{\tau \to \tau'}(v')$ *for some* v'

(II) If $\Gamma \vdash n : \tau \hookrightarrow v$ *then*

- *(a)* $v \equiv n$*, or*
- *(b)* $v \equiv \mathsf{dead}\downarrow^{\cdot}_{\tau}(v')$ *for some* v'

B.2 Auxiliary lemmas

Lemma B.1 (Multi-Step Source Evaluation Context)**.** *If* $e \longrightarrow^* e'$ *then* $\mathsf{E}\langle e \rangle \longrightarrow^* \mathsf{E}\langle e' \rangle$.

Proof. Based on Dunfield [31, Lemma 7]. □

Lemma B.2 (Multi-Step Target Evaluation Context)**.**

I. If $w \longrightarrow^* w'$ *then* $\mathcal{E}\langle w \rangle \longrightarrow^* \mathcal{E}\langle w' \rangle$.

II. If $w \longrightarrow^{+} w'$ *then* $\mathcal{E}\langle w \rangle \longrightarrow^{+} \mathcal{E}\langle w' \rangle$.

Proof. Similar to proof of Lemma B.1. □

Lemma B.3 (Unions/Injections). *If* $\Gamma \vdash e : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_k\ w$ *then* $\Gamma \vdash e : \tau_k \hookrightarrow w$.

Proof. Based on Dunfield [31, Lemma 8]. □

Lemma B.4 (Intersections/Pairs). *If* $\Gamma \vdash e : \tau_1 \wedge \tau_2 \hookrightarrow w_1, w_2$ *then there exist* e_1' *and* e_2' *such that:*

(a) $e_1 \longrightarrow^{*} e_1'$ *and* $\Gamma \vdash e_1' : \tau_1 \hookrightarrow w_1$

(b) $e_2 \longrightarrow^{*} e_2'$ *and* $\Gamma \vdash e_2' : \tau_2 \hookrightarrow w_2$

Proof. Based on Dunfield [31, Lemma 9]. □

Lemma B.5 (Beta Reduction Canonical Form). *If*

(i) $\cdot \vdash (x) \Rightarrow e : \tau \to \tau' \hookrightarrow v_1$

(ii) $\cdot \vdash v_2 : \tau \hookrightarrow v_2$

(iii) $((x) \Rightarrow e)(v_2) \longrightarrow [v_2/x]\,(e)$

then $v_1 \equiv (x) \Rightarrow w$ *for some* w.

Lemma B.6 (Primitive Reduction Canonical Form). *If*

(i) $\cdot \vdash n : \tau \to \tau' \hookrightarrow v_1$

(ii) $\cdot \vdash v : \tau \hookrightarrow v_2$

(iii) $n(v) \longrightarrow [\![n]\!](v)$

then

(a) $v_1 \equiv n$

(b) $v_2 \not\equiv \mathsf{dead}\downarrow_{\tau}^{\cdot}(\cdot)$

Lemma B.7 (Conditional Canonical Form). *If*

(i) $\cdot \vdash n : \mathsf{Bool} \hookrightarrow v$

(ii) $\cdot \vdash e_1 : \tau \hookrightarrow w_1$ *and* $\cdot \vdash e_2 : \tau \hookrightarrow w_2$

(iii) $\mathtt{if}\ (n)\ \{e_1\}\ \mathtt{else}\ \{e_2\} \longrightarrow e_k$

Then:

(a $k = 1 \implies n = v \equiv \mathtt{true}$

(b $k = 2 \implies n = v \equiv \mathtt{false}$

Lemma B.8 (Value Monotonicity)**.** *If*

$$\Gamma \vdash e : \tau \hookrightarrow v$$

then there exists v *s.t.:*

(a) $e \longrightarrow^* v$

(b) $\Gamma \vdash v : \tau \hookrightarrow v$

(c) $\forall i\,(e \longrightarrow^* e_i)\,.\,\Gamma \vdash e_i : \tau \hookrightarrow v$

Proof. Parts (a) and (b) of the lemma has been proven by Dunfield [31] for a similar language, so here we are just going to prove part (c).

We will show this by induction on the length i of the path: $e \longrightarrow^* e_i$.

- $i = 0$: $e \equiv e_i$, so it trivially holds.
- Suppose it holds for $i = k$, *i.e.* for $e \longrightarrow^* e_k$, it holds that:

$$\Gamma \vdash e_k : \tau \hookrightarrow v \tag{B.8.1}$$

We will show that it holds for $i = k + 1$, *i.e.* for e_{k+1} such that:

$$e_k \longrightarrow e_{k+1} \tag{B.8.2}$$

We will do this by induction on the derivation (B.8.1), but limit ourselves to the terms e_k that elaborate to *values*:

- ♦ Cases T-CST, T-VAR, T-∧I, T-ARROW: For these cases, term e_k is already a value, so doesn't step.
- ♦ Case T-∨I (assume left injection – the case for right injection is similar):

$$\frac{\Gamma \vdash e_k : \tau_1 \hookrightarrow v \qquad \vdash \tau_1 \vee \tau_2}{\Gamma \vdash e_k : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_1\ v}$$

By inversion:

$$\Gamma \vdash e_k : \tau_1 \hookrightarrow v$$

By induction hypothesis, using (B.8.2):

$$\Gamma \vdash e_{k+1} : \tau_1 \hookrightarrow v$$

Applying rule T-∨I on the latter one:

$$\Gamma \vdash e_{k+1} : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_1\ v$$

□

Lemma B.9 (Reverse Value Monotonicity). *If* $\Gamma \vdash v : \tau \hookrightarrow w$*, then there exists* v *s.t.* $w \longrightarrow^* v$ *and* $\Gamma \vdash v : \tau \hookrightarrow v$.

Proof. Similar to proof of Lemma B.8. □

Lemma B.10 (Substitution). *If* $\Gamma, x\colon \tau \vdash e : \tau' \hookrightarrow w$ *and* $\Gamma \vdash v : \tau \hookrightarrow v$*, then* $\Gamma \vdash [v/x](e) : \tau' \hookrightarrow [v/x](w)$.

Proof. Based on Dunfield [31, Lemma 12]. □

Corollary B.2.1 (Target Multi-step Preservation). *If* $\cdot \vdash w :: T$ *and* $w \longrightarrow^* w'$*, then* $\cdot \vdash w' :: T$.

Proof. Stems from Theorem 3.7. □

Corollary B.2.2 (dead-cast Invalid). $\cdot \not\vdash \mathsf{dead}\downarrow^{\tau}_{\tau'} (w) :: T$

Lemma B.11 (Correspondence). *If*

(i) $\Gamma \vdash e : \tau \hookrightarrow w$

(ii) $G \vdash w :: T$

(iii) $[\Gamma] = \|G\|$

then

$[\tau] = \|T\|$

Proof. We prove this by induction on pairs T-*Rule*/R-*Rule* of derivations:

$$\frac{\Gamma \vdash e : \tau \hookrightarrow w}{G \vdash w :: T}$$

- T-CST/T-CST:

$$\frac{\Gamma \vdash n : b \hookrightarrow n}{G \vdash n :: \{\nu\colon b \mid \nu = n\}}$$

  Meta-function sngl operates entirely on the refinement so it holds that:

$$\|\{\nu\colon b \mid \nu = n\}\| = \|b\|$$

  Also, it holds that:

$$[b] = \|b\|$$

- T-VAR/T-VAR:

$$\frac{x\colon \tau \in \Gamma}{\Gamma \vdash x : \tau \hookrightarrow x} \qquad \frac{x\colon T \in G}{G \vdash x :: \mathsf{sngl}\,(T,\, x)}$$

By inversion:

$$x\colon \tau \in \Gamma \tag{B.11.1}$$

$$x\colon T \in G \tag{B.11.2}$$

If x is bound multiple times in Γ and G, we assume the we have picked the correct instances from each environment.

By (iii) we have that:

$$[\Gamma(x)] = \|G(x)\|$$

Also meta-function sngl operates entirely on the refinement so it holds that:

$$\|T\| = \|\mathsf{sngl}\,(T,\, x)\| \tag{B.11.3}$$

By (B.11.1), (B.11.2) and (B.11.3) it holds that:

$$[\tau] = \|\mathsf{sngl}\,(T,\, x)\|$$

- T-IF/T-IF: *Similar to previous case*.

- T-$\wedge$I/T-PPAIR:

From the first premise of the implication:

$$\frac{\forall k \in \{1,2\}\,.\,\Gamma \vdash v : \tau_k \hookrightarrow v_k}{\Gamma \vdash v : \tau_1 \wedge \tau_2 \hookrightarrow v_1,\, v_2}$$

By inversion:

$$\forall k \in \{1,2\}\,.\,\Gamma \vdash v : \tau_k \hookrightarrow v_k \tag{B.11.4}$$

From the second premise of the implication:

$$\frac{\forall k \in \{1,2\}\,.\,G \vdash v_k :: T_k}{G \vdash v_1,\, v_2 :: T_1 \times T_2}$$

By inversion:

$$\forall k \in \{1,2\}\,.\, G \vdash v_k :: T_k \tag{B.11.5}$$

By induction hypothesis on (iii), (B.11.4) and (B.11.5):

$$\forall k \in \{1,2\}\,.\, [\![\tau_k]\!] = \|T_k\| \tag{B.11.6}$$

Using properties of $[\![\cdot]\!]$ and $\|\cdot\|$:

$$[\![\tau_1 \wedge \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!] = \|T_1\| \times \|T_2\| = \|T_1 \times T_2\|$$

- T-$\wedge$E/T-PROJ: *Straightforward based on earlier cases.*
- T-ARROW/T-LAM: *Straightforward based on earlier cases.*
- T-APP/T-APP: *Straightforward based on earlier cases.*
- T-$\vee$I/T-INJ: *Straightforward based on earlier cases.*
- T-$\vee$E/T-CASE:

  From the first premise of the implication:

$$\frac{\Gamma \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \qquad \begin{array}{c}\Gamma, x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \\ \Gamma, x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2\end{array}}{\Gamma \vdash E\langle e_0 \rangle : \tau' \hookrightarrow \texttt{case}\ w_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2}$$

  By inversion:

$$\Gamma \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \tag{B.11.7}$$

$$\Gamma, x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \tag{B.11.8}$$

$$\Gamma, x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2 \tag{B.11.9}$$

  From the second premise of the implication:

$$\frac{G \vdash w_0 :: T_1 + T_2 \qquad G, x_1 : T_1 \vdash w_1 :: T \qquad G, x_2 : T_2 \vdash w_2 :: T}{G \vdash \texttt{case}\ w_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 :: T}$$

  By inversion:

$$G \vdash w_0 :: T_1 + T_2 \tag{B.11.10}$$

$$G, x_1 : T_1 \vdash w_1 :: T \tag{B.11.11}$$

$$G, x_2 : T_2 \vdash w_2 :: T \tag{B.11.12}$$

By induction hypothesis on (iii), (B.11.7) and (B.11.10):

$$[\tau_1 \vee \tau_2] = \|T_1 + T_2\|$$

From properties of type elaboration and refinement types:

$$[\tau_1 \vee \tau_2] = [\tau_1] + [\tau_2]$$
$$\|T_1 + T_2\| = \|T_1\| + \|T_2\|$$

The right-hand side of the last two equations are tagged unions, so it is possible to match the consituent parts by structure:

$$[\tau_1] = \|T_1\| \quad \text{and} \quad [\tau_2] = \|T_2\|$$

Combining the last equation with (iii):

$$[\Gamma, x\colon \tau_1] = \|G, x\colon T_1\| \tag{B.11.13}$$
$$[\Gamma, x\colon \tau_2] = \|G, x\colon T_2\| \tag{B.11.14}$$

By induction hypothesis on (B.11.8), (B.11.11) and (B.11.13) (or (B.11.9), (B.11.12) and (B.11.14)):

$$[\tau'] = \|T\|$$

- T-$\bot$/T-App:

  From the first premise of the implication:

$$\frac{\Gamma \vdash e : \tau \hookrightarrow w \qquad tag(\tau) \cap tag(\tau') = \varnothing}{\Gamma \vdash e : \tau' \hookrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w)}$$

  From the second premise of the implication:

$$\frac{G \vdash \mathsf{dead}\downarrow^{\tau}_{\tau'} :: \mathsf{Bot}([\tau]) \to \mathsf{Bot}([\tau']) \qquad G \vdash w :: T'}{G \vdash \mathsf{dead}\downarrow^{\tau}_{\tau'}(w) :: [w/x]\left(\mathsf{Bot}([\tau'])\right)}$$

  The result type of the last derivation can also be written as:

$$[w/x]\left(\mathsf{Bot}([\tau'])\right) = \mathsf{Bot}([\tau'])$$

  Because after the application of $\mathsf{Bot}(\cdot)$ all original refinements get erased. Also, after removing the refinements:

$$\|\mathsf{Bot}([\tau'])\| = [\tau']$$

□

B.3 Theorems

Theorem B.12 (Consistency). *If* $\cdot \vdash e : \tau \hookrightarrow w$ *and* $w \longrightarrow w'$ *then there exists* e' *such that* $e \longrightarrow^* e'$ *and* $\cdot \vdash e' : \tau \hookrightarrow w'$.

Proof. By induction on the derivation $\cdot \vdash e : \tau \hookrightarrow w$:

- T-CST, T-VAR, T-∧I, T-∧E and T-ARROW: The respective target expression does not step.
- T-IF:

$$\frac{\cdot \vdash e_c : \texttt{boolean} \hookrightarrow w \qquad \forall i \in \{1,2\}. \cdot \vdash e_i : \tau \hookrightarrow w_i}{\cdot \vdash \texttt{if}\ (e_c)\ \{e_1\}\ \texttt{else}\ \{e_2\} : \tau \hookrightarrow \texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\}}$$

  By inversion:

$$\cdot \vdash e_c : \texttt{boolean} \hookrightarrow w_c \tag{B.12.1}$$
$$\cdot \vdash e_1 : \tau \hookrightarrow w_1 \tag{B.12.2}$$
$$\cdot \vdash e_2 : \tau \hookrightarrow w_2 \tag{B.12.3}$$

  Cases on the form of $w \longrightarrow w'$:

  - ♦ Rule:

$$\frac{w_c \longrightarrow w_c'}{\texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow \texttt{if}\ (w_c')\ \{w_1\}\ \texttt{else}\ \{w_2\}}$$

    By inversion:

$$w_c \longrightarrow w_c' \tag{B.12.4}$$

    By induction hypothesis using (B.12.1) and (B.12.4) there exists e_c' such that

$$e_c \longrightarrow^* e_c'$$
$$\cdot \vdash e_c' : \texttt{boolean} \hookrightarrow w_c' \tag{B.12.5}$$

    Applying rule T-IF on (B.12.5), (B.12.2) and (B.12.3) we get:

$$\cdot \vdash \texttt{if}\ (e_c')\ \{e_1\}\ \texttt{else}\ \{e_2\} : \tau \hookrightarrow \texttt{if}\ (w_c')\ \{w_1\}\ \texttt{else}\ \{w_2\}$$

  - ♦ Rule:

$$\texttt{if}\ (\texttt{true})\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow w_1$$

Equation (B.12.1) becomes:

$$\cdot \vdash e_c : \texttt{boolean} \hookrightarrow \texttt{true}$$

By Lemma B.8 there exists v_c such that:

$$e_c \longrightarrow^* v_c \tag{B.12.6}$$
$$\cdot \vdash v_c : \texttt{boolean} \hookrightarrow \texttt{true} \tag{B.12.7}$$

The only possible case for (B.12.7) to hold is:

$$v_c \equiv \texttt{true}$$

By Lemma B.1 using (B.12.6) on $E \equiv \texttt{if}\ (\langle\rangle)\ \{e_1\}\ \texttt{else}\ \{e_2\}$:

$$\texttt{if}\ (e_c)\ \{e_1\}\ \texttt{else}\ \{e_2\} \longrightarrow^* \texttt{if}\ (\texttt{true})\ \{e_1\}\ \texttt{else}\ \{e_2\}$$

By E-COND:

$$\texttt{if}\ (\texttt{true})\ \{e_1\}\ \texttt{else}\ \{e_2\} \longrightarrow e_1$$

So there exists $e' \equiv e_1$, such that $e \longrightarrow^* e'$ and by (B.12.2) it holds that:

$$\cdot \vdash e' : \tau \hookrightarrow w_1$$

- Rule:

$$\texttt{if}\ (\texttt{false})\ \{w_1\}\ \texttt{else}\ \{w_1\} \longrightarrow w_2$$

This case is similar to the previous one.

- T-APP: *Similar to Dunfield [31, Proof of Theorem 13]*
- T-$\vee$I:

$$\frac{\cdot \vdash e : \tau_k \hookrightarrow w_0 \qquad \vdash \tau_1 \vee \tau_2}{\cdot \vdash e : \tau_1 \vee \tau_2 \hookrightarrow \texttt{inj}_k\ w_0}$$

By inversion:

$$\cdot \vdash e : \tau_k \hookrightarrow w_0 \tag{B.12.8}$$
$$\vdash \tau_1 \vee \tau_2 \tag{B.12.9}$$

The only possible case for $w \longrightarrow w'$ is:

$$\frac{w_0 \longrightarrow w_0'}{\texttt{inj}_k\ w_0 \longrightarrow \texttt{inj}_k\ w_0'}$$

By inversion:

$$w_0 \longrightarrow w_0' \tag{B.12.10}$$

By induction hypothesis using (B.12.8) and (B.12.10) there exists an e' such that:

$$e \longrightarrow^* e' \tag{B.12.11}$$
$$\cdot \vdash e' : \tau_k \hookrightarrow w_0' \tag{B.12.12}$$

By T-$\vee$I on (B.12.12) and (B.12.9):

$$\cdot \vdash e' : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_k\ w_0'$$

- T-$\vee$E:

$$\frac{\cdot \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \qquad \begin{array}{r} x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \\ x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2 \end{array}}{\cdot \vdash E\langle e_0 \rangle : \tau' \hookrightarrow \mathtt{case}\ w_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2}$$

By inversion:

$$\cdot \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \tag{B.12.13}$$
$$x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \tag{B.12.14}$$
$$x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2 \tag{B.12.15}$$

Cases on the form of $w \longrightarrow w'$:

- ♦ Rule:

$$\frac{w_0 \longrightarrow w_0'}{\begin{array}{c}\mathtt{case}\ w_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2 \longrightarrow \\ \mathtt{case}\ w_0'\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2\end{array}}$$

  By inversion:

$$w_0 \longrightarrow w_0' \tag{B.12.16}$$

  By induction hypothesis using (B.12.13) and (B.12.16) there exists e_0' such that

$$e_0 \longrightarrow^* e_0' \tag{B.12.17}$$
$$\cdot \vdash e_0' : \tau_1 \vee \tau_2 \hookrightarrow w_0' \tag{B.12.18}$$

Applying T-$\vee$E on (B.12.18), (B.12.14) and (B.12.15):

$$\cdot \vdash \mathsf{E}\langle e_0' \rangle : \tau_1 \hookrightarrow \texttt{case}\ w_0'\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2$$

By Lemma B.2 using (B.12.17):

$$\mathsf{E}\langle e_0 \rangle \longrightarrow^* \mathsf{E}\langle e_0' \rangle$$

- Rule:

$$\texttt{case}\ \texttt{inj}_1\ v\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 \longrightarrow [v/x_1]\,(w_1)$$

Equation (B.12.13) becomes:

$$\cdot \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow \texttt{inj}_1\ v \tag{B.12.19}$$

By Lemma B.8 on (B.12.19), there exists v_0 such that:

$$e_0 \longrightarrow^* v_0 \tag{B.12.20}$$

$$\cdot \vdash v_0 : \tau_1 \vee \tau_2 \hookrightarrow \texttt{inj}_1\ v \tag{B.12.21}$$

By Lemma B.3 on (B.12.21):

$$\cdot \vdash v_0 : \tau_1 \hookrightarrow v \tag{B.12.22}$$

By Lemma B.10 on (B.12.14) and (B.12.22):

$$\cdot \vdash [v_0/x_1]\,(\mathsf{E}\langle x_1 \rangle) : \tau_1 \hookrightarrow [v/x_1]\,(w_1)$$

Or, after the substitutions[1]:

$$\cdot \vdash \mathsf{E}\langle v_0 \rangle : \tau_1 \hookrightarrow [v/x_1]\,(w_1) \tag{B.12.23}$$

By Lemma B.1 on (B.12.20):

$$\mathsf{E}\langle e_0 \rangle \longrightarrow^* \mathsf{E}\langle v_0 \rangle$$

- Rule

$$\texttt{case}\ \texttt{inj}_2\ v\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow w_1 \mid \texttt{inj}_2\ x_2 \Rightarrow w_2 \longrightarrow [v/x_2]\,(w_2)$$

This case is similar to the previous one.

[1] Variable x_1 is only referenced in the "hole" of the evaluation context $\mathsf{E}\langle x_1 \rangle$.

- T-$\bot$:

$$\frac{\cdot \vdash e : \tau \hookrightarrow w \qquad \mathtt{tag}(\tau) \cap \mathtt{tag}(\tau') = \varnothing}{\cdot \vdash e : \tau' \hookrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w)}$$

By inversion:

$$\cdot \vdash e : \tau \hookrightarrow w \tag{B.12.24}$$

$$\mathtt{tag}(\tau) \cap \mathtt{tag}(\tau') = \varnothing \tag{B.12.25}$$

The only possible step here is:

$$\frac{w \longrightarrow w'}{\mathsf{dead}\downarrow^{\tau}_{\tau'}(w) \longrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w')}$$

By inversion:

$$w \longrightarrow w' \tag{B.12.26}$$

By induction hypothesis using (B.12.24) and (B.12.26) there exists e' such that:

$$e \longrightarrow^{*} e'$$

$$\cdot \vdash e' : \tau \hookrightarrow w' \tag{B.12.27}$$

By applying T-$\bot$ on (B.12.27) and (B.12.25):

$$\cdot \vdash e' : \tau' \hookrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w')$$

□

Theorem B.13 (Reverse Consistency)**.** *If* $\cdot \vdash e : \tau \hookrightarrow w$ *and* $e \longrightarrow e'$*, then there exists* w' *such that* $\cdot \vdash e' : \tau \hookrightarrow w'$ *and* $w \longrightarrow^{+} w'$*.*

Proof. By induction on the derivation $\cdot \vdash e : \tau \hookrightarrow w$:

- T-CST, T-VAR, T-$\wedge$I, T-ARROW: The respective source expression does not step.
- T-IF:

$$\frac{\cdot \vdash e_c : \mathtt{boolean} \hookrightarrow w \qquad \forall i \in \{1,2\}. \cdot \vdash e_i : \tau \hookrightarrow w_i}{\cdot \vdash \mathtt{if}\,(e_c)\,\{e_1\}\,\mathtt{else}\,\{e_2\} : \tau \hookrightarrow \mathtt{if}\,(w_c)\,\{w_1\}\,\mathtt{else}\,\{w_2\}} \tag{B.13.1}$$

By inversion:

$$\cdot \vdash e_c : \texttt{boolean} \hookrightarrow w_c \tag{B.13.2}$$
$$\cdot \vdash e_1 : \tau \hookrightarrow w_1 \tag{B.13.3}$$
$$\cdot \vdash e_2 : \tau \hookrightarrow w_2 \tag{B.13.4}$$

Cases on the form of $e \longrightarrow e'$:

- Rule:

$$\frac{e_c \longrightarrow e_c'}{\texttt{if}\ (e_c)\ \{e_1\}\ \texttt{else}\ \{e_2\} \longrightarrow \texttt{if}\ (e_c')\ \{e_1\}\ \texttt{else}\ \{e_2\}}$$

  By inversion:

$$e_c \longrightarrow e_c' \tag{B.13.5}$$

  By induction hypothesis using (B.13.2) and (B.13.5) there exists w_c' such that:

$$\cdot \vdash e_c' : \texttt{boolean} \hookrightarrow w_c' \tag{B.13.6}$$
$$w_c \longrightarrow^+ w_c' \tag{B.13.7}$$

  By Lemma B.2 using (B.13.7):

$$\texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow^+ \texttt{if}\ (w_c')\ \{w_1\}\ \texttt{else}\ \{w_2\}$$

  Applying rule T-If on (B.13.6), (B.13.3) and (B.13.4) we get:

$$\cdot \vdash \texttt{if}\ (e_c')\ \{e_1\}\ \texttt{else}\ \{e_2\} : \tau \hookrightarrow \texttt{if}\ (w_c')\ \{w_1\}\ \texttt{else}\ \{w_2\}$$

- Rule:

$$\texttt{if}\ (\texttt{true})\ \{e_1\}\ \texttt{else}\ \{e_2\} \longrightarrow e_1 \tag{B.13.8}$$

  Equation (B.13.2) becomes:

$$\cdot \vdash \texttt{true} : \texttt{boolean} \hookrightarrow w_c \tag{B.13.9}$$

  By Lemma B.9 there exists v_c such that:

$$w_c \longrightarrow^* v_c \tag{B.13.10}$$
$$\cdot \vdash \texttt{true} : \texttt{boolean} \hookrightarrow v_c \tag{B.13.11}$$

By Lemma B.2 using (B.13.10):

$$\texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow^* \texttt{if}\ (v_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} \tag{B.13.12}$$

By Lemma B.7 on (B.13.11), (B.13.3), (B.13.4) and (B.13.8):

$$v_c \equiv \texttt{true} \tag{B.13.13}$$

By E-COND:

$$\texttt{if}\ (\texttt{true})\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow w_1 \tag{B.13.14}$$

By (B.13.12) and (B.13.14):

$$\texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} \longrightarrow^+ w_1$$

Combining with (B.13.3) we get the wanted relation.

- Rule:

$$\texttt{if}\ (\texttt{false})\ \{e_1\}\ \texttt{else}\ \{e_1\} \longrightarrow e_2$$

Similar to the previous case.

- T-∧E: *Similar to earlier cases.*
- T-APP:

$$\frac{\cdot \vdash e_1 : \tau \to \tau' \hookrightarrow w_1 \qquad \cdot \vdash e_2 : \tau \hookrightarrow w_2}{\cdot \vdash e_1\,(e_2) : \tau' \hookrightarrow w_1\,(w_2)} \tag{B.13.15}$$

By inversion:

$$\cdot \vdash e_1 : \tau \to \tau' \hookrightarrow w_1 \tag{B.13.16}$$

$$\cdot \vdash e_2 : \tau \hookrightarrow w_2 \tag{B.13.17}$$

Cases on the form of $e \longrightarrow e'$:

- Rule:

$$\frac{e_1 \longrightarrow e_1'}{e_1\,(e_2) \longrightarrow e_1'\,(e_2)}$$

Similar to earlier cases.

- Rule:

$$\frac{e_2 \longrightarrow e_2'}{v_1\,(e_2) \longrightarrow v_1\,(e_2')}$$

By inversion:

$$e_2 \longrightarrow e_2' \tag{B.13.18}$$

By Lemma B.9 on (B.13.16) there exists v_1 such that:

$$w_1 \longrightarrow^* v_1 \tag{B.13.19}$$
$$\cdot \vdash v_1 : \tau_2 \hookrightarrow v_1 \tag{B.13.20}$$

By Lemma B.2 using (B.13.19):

$$w_1\,(w_2) \longrightarrow^* v_1\,(w_2) \tag{B.13.21}$$

By induction hypothesis using (B.13.17) and (B.13.18) there exists w_2' such that:

$$w_2 \longrightarrow^+ w_2' \tag{B.13.22}$$
$$\cdot \vdash e_2' : \tau \hookrightarrow w_2' \tag{B.13.23}$$

By Lemma B.2 using (B.13.22) on the target of (B.13.20):

$$v_1\,(w_2) \longrightarrow^+ v_1\,(w_2')$$

And combining with (B.13.21):

$$w_1\,(w_2) \longrightarrow^+ v_1\,(w_2')$$

By Rule T-App using (B.13.16) and (B.13.23):

$$\cdot \vdash v_1\,(e_2') : \tau \hookrightarrow v_1\,(w_2')$$

♦ Rule:

$$((x) \Rightarrow e_0)(v_2) \longrightarrow [v_2/x]\,(e_0) \tag{B.13.24}$$

By Lemma B.9 on (B.13.16) there exists v_1 such that:

$$\cdot \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow v_1 \tag{B.13.25}$$
$$w_1 \longrightarrow^* v_1 \tag{B.13.26}$$

By applying Lemma B.2 on $w \equiv w_1\,(w_2)$ given (B.13.26):

$$w_1\,(w_2) \longrightarrow^* v_1\,(w_2) \tag{B.13.27}$$

Equation (B.13.17) is:

$$\cdot \vdash v_2 : \tau \hookrightarrow w_2 \tag{B.13.28}$$

By Lemma B.9 on (B.13.28), there exists v_2 such that:

$$w_2 \longrightarrow^* v_2 \tag{B.13.29}$$

$$\cdot \vdash v_2 : \tau \hookrightarrow v_2 \tag{B.13.30}$$

By Lemma B.5 on (B.13.25), (B.13.30) and (B.13.24), there is a w_0 such that:

$$v_1 \equiv (x) \Rightarrow w_0$$

So (B.13.16) becomes:

$$\cdot \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow (x) \Rightarrow w_0 \tag{B.13.31}$$

The only production of (B.13.31) is by T-ARROW:

$$\frac{\vdash \tau \to \tau' \qquad x : \tau \vdash e_0 : \tau' \hookrightarrow w_0}{\Gamma \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow (x) \Rightarrow w_0}$$

By inversion:

$$x : \tau \vdash e_0 : \tau' \hookrightarrow w_0 \tag{B.13.32}$$

By applying Lemma B.10 on (B.13.32) and (B.13.30) we get:

$$\cdot \vdash [v_2/x](e_0) : \tau' \hookrightarrow [v_2/x](w_0) \tag{B.13.33}$$

By applying Lemma B.2 on $w \equiv ((x) \Rightarrow w_0)(w_2)$ given (B.13.29):

$$((x) \Rightarrow w_0)(w_2) \longrightarrow^* ((x) \Rightarrow w_0)(v_2) \tag{B.13.34}$$

By Rule TE-APP-2:

$$((x) \Rightarrow w_0)(v_2) \longrightarrow [v_2/x](w_0) \tag{B.13.35}$$

By (B.13.27), (B.13.34) and (B.13.35) we get:

$$w_1(w_2) \longrightarrow^+ [v_2/x](w_0) \tag{B.13.36}$$

By (B.13.33) and (B.13.36) we get the wanted relation.

♦ Rule:

$$n(v) \longrightarrow [\![n]\!](v) \tag{B.13.37}$$

Equations (B.13.16) and (B.13.17) become:

$$\cdot \vdash n : \tau \to \tau' \hookrightarrow w_1 \tag{B.13.38}$$
$$\cdot \vdash v : \tau \hookrightarrow w_2 \tag{B.13.39}$$

By Lemma B.9 on (B.13.38) there exists v_1 such that:

$$\cdot \vdash n : \tau \to \tau' \hookrightarrow v_1 \tag{B.13.40}$$
$$w_1 \longrightarrow^* v_1 \tag{B.13.41}$$

By Lemma B.2 on $w \equiv w_1\,(w_2)$ given (B.13.41):

$$w_1\,(w_2) \longrightarrow^* v_1\,(w_2) \tag{B.13.42}$$

By Lemma B.9 on (B.13.39) there exists v_2 such that:

$$w_2 \longrightarrow^* v_2 \tag{B.13.43}$$
$$\cdot \vdash v : \tau \hookrightarrow v_2 \tag{B.13.44}$$

By Lemma B.2 on (B.13.43):

$$n(w_2) \longrightarrow^* n(v_2) \tag{B.13.45}$$

By Lemma B.6 on (B.13.40), (B.13.44) and (B.13.37):

$$v_1 \equiv n \tag{B.13.46}$$
$$v_2 \not\equiv \mathsf{dead}\downarrow^{\cdot}_{\tau}(\,\cdot\,) \tag{B.13.47}$$

So (B.13.40) becomes:

$$\cdot \vdash n : \tau \to \tau' \hookrightarrow n \tag{B.13.48}$$

So we can apply TE-APP-1:

$$n(v_2) \longrightarrow [\![n]\!](v_2) \tag{B.13.49}$$

By (B.13.42), (B.13.45) and (B.13.49):

$$w_1\,(w_2) \longrightarrow^+ [\![n]\!](v_2)$$

By assumption B.1.1 using (B.13.48), (B.13.47) and (B.13.44):

$$\cdot \vdash [\![n]\!](v) : \tau \hookrightarrow [\![n]\!](v_2)$$

- T-$\vee$I:

$$\frac{\cdot \vdash e : \tau_k \hookrightarrow w \qquad \vdash \tau_1 \vee \tau_2}{\cdot \vdash e : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_k\ w}$$

By inversion:

$$\cdot \vdash e : \tau_k \hookrightarrow w \tag{B.13.50}$$
$$\vdash \tau_1 \vee \tau_2 \tag{B.13.51}$$

By induction hypothesis using (B.13.50) with $e \longrightarrow e'$, there exists w' such that:

$$\cdot \vdash e' : \tau_k \hookrightarrow w' \tag{B.13.52}$$
$$w \longrightarrow^+ w' \tag{B.13.53}$$

By Lemma B.2 using (B.13.53):

$$\mathtt{inj}_k\ w \longrightarrow^+ \mathtt{inj}_k\ w'$$

Applying T-$\vee$I with premises (B.13.52) and (B.13.51):

$$\cdot \vdash e' : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_k\ w'$$

- T-$\vee$E:

$$\frac{\cdot \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \qquad \begin{array}{c} x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \\ x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2 \end{array}}{\cdot \vdash E\langle e_0 \rangle : \tau' \hookrightarrow \mathtt{case}\ w_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2}$$

By inversion:

$$\cdot \vdash e_0 : \tau_1 \vee \tau_2 \hookrightarrow w_0 \tag{B.13.54}$$
$$x_1 : \tau_1 \vdash E\langle x_1 \rangle : \tau' \hookrightarrow w_1 \tag{B.13.55}$$
$$x_2 : \tau_2 \vdash E\langle x_2 \rangle : \tau' \hookrightarrow w_2 \tag{B.13.56}$$

Cases on the form of $e \longrightarrow e'$:

- Rule:

$$\frac{e_0 \longrightarrow e_0'}{\mathsf{E}\langle e_0\rangle \longrightarrow \mathsf{E}\langle e_0'\rangle}$$

By inversion:

$$e_0 \longrightarrow e_0' \tag{B.13.57}$$

By induction hypothesis using (B.13.54) and (B.13.57):

$$\cdot \vdash e_0' : \tau_1 \vee \tau_2 \hookrightarrow w_0' \tag{B.13.58}$$
$$w_0 \longrightarrow^+ w_0' \tag{B.13.59}$$

Using rule T-$\vee$E on (B.13.58), (B.13.55) and (B.13.56):

$$\cdot \vdash \mathsf{E}\langle e_0'\rangle : \tau' \hookrightarrow \mathtt{case}\ w_0'\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2$$

Also, applying Lemma B.2 on $\mathcal{E} \equiv \mathtt{case}\ \langle\rangle\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2$ using (B.13.59):

$$\mathtt{case}\ w_0\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2 \longrightarrow^+ \mathtt{case}\ w_0'\ \mathtt{of}\ \mathtt{inj}_1\ x_1 \Rightarrow w_1 \mid \mathtt{inj}_2\ x_2 \Rightarrow w_2$$

- Rule:

$$e_0 \equiv v_0 \tag{B.13.60}$$
$$\mathsf{E}\langle v_0\rangle \longrightarrow e' \tag{B.13.61}$$

Because v_0 is a value, we can split cases for its type. Without loss of generality we can assume that its type is τ_1 (the same exact holds for τ_2). This is depicted on the form of w_0 in equation (B.13.54), which now becomes (for some w_{01}):

$$\cdot \vdash v_0 : \tau_1 \vee \tau_2 \hookrightarrow \mathtt{inj}_1\ w_{01} \tag{B.13.62}$$

By Lemma B.3 on (B.13.62):

$$\cdot \vdash v_0 : \tau_1 \hookrightarrow w_{01} \tag{B.13.63}$$

By Lemma B.9 on (B.13.63) there exists v_{01} such that:

$$w_{01} \longrightarrow^* v_{01} \tag{B.13.64}$$
$$\cdot \vdash v_0 : \tau_1 \hookrightarrow v_{01} \tag{B.13.65}$$

By Lemma B.9 on (B.13.62) there exists v_{01}[2] such that:

$$\texttt{inj}_1\ w_{01} \longrightarrow^* \texttt{inj}_1\ v_{01} \tag{B.13.66}$$

$$\cdot \vdash v_0 : \tau_1 \vee \tau_2 \hookrightarrow \texttt{inj}_1\ v_{01} \tag{B.13.67}$$

Cases for the form of $\mathsf{E}\langle v_0 \rangle$:

- $\mathsf{E}\langle v_0 \rangle \equiv \texttt{if}\ (v_0)\ \{e_1\}\ \texttt{else}\ \{e_2\}$: *Similar to case* $\texttt{if}\ (e_c)\ \{e_1\}\ \texttt{else}\ \{e_2\}$
- $\mathsf{E}\langle v_0 \rangle \equiv v_0\,(e)$: *Similar to earlier cases.*
- $\mathsf{E}\langle v_0 \rangle \equiv (\texttt{(x)} \Rightarrow e_0)(v_0)$: *Similar to earlier cases.*

- T-$\bot$:

$$\frac{\cdot \vdash e : \tau \hookrightarrow w \qquad \mathtt{tag}(\tau) \cap \mathtt{tag}(\tau') = \varnothing}{\cdot \vdash e : \tau' \hookrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w)}$$

By inversion:

$$\cdot \vdash e : \tau \hookrightarrow w \tag{B.13.68}$$

$$\mathtt{tag}(\tau) \cap \mathtt{tag}(\tau') = \varnothing \tag{B.13.69}$$

There also exists e' such that:

$$e \longrightarrow e' \tag{B.13.70}$$

By induction hypothesis on (B.13.68) and (B.13.70) there exists w', such that:

$$w \longrightarrow^+ w' \tag{B.13.71}$$

$$\cdot \vdash e' : \tau \hookrightarrow w' \tag{B.13.72}$$

By Lemma B.2 on (B.13.71):

$$\mathsf{dead}\downarrow^{\tau}_{\tau'}(w) \longrightarrow^+ \mathsf{dead}\downarrow^{\tau}_{\tau'}(w')$$

Applying rule T-$\bot$ on (B.13.72) and (B.13.69):

$$\cdot \vdash e' : \tau' \hookrightarrow \mathsf{dead}\downarrow^{\tau}_{\tau'}(w')$$

□

Theorem B.14 (Two-Phase Safety)**.** *If*

(i) $\cdot \vdash e : \tau \hookrightarrow w$

[2] This is the same that we got right before, due to uniqueness of normal forms.

(ii) $\cdot \vdash w :: \mathsf{T}$

then, either e *is a value, or there exists* e' *s.t.* $e \longrightarrow e'$ *and* $\cdot \vdash e' : \tau \hookrightarrow w'$ *for* w'*, s.t.* $w \longrightarrow^{+} w'$ *and* $\cdot \vdash w' :: \mathsf{T}$.

Proof. By induction on pairs T-*Rule*/R-*Rule* of derivations:

$$\frac{\Gamma \vdash e : \tau \hookrightarrow w}{G \vdash w :: \mathsf{T}}$$

- T-CST/T-CST, T-VAR/T-VAR, T-$\wedge$I/T-PPAIR, T-ARROW/T-LAM: The term e is a value.
- T-IF/T-IF:

  From (i) we have:

$$\frac{\cdot \vdash e_c : \mathsf{Bool} \hookrightarrow w \qquad \forall i \in\in \{1,2\}. \cdot \vdash e_i : \tau \hookrightarrow w_i}{\cdot \vdash \texttt{if}\ (e_c)\ \{e_1\}\ \texttt{else}\ \{e_2\} : \tau \hookrightarrow \texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\}}$$

  By inversion:

$$\cdot \vdash e_c : \mathsf{Bool} \hookrightarrow w_c \tag{B.14.1}$$
$$\cdot \vdash e_1 : \tau \hookrightarrow w_1 \tag{B.14.2}$$
$$\cdot \vdash e_2 : \tau \hookrightarrow w_2$$

  From (ii):

$$\frac{\cdot \vdash w_c :: \mathsf{Bool} \qquad w_c \vdash w_1 :: \mathsf{T} \qquad \neg w_c \vdash w_2 :: \mathsf{T}}{\cdot \vdash \texttt{if}\ (w_c)\ \{w_1\}\ \texttt{else}\ \{w_2\} :: \mathsf{T}}$$

  By inversion:

$$\cdot \vdash w_c :: \mathsf{Bool} \tag{B.14.3}$$
$$w_c \vdash w_1 :: \mathsf{T} \tag{B.14.4}$$
$$\neg w_c \vdash w_2 :: \mathsf{T} \tag{B.14.5}$$

  By induction hypothesis using (B.14.1) and (B.14.3) we have two case on the form of e_c:

  - Expression e_c is a value:

$$e_c \equiv v_c$$

    By a standard canonical forms lemma v_c is either `true` or `false`. Assume the first case (the latter case is identical but involving the "else" branch of the conditional). By E-COND-TRUE:

$$\texttt{if}\ (\texttt{true})\ \{e_1\}\ \texttt{else}\ \{e_2\} \longrightarrow e_1$$

- ♦ There exists e_c' such that:

$$e_c \longrightarrow e_c' \tag{B.14.6}$$

Hence, by E-ECTX:

$$\texttt{if}\,(e_c)\,\{e_1\}\,\texttt{else}\,\{e_2\} \longrightarrow \texttt{if}\,(e_c')\,\{e_1\}\,\texttt{else}\,\{e_2\}$$

In either case, there exists e' such that:

$$e \longrightarrow e' \tag{B.14.7}$$

By Theorem 3.5 on (i) and (B.14.7), there exists w' such that:

$$w \longrightarrow^{+} w'$$
$$\cdot \vdash e : \tau \hookrightarrow w'$$

And by Corollary B.2.1:

$$\cdot \vdash w' :: T$$

- T-∧E/T-PROJ: Without loss of generality we're going to assume first projection (the same holds for the second projection). From (i):

$$\frac{\cdot \vdash e : \tau_1 \wedge \tau_2 \hookrightarrow w_0}{\cdot \vdash e : \tau_1 \hookrightarrow \texttt{proj}_1\, w_0}$$

By inversion:

$$\cdot \vdash e : \tau_1 \wedge \tau_2 \hookrightarrow w_0 \tag{B.14.8}$$

From (ii):

$$\frac{\cdot \vdash w_0 :: T_1 \times T_2}{\cdot \vdash \texttt{proj}_1 w_0 :: T_1}$$

By inversion:

$$\cdot \vdash w_0 :: T_1 \times T_2 \tag{B.14.9}$$

By induction hypothesis using (B.14.8) and (B.14.9) we have two case on the form of e:

- ♦ Expression e is a value:

$$e \equiv v$$

So the source term does not step.

- There exists e' such that:

$$e \longrightarrow e' \tag{B.14.10}$$

By Theorem 3.5 on (i) and (B.14.10), there exists w' such that:

$$w \longrightarrow^{+} w'$$
$$\cdot \vdash e : \tau \hookrightarrow w'$$

And by Corollary B.2.1:

$$\cdot \vdash w' :: \mathsf{T}$$

- T-App/T-App: From (i):

$$\frac{\cdot \vdash e_1 : \tau \to \tau' \hookrightarrow w_1 \qquad \cdot \vdash e_2 : \tau \hookrightarrow w_2}{\cdot \vdash e_1\,(e_2) : \tau' \hookrightarrow w_1\,(w_2)}$$

By inversion:

$$\cdot \vdash e_1 : \tau \to \tau' \hookrightarrow w_1 \tag{B.14.11}$$
$$\cdot \vdash e_2 : \tau \hookrightarrow w_2 \tag{B.14.12}$$

From (ii):

$$\frac{\cdot \vdash w_1 :: \mathsf{T}_x \to \mathsf{T} \qquad \cdot \vdash w_2 :: \mathsf{T}_x}{\cdot \vdash w_1\,(w_2) :: [w_2/x]\,(\mathsf{T})}$$

By inversion:

$$\cdot \vdash w_1 :: \mathsf{T}_x \to \mathsf{T} \tag{B.14.13}$$
$$\cdot \vdash w_2 :: \mathsf{T}_x \tag{B.14.14}$$

By induction hypothesis using (B.14.11) and (B.14.13) we have three cases on the form of e_1:

- Expression e_1 is a *primitive* value:

$$e_1 \equiv n$$

Elaboration (B.14.11) becomes:

$$\cdot \vdash n : \tau \to \tau' \hookrightarrow w_1 \tag{B.14.15}$$

By Lemma B.9 on (B.14.15) there exists v_1, such that:

$$w_1 \longrightarrow^* v_1 \tag{B.14.16}$$

$$\cdot \vdash n : \tau \to \tau' \hookrightarrow v_1 \tag{B.14.17}$$

By Corollary B.2.1 using (B.14.13) and (B.14.32):

$$\cdot \vdash v_1 :: T_x \to T \tag{B.14.18}$$

By Assumption B.1.3 on (B.14.17):

$$v_1 \equiv n$$

or

$$v_1 \equiv \mathsf{dead}\downarrow^{\cdot}_{\tau} (v_1')$$

The latter case combined with (B.14.18) contradicts Corollary B.2.2, so we end up with:

$$v_1 \equiv n \tag{B.14.19}$$

By Lemma B.2 using (B.14.16):

$$w_1(w_2) \longrightarrow^* n(w_2) \tag{B.14.20}$$

By induction hypothesis using (B.14.12) and (B.14.14) we have two cases on the form of e_2:

- Expression e_2 is a value:

  $$e_2 \equiv v_2$$

  Elaboration (B.14.12) becomes:

  $$\cdot \vdash v_2 : \tau \hookrightarrow w_2 \tag{B.14.21}$$

  By Lemma B.9 on (B.14.21) there exists v_2, such that:

  $$w_2 \longrightarrow^* v_2 \tag{B.14.22}$$

  $$\cdot \vdash v_2 : \tau \hookrightarrow v_2 \tag{B.14.23}$$

  By Lemma B.2 using (B.14.22):

  $$n(w_2) \longrightarrow^* n(v_2) \tag{B.14.24}$$

For the sake of contradiction assume:

$$v_2 \equiv \mathsf{dead}\downarrow_{\tau}^{\tau'}(v_2') \tag{B.14.25}$$

for some v_2'. By (B.14.14):

$$\cdot \vdash v_2 :: T_x \tag{B.14.26}$$

So, by (B.14.26) and Corollary B.2.2 we have a contradiction. So:

$$v_2 \not\equiv \mathsf{dead}\downarrow_{\tau}^{\tau'}(v_2') \tag{B.14.27}$$

By Assumption B.1.1 on (B.14.17), (B.14.19) (B.14.23) and (B.14.27):

$$n(v_2) \longrightarrow [\![n]\!](v_2) \tag{B.14.28}$$

- ▶ There exists e_2' such that:

$$e_2 \longrightarrow e_2' \tag{B.14.29}$$

By E-ECTX:

$$n(e_2) \longrightarrow n(e_2')$$

- ♦ Expression e_1 is an abstraction:

$$e_1 \equiv (x) \Rightarrow e_0 \tag{B.14.30}$$

Elaboration (B.14.11) becomes:

$$\cdot \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow w_1 \tag{B.14.31}$$

By applying lemma B.9 on (B.14.15) there exists v_1, such that:

$$w_1 \longrightarrow^* v_1 \tag{B.14.32}$$

$$\cdot \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow v_1 \tag{B.14.33}$$

By Corollary B.2.1 using (B.14.13) and (B.14.32):

$$\cdot \vdash v_1 :: T_x \to T \tag{B.14.34}$$

By Assumption B.1.3 on (B.14.33):

$$v_1 \equiv (x) \Rightarrow w_0$$

or

$$v_1 \equiv \mathsf{dead}\downarrow^{\cdot}_{\tau\to\tau'}(v_1')$$

The latter case combined with (B.14.34) contradicts Corollary B.2.2, so we end up with:

$$v_1 \equiv (x) \Rightarrow w_0 \tag{B.14.35}$$

So (B.14.11) becomes:

$$\cdot \vdash (x) \Rightarrow e_0 : \tau \to \tau' \hookrightarrow (x) \Rightarrow w_0 \tag{B.14.36}$$

By induction hypothesis using (B.14.12) and (B.14.14) we have two cases on the form of e_2:

- Expression e_2 is a value:

  $$e_2 \equiv v_2$$

  Equation (B.14.12) becomes (for some w_2):

  $$\cdot \vdash v_2 : \tau \hookrightarrow w_2$$

  By Lemma B.9, there exists v_2 such that:

  $$\cdot \vdash v_2 : \tau \hookrightarrow v_2 \tag{B.14.37}$$

  For the sake of contradiction assume:

  $$v_2 \equiv \mathsf{dead}\downarrow^{\tau'}_{\tau}(v_2') \tag{B.14.38}$$

  for some v_2'. By (B.14.14):

  $$\cdot \vdash v_2 :: \mathsf{T}_x \tag{B.14.39}$$

  So, by (B.14.39) and Corollary B.2.2 we have a contradiction. So:

  $$v_2 \not\equiv \mathsf{dead}\downarrow^{\tau'}_{\tau}(v_2') \tag{B.14.40}$$

  By Assumption B.1.2 on (B.14.36), (B.14.37) and (B.14.40):

  $$((x) \Rightarrow e_0)(v_2) \longrightarrow [v_2/x](e_0)$$

- There exists e_2' such that:

  $$e_2 \longrightarrow e_2' \tag{B.14.41}$$

By E-ECTX:

$$((x) \Rightarrow e_0)(e_2) \longrightarrow ((x) \Rightarrow e_0)(e_2')$$

- There exists e_1' such that:

$$e_1 \longrightarrow e_1' \tag{B.14.42}$$

By E-ECTX:

$$e_1(e_2) \longrightarrow e_1'(e_2)$$

In all cases, there exists e' such that:

$$e \longrightarrow e' \tag{B.14.43}$$

By Theorem 3.5 on (i) and (B.14.43), there exists w' such that:

$$w \longrightarrow^+ w'$$
$$\cdot \vdash e : \tau' \hookrightarrow w'$$

And by Corollary B.2.1:

$$\cdot \vdash w' :: T$$

- T-$\vee$I/T-INJ: *Following similar methodology as before.*
- T-$\vee$E/T-CASE: *Following similar methodology as before.*
- T-$\bot$/T-APP: Corollary B.2.2 contradicts the second premise (ii), so the theorem does not apply here.

□

Appendix C

Refinement Types for TypeScript

C.1 Full System

In this section we present the full type system for the core language of Chapter 4.

C.1.1 Formal Languages

Figure C.1 shows the runtime syntax for the input language $\mathsf{I}_{\mathsf{rsc}}$, building up on the language described in Figure 4.4. The type language is the same as described in Figure 4.7. The operational semantics, shown in Figure C.2, is borrowed from Safe TypeScript [87], with certain simplifications since the language we are dealing with is simpler than the one used there. We use evaluation contexts E, with a left to right evaluation order.

Figure C.3 shows the runtime syntax for the SSA transformed language λ_{rsc}, building up on the language described in Figure 4.5. The reduction rules of the operational semantics for language λ_{rsc} are shown in Figure C.4. We use evaluation contexts $\mathcal{E}$, with a left to right evaluation order.

Evaluation Context	E	$::=$	$\langle\rangle \mid \mathsf{E}.\mathtt{f} \mid \mathsf{E}.\mathtt{m}(\overline{e}) \mid v.\mathtt{m}(\overline{v}, \mathsf{E}, \overline{e}) \mid \mathtt{new}\ \mathsf{C}(\overline{v}, \mathsf{E}, \overline{e})$
		$\mid$	$\mathtt{<T>}\mathsf{E} \mid \mathtt{var}\ x = \mathsf{E} \mid \mathsf{E}.\mathtt{f} = e \mid v.\mathtt{f} = \mathsf{E}$
		$\mid$	$x = \mathsf{E} \mid \mathtt{if}\ (\mathsf{E})\ \{s\}\ \mathtt{else}\ \{s\} \mid \mathtt{return}\ \mathsf{E} \mid \mathsf{E}; s$
		$\mid$	$\mathsf{E}; \mathtt{return}\ e$
Runtime Conf.	R	$::=$	S, s
State	S	$::=$	$\langle \mathsf{K}, \mathsf{L}, \mathsf{X}\, \mathsf{H} \rangle$
Store	L	$::=$	$\cdot \mid x \mapsto v \mid \mathsf{L}_1, \mathsf{L}_2$
Value	v	$::=$	$\ell \mid n$
Stack	X	$::=$	$\cdot \mid \mathsf{X}, \mathsf{L}.\mathsf{E}$
Heap	H	$::=$	$\cdot \mid \ell \mapsto \mathsf{O} \mid \mathsf{H}_1, \mathsf{H}_2$
Field Bindings	$\vec{\mathsf{F}}$	$\in$	$\mathsf{F} \to \mathsf{Vals}$
Objects	O	$::=$	$\{\mathtt{proto}: \ell; \mathtt{f}: \vec{\mathsf{F}}; \ldots\} \mid \{\mathtt{name}: \mathsf{C}; \mathtt{proto}: \ell; \mathtt{m}: \mathsf{M}\}$

Figure C.1. Syntax and Runtime Configuration of $\mathrm{I}_{\mathsf{rsc}}$

Expression Reduction Rules (Selected) $\boxed{S;\, e \longrightarrow S';\, e'}$

$$\frac{\langle K; L; \cdot; H\rangle;\, e \longrightarrow \langle K; L'; \cdot; H'\rangle;\, e'}{\langle K; L; \cdot; H\rangle;\, E\langle e\rangle \longrightarrow \langle K; L'; \cdot; H'\rangle;\, E\langle e'\rangle}\ \text{[R-EvalCtx]} \qquad \frac{}{S;\, x \longrightarrow S;\, S.L(x)}\ \text{[R-Var]}$$

$$\frac{S.H(\ell) = \{\texttt{proto}\colon \ell'\texttt{;}\ \texttt{f}\colon \texttt{f:=}v\texttt{;}\ \ldots\}}{S;\, \ell\texttt{.f} \longrightarrow S;\, v}\ \text{[R-DotRef]}$$

$$\frac{\begin{array}{c} H(\ell_0) = \{\texttt{name}\colon C\texttt{;}\ \texttt{proto}\colon \ell_0'\texttt{;}\ \texttt{m}\colon M\} \\ \mathsf{fields}(K,\, C) = \overline{f} \qquad \{\texttt{proto}\colon \ell_0\texttt{;}\ \overline{f}\colon \overline{v}\} = O \qquad H[\,\ell \mapsto O\,] = H' \qquad \ell\ \mathsf{fresh} \end{array}}{\langle K; L; X; H\rangle;\, \texttt{new}\ C(\overline{v}) \longrightarrow \langle K; L; X; H'\rangle;\, \ell}\ \text{[R-New]}$$

$$\frac{\mathsf{resolveMethod}(H,\, \ell) = m(\overline{x})\colon \{\texttt{s; return}\ e\} \qquad L' = \overline{x}\colon \overline{v},\ \texttt{this}\colon \ell \qquad X' = X,\, L.E}{\langle K; L; X; H\rangle;\, E\langle \ell\texttt{.}m(\overline{v})\rangle \longrightarrow \langle K; L'; X'; H\rangle;\, \texttt{s; return}\ e}\ \text{[R-Call]}$$

$$\frac{}{S;\, \texttt{<T>}e \longrightarrow S;\, e}\ \text{[R-Cast]}$$

Statement Reduction Rules (Selected) $\boxed{S;\, s \longrightarrow S';\, s'}$

$$\frac{}{S;\, \texttt{skip; s} \longrightarrow S;\, s}\ \text{[R-Skip]} \qquad \frac{L' = S.L[\,x \mapsto v\,]}{S;\, \texttt{let}\ x = v \longrightarrow S \triangleleft L';\, v}\ \text{[R-VarDecl]}$$

$$\frac{H' = S.H[\,\ell \mapsto S.H(\ell)[\,f \mapsto v\,]\,]}{S;\, \ell\texttt{.f} = v \longrightarrow S \triangleleft H';\, v}\ \text{[R-DotAsgn]} \qquad \frac{L' = S.L[\,x \mapsto v\,]}{S;\, x = v \longrightarrow S \triangleleft L';\, v}\ \text{[R-Asgn]}$$

$$\frac{n \equiv \texttt{true} \implies i = 1 \qquad n \equiv \texttt{false} \implies i = 2}{S;\, \texttt{if (n) \{s}_1\texttt{\} else \{s}_2\texttt{\}} \longrightarrow S;\, s_i}\ \text{[R-Ite]} \qquad \frac{S.X = X',\, L.E}{S;\, \texttt{return}\ v \longrightarrow S \triangleleft X',\, L;\, E\langle v\rangle}\ \text{[R-Ret]}$$

Figure C.2. Operational Semantics for I_{rsc} (adapted from Safe TypeScript [87])

Evaluation Context	$\mathcal{E}$	::=	$\langle\rangle \mid \mathcal{E}.\mathsf{f} \mid \mathcal{E}.\mathsf{m}(\overline{w}) \mid v.\mathsf{m}(\overline{v}, \mathcal{E}, \overline{w})$
		|	$\textbf{new}\ \mathsf{C}(\overline{v}, \mathcal{E}, \overline{w}) \mid \mathcal{E}\ \textbf{as}\ \mathsf{T} \mid \textbf{let}\ \mathsf{x} = \mathcal{E}\ \textbf{in}\ w$
		|	$\mathcal{E}.\mathsf{f} \leftarrow w \mid v.\mathsf{f} \leftarrow \mathcal{E} \mid \textbf{if}\,[\overline{\phi}]\,\mathcal{E}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2$
SSA Eval. Context	$\mathcal{U}$	::=	$\textbf{let}\ \mathsf{x} = \mathcal{E}\ \textbf{in}\ \langle\rangle \mid \textbf{if}\,[\overline{\phi}]\,\mathcal{E}\ \textbf{then}\ \mathsf{u}_1\ \textbf{else}\ \mathsf{u}_2$
Term Eval. Context	$\mathcal{W}$	::=	$\mathcal{E} \mid \mathcal{U}$
Runtime Conf.	$\mathcal{R}$	::=	$\mathcal{S}, w$
State	$\mathcal{S}$	::=	$\mathcal{K}, \mathcal{H}$
Heap	$\mathcal{H}$	::=	$\cdot \mid \ell \mapsto \mathcal{O} \mid \mathcal{H}_1, \mathcal{H}_2$
Value	v	::=	$\ell \mid \mathsf{n}$
Field Bindings	$\vec{\mathcal{F}}$	$\in$	$\mathcal{F} \rightharpoonup \mathsf{Vals}$
Object	$\mathcal{O}$	::=	$\{\texttt{proto}: \ell;\ \mathsf{f}: \vec{\mathcal{F}};\ \ldots\}$
		|	$\{\texttt{name}: \mathsf{C};\ \texttt{proto}: \ell;\ \mathsf{m}: \mathcal{M}\}$

Figure C.3. Syntax and Runtime Configuration for λ_{rsc}

Operational Semantics for λ_{rsc} $\boxed{\mathcal{S}; w \longrightarrow \mathcal{S}'; w'}$

$$\frac{\mathcal{S}; w \longrightarrow \mathcal{S}'; w'}{\mathcal{S}; \mathcal{E}\langle w\rangle \longrightarrow \mathcal{S}'; \mathcal{E}\langle w'\rangle}\ [\text{RC-ECTX}] \qquad \frac{\mathcal{S}.\mathcal{H}(\ell) = \{\texttt{proto}: \ell';\ \mathsf{f}: \vec{\mathcal{F}};\ \ldots\} \qquad \mathsf{f} := v \in \vec{\mathcal{F}}}{\mathcal{S}; \ell.\mathsf{f} \longrightarrow \mathcal{S}; v}\ [\text{R-FIELD}]$$

$$\frac{\mathsf{resolveMethod}(\mathcal{H}, \ell) = (\textbf{def}\ \mathsf{m}(\overline{\mathsf{x}}) = w)}{\mathcal{S}; \ell.\mathsf{m}(\overline{v}) \longrightarrow \mathcal{S}; [\overline{v}/\overline{\mathsf{x}}, \ell/\texttt{this}]\,(w)}\ [\text{R-CALL}] \qquad \frac{\Gamma \vdash \mathcal{S}.\mathcal{H}(\ell): \mathsf{T}'; \mathsf{T}' \leq \mathsf{T}}{\mathcal{S}; \ell\ \textbf{as}\ \mathsf{T} \longrightarrow \mathcal{S}; \ell}\ [\text{R-CAST}]$$

$$\frac{\mathsf{H}(\ell_0) = \{\texttt{name}: \mathsf{C};\ \texttt{proto}: \ell_0';\ \mathsf{m}: \mathcal{M}\} \qquad \mathsf{fields}(\mathcal{K}, \mathsf{C}) = \overline{\mathsf{f}} := \overline{\mathsf{T}} \qquad \mathcal{O} = \{\texttt{proto}: \ell_0;\ \overline{\mathsf{f}}: \overline{v}\} \qquad \mathcal{H}' = \mathcal{H}[\ell \mapsto \mathcal{O}] \qquad \ell\ \mathsf{fresh}}{\mathcal{K}, \mathcal{H}; \textbf{new}\ \mathsf{C}(\overline{v}) \longrightarrow \mathcal{K}, \mathcal{H}'; \ell}\ [\text{R-NEW}]$$

$$\frac{}{\mathcal{S}; \textbf{let}\ \mathsf{x} = v\ \textbf{in}\ w \longrightarrow \mathcal{S}; [v/\mathsf{x}]\,(w)}\ [\text{R-LETIN}] \qquad \frac{\mathcal{H}' = \mathcal{S}.\mathcal{H}[\ell \mapsto \mathcal{S}.\mathcal{H}(\ell)[\mathsf{f} \mapsto v]]}{\mathcal{S}; \ell.\mathsf{f} \leftarrow v \longrightarrow \mathcal{S} \triangleleft \mathcal{H}'; v}\ [\text{R-DOTASGN}]$$

$$\frac{\mathsf{n} \equiv \textbf{true} \implies \mathsf{i} = 1 \qquad \mathsf{n} \equiv \textbf{false} \implies \mathsf{i} = 2}{\mathcal{S}; \textbf{if}\,[\overline{\mathsf{x}}, \overline{\mathsf{x}}_1, \overline{\mathsf{x}}_2]\,\mathsf{n}\ \textbf{then}\ \mathsf{u}_1\ \textbf{else}\ \mathsf{u}_2 \longrightarrow \mathcal{S}; \mathsf{u}_\mathsf{i}\langle[\overline{\mathsf{x}_\mathsf{i}}/\overline{\mathsf{x}}]\ \langle\rangle\rangle}\ [\text{R-LIF}]$$

Figure C.4. Reduction Rules for λ_{rsc}

C.1.2 SSA Transformation

Section 4.2.3 describes the SSA transformation from I_{rsc} to λ_{rsc}. This section provides more details and extends the transformation to runtime configurations, to enable the statement and proof of our consistency theorem.

Static Tranformation

Figure C.5 includes some additional transformation rules that supplement the rules of Figure 4.6. The main program transformation judgment is:

$$P \hookrightarrow \mathcal{P} \triangleright \Delta$$

A global SSA enviornment Δ is the result of the translation of the entire program P to $\mathcal{P}$. In particular, in a program translation tree:

- each expression node introduces a single binding to the relevant SSA environment

$$\delta \vdash e \hookrightarrow w \qquad \text{produces binding} \qquad e \mapsto \delta$$

- each statement introduces two bindings, one for the input environment and one for the output (we use the notation $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$, respectively):

$$\delta_0 \vdash s \hookrightarrow u \dashv \delta_1 \qquad \text{produces bindings} \qquad \lceil s \rceil \mapsto \delta_0 \qquad \lfloor s \rfloor \mapsto \delta_1$$

We assume all AST nodes are uniquely identified.

Runtime Configuration Tranformation

Figures C.6, C.7, C.8, C.9 and C.10 include rules for translating runtime configurations. The main judgment is of the form:

$$S,\ M \overset{\Delta}{\hookrightarrow} \mathcal{S},\ e$$

This assumes that the program containing expression (or body) M was SSA-translated producing a global SSA environment Δ. Rule S-EXP-RTCONF translates a term M under a state S. This process gets factored into the translation of:

- the signatures S.K, which is straight-forward (same as in static translation),
- the heap S.H, which is described in Figure C.10, and
- term M under a local store S.L and a stack S.X.

The last part breaks down into rules that expose the structure of the stack. Rule S-STACK-E translates configurations involving an empty stack, which are delegated to the judgment

Program Translation Rules $\boxed{P \hookrightarrow \mathcal{P} \triangleright \Delta}$

$$\frac{K \hookrightarrow \mathcal{K} \triangleright \Delta_1 \qquad \cdot \vdash B \hookrightarrow e \triangleright \Delta_2}{K,\, B \hookrightarrow \mathcal{K},\, e \triangleright \Delta_1 \cup \Delta_2}\ \text{[S-PROG]}$$

Signature Translation $\boxed{K \hookrightarrow \mathcal{K}}$

$$\frac{}{\cdot \hookrightarrow \cdot}\ \text{[S-SIGS-EMP]} \qquad \frac{F \hookrightarrow \mathcal{F} \qquad M \hookrightarrow \mathcal{M}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{F,\, M\} \hookrightarrow \textbf{class}\ C\ \textbf{extends}\ D\ \{\mathcal{F},\, \mathcal{M}\}}\ \text{[S-SIGS-BND]}$$

$$\frac{K_1 \hookrightarrow \mathcal{K}_1 \qquad K_2 \hookrightarrow \mathcal{K}_2}{K_1,\, K_2 \hookrightarrow \mathcal{K}_1,\, \mathcal{K}_2}\ \text{[S-SIGS-CONS]}$$

Expression and Statement Translations (selected) $\boxed{\delta \vdash e \hookrightarrow w}$ $\boxed{\delta \vdash s \hookrightarrow u \dashv \delta'}$

$$\frac{}{n \hookrightarrow n}\ \text{[S-CONST]} \qquad \frac{\delta \vdash e \hookrightarrow e \qquad \delta \vdash e_i \hookrightarrow \overline{e}_i \qquad m\ \text{fresh}}{\delta \vdash e.m(\overline{e}_i) \hookrightarrow e.m(\overline{e}_i)}\ \text{[S-CALL]}$$

Figure C.5. Additional SSA Transformation Rules in RSC

$L,\, M \xhookrightarrow{H,\Delta} e$, and rule S-STACK-C separately translates the top of the stack and the rest of the stack frames, and then composes them into a single target expression.

Finally, judgments of the forms $L;\, X;\, M \xhookrightarrow{H,\Delta} e$ and $L;\, X;\, E \xhookrightarrow{H,\Delta} \mathcal{W}$ translate expressions and statements under a local store L. The rules here are similar to their static counterparts. The *key difference* stems from the fact that in λ_{rsc} variable are replaced with the respective values as soon as they come into scope. On the contrary, in I_{rsc} variables are only instantiated with the matching (in the store) value when they get into an evaluation position. To wit, rule SR-VARREF performs the necessary substitution θ on the translated variable, which we calculate though the meta-function toSubst, defined as follows:

$$\mathsf{toSubst}(\delta, L, H) \quad \doteq \quad \begin{cases} \{[v/z] \mid x \mapsto z \in \delta,\, x \mapsto v \in L,\, H; v \hookrightarrow v\} & \text{if } \mathsf{dom}(\delta) = \mathsf{dom}(L) \\ \mathsf{impossible} & \text{otherwise} \end{cases}$$

C.1.3 Object Constraint System

Our system leverages the idea introduced in the formall core of X10 [76] to extend a base constraint system $\mathcal{C}$ with a larger constraint system $\mathcal{O}(\mathcal{C})$, built on top of $\mathcal{C}$. The original system $\mathcal{C}$ comprises formulas taken from a decidable SMT logic [74], including, for example,

Runtime Configuration Translation Rules $\boxed{S,\, M \overset{\Delta}{\hookrightarrow} \mathcal{S},\, e}$ $\boxed{S,\, s \overset{\Delta}{\hookrightarrow} \mathcal{S},\, u}$

$$\frac{S.K \overset{\Delta}{\hookrightarrow} \mathcal{K} \qquad S; S.H \hookrightarrow \mathcal{H} \qquad S.L;\, S.X;\, M \overset{S.H,\Delta}{\longleftarrow\!\!\shortmid} e}{S,\, M \overset{\Delta}{\hookrightarrow} \mathcal{K},\, \mathcal{H},\, e}\ [\text{S-Exp-RtConf}]$$

$$\frac{S.K \overset{\Delta}{\hookrightarrow} \mathcal{K} \qquad S; S.H \hookrightarrow \mathcal{H} \qquad S.L;\, S.X;\, s \overset{S.H,\Delta}{\longleftarrow\!\!\shortmid} u}{S,\, s \overset{\Delta}{\hookrightarrow} \mathcal{K},\, \mathcal{H},\, u}\ [\text{S-Stmt-RtConf}]$$

Figure C.6. Runtime Configuration Translation in RSC

Runtime Stack Translation Rules $\boxed{L;\, X;\, M \overset{H,\Delta}{\longleftarrow\!\!\shortmid} e}$ $\boxed{L;\, X;\, E \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{W}}$

$$\frac{L,\, M \overset{H,\Delta}{\longleftarrow\!\!\shortmid} e}{L;\, \cdot;\, M \overset{H,\Delta}{\longleftarrow\!\!\shortmid} e}\ [\text{S-Stack-E}] \qquad \frac{L; E \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{W}}{L;\, \cdot;\, E \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{W}}\ [\text{S-Ec-Stack-E}]$$

$$\frac{L_0;\, \cdot;\, M \overset{H,\Delta}{\longleftarrow\!\!\shortmid} e_0 \qquad L;\, X;\, E \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{E}}{L_0;\, (X,\, L.E);\, M \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{E}\langle e_0 \rangle}\ [\text{S-Stack-C}]$$

$$\frac{L_0;\, \cdot;\, E_0 \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{W}_0 \qquad L;\, X;\, E \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{E}}{L_0;\, (X,\, L.E);\, E_0 \overset{H,\Delta}{\longleftarrow\!\!\shortmid} \mathcal{E}\langle \mathcal{W}_0 \rangle}\ [\text{S-Ec-Stack-C}]$$

Figure C.7. Runtime Stack Translation in RSC

linear arithmetic constraints and uninterpreted predicates. The Object Constraint System $\mathcal{O}(\mathcal{C})$ introduces the constraints:

- $\mathsf{class}(C)$, which it true for all classes C defined in the program;
- x hasImm f, to denote that the *immutable* field f is accessible from variable x;
- x hasMut f, to denote that the *mutable* field f is accessible from variable x; and
- $\mathsf{fields}(x) = F$, to expose all fields available to x.

Figure C.11 shows the constraint system as ported from CFG [76]. We refer the reader to that work for details. The main differences are syntactic changes to account for our notion of *strengthening*. Also the SC-Field rule accounts now for both immutable and mutable fields. The

Runtime Term Translation Rules (selected) $\boxed{M \overset{\Delta}{\hookrightarrow} \mathcal{M}}$ $\boxed{L,\, M \overset{H,\Delta}{\longleftarrow} e}$ $\boxed{L,\, s \overset{H,\Delta}{\longleftarrow} u}$

$$\frac{\cdot,\, B \overset{\cdot,\Delta}{\longleftarrow} e}{m(\overline{x}) : B \overset{\Delta}{\hookrightarrow} \textbf{def}\ m(\overline{x}) = e}\ \text{[SR-Meth]} \qquad \frac{H; v \hookrightarrow v}{L,\, v \overset{H,\Delta}{\longleftarrow} v}\ \text{[SR-Val]}$$

$$\frac{\Delta(x) \vdash x \hookrightarrow x \qquad \theta = \mathsf{toSubst}(\Delta(x), L, H)}{L,\, x \overset{H,\Delta}{\longleftarrow} \theta\,(x)}\ \text{[SR-VarRef]} \qquad \frac{L,\, e \overset{H,\Delta}{\longleftarrow} e \qquad L,\, \overline{e} \overset{H,\Delta}{\longleftarrow} \overline{e}}{L,\, e\,.\,m(\overline{e}) \overset{H,\Delta}{\longleftarrow} e.m(\overline{e})}\ \text{[SR-Call]}$$

$$\frac{L,\, s \overset{H,\Delta}{\longleftarrow} u \qquad \Delta' = \Delta[\, e \mapsto \Delta\lfloor s \rfloor\,] \qquad L,\, e \overset{H,\Delta'}{\longleftarrow} e}{L,\, s\,;\, \texttt{return}\ e \overset{H,\Delta}{\longleftarrow} u\langle e \rangle}\ \text{[SR-Body]}$$

$$\frac{x \mapsto x \in \Delta\lfloor \texttt{var}\ x = e \rfloor \qquad L,\, e \overset{H,\Delta}{\longleftarrow} e}{L,\, \texttt{var}\ x = e \overset{H,\Delta}{\longleftarrow} \textbf{let}\ x = w\ \textbf{in}\ \langle\,\rangle}\ \text{[SR-VarDecl]}$$

$$\frac{L,\, e \overset{H,\Delta}{\longleftarrow} e \qquad L,\, s_1 \overset{H,\Delta}{\longleftarrow} u_1 \qquad L,\, s_2 \overset{H,\Delta}{\longleftarrow} u_2 \qquad (\overline{x}; \overline{x}_1; \overline{x}_2) = \Delta\lfloor s_1 \rfloor \bowtie \Delta\lfloor s_2 \rfloor \qquad \overline{x} = \Delta\lfloor \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \rfloor(\overline{x})}{L,\, \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \overset{H,\Delta}{\longleftarrow} \textbf{if}\,[\,\overline{x}, \overline{x}_1, \overline{x}_2\,]\, w\ \textbf{then}\ u_1\ \textbf{else}\ u_2}\ \text{[SR-Ite]}$$

$$\frac{x \mapsto x \in \Delta\lfloor x = e \rfloor \qquad L,\, e \overset{H,\Delta}{\longleftarrow} e}{L,\, x = e \overset{H,\Delta}{\longleftarrow} \textbf{let}\ x = w\ \textbf{in}\ \langle\,\rangle}\ \text{[SR-Asgn]}$$

Figure C.8. Runtime Term Translation in RSC

main judgment here is of the form:

$$\Gamma \vdash_{\mathcal{K}} P$$

where $\mathcal{K}$ is the set of classes defined in the program. Substitutions and strengthening operations on field declarations are performed on the types of the declared fields (*e.g.* SC-Field-I, SC-Field-C).

C.1.4 Well-formedness Constraints

The well-formedness rules for predicates, terms, types and heaps can be found in Figure C.12. The majority of these rules are routine.

The judgment for term well-formedness assigns a *sort* to each term t, which can be

Evaluation Context Translation Rules (selected) $\boxed{\mathsf{L}; \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{W}}$

$$\frac{}{\mathsf{L}; \langle\rangle \overset{\mathsf{H},\Delta}{\longleftarrow} \langle\rangle} \qquad \frac{\mathsf{L}; \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{E} \qquad \mathsf{f}\ \text{fresh}}{\mathsf{L}; \mathsf{E}.\mathsf{f} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{E}.\mathsf{f}} \qquad \frac{\mathsf{L}; \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{E} \qquad \mathsf{m}\ \text{fresh} \qquad \mathsf{L}; \cdot; \overline{e} \overset{\mathsf{H},\Delta}{\longleftarrow} \overline{e}}{\mathsf{L}; \mathsf{E}.\mathsf{m}(\overline{e}) \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{E}.\mathsf{m}(\overline{e})}$$

$$\frac{\mathsf{L}; \cdot; \mathsf{x} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathsf{x} \qquad \mathsf{L}; \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{E}}{\mathsf{L}; \mathtt{var}\ \mathsf{x} = \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \textbf{let}\ \mathsf{x} = \mathcal{E}\ \textbf{in}\ \langle\,\rangle} \qquad \frac{\mathsf{L}; \mathsf{E} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{U} \qquad \mathsf{L}; \mathsf{s} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathsf{u}}{\mathsf{L}; \mathsf{E}\,; \mathsf{s} \overset{\mathsf{H},\Delta}{\longleftarrow} \mathcal{U}\langle \mathsf{u} \rangle}$$

Figure C.9. Evaluation Context Translation Rules in RSC

thought of as a base type. The judgment $\Gamma \vdash_{\mathsf{q}} \overline{\mathsf{t}}$ is used as a shortcut for any further constraints that the f operator might impose on its arguments $\overline{\mathsf{t}}$. For example if f is the equality operator then the two arguments are required to have types that are related via subtyping, *i.e.* if $\mathsf{t}_1 : \mathsf{N}_1$ and $\mathsf{t}_2 : \mathsf{N}_2$, it needs to be the case that $\mathsf{N}_1 \leq \mathsf{N}_2$ or $\mathsf{N}_2 \leq \mathsf{N}_1$.

Type well-formedness is typical among similar refinement types [65].

C.1.5 Subtyping

Figure C.13 presents the full set of sybtyping rules, which borrows ideas from similar systems [65, 91].

Heap Translation Rules $\boxed{S; H \hookrightarrow \mathcal{H}}$

$$\frac{}{S; \cdot \hookrightarrow \cdot}\ \text{[S-HEAP-EMP]} \qquad \frac{S.H; O \hookrightarrow \mathcal{O} \qquad \ell \text{ fresh}}{S; (\ell \mapsto O) \hookrightarrow (\ell \mapsto \mathcal{O})}\ \text{[S-HEAP-BND]}$$

$$\frac{S; H_1 \hookrightarrow \mathcal{H}_1 \qquad S; H_2 \hookrightarrow \mathcal{H}_2}{S; (H_1, H_2) \hookrightarrow \mathcal{H}_1, \mathcal{H}_2}\ \text{[S-HEAP-CONS]}$$

Value Translation Rules $\boxed{H; v \hookrightarrow v}$

$$\frac{\ell \mapsto O \in H \qquad H; (\ell \mapsto O) \hookrightarrow (\ell \mapsto \mathcal{O})}{H; \ell \hookrightarrow \ell}\ \text{[S-LOC]} \qquad \frac{}{H; n \hookrightarrow n}\ \text{[S-CONST]}$$

Heap Object Translation Rules $\boxed{H; O \hookrightarrow \mathcal{O}}$

$$\frac{H; \ell \hookrightarrow \ell \qquad H; F \hookrightarrow \vec{\mathcal{F}}}{H; \{\texttt{proto}: \ell;\ \texttt{f}: \vec{F};\ \ldots\} \hookrightarrow \{\texttt{proto}: \ell;\ \texttt{f}: \vec{\mathcal{F}};\ \ldots\}}$$

$$\frac{H; \ell \hookrightarrow \ell \qquad M \hookrightarrow \mathcal{M}}{H; \{\texttt{name}: C;\ \texttt{proto}: \ell;\ \texttt{m}: M\} \hookrightarrow \{\texttt{name}: C;\ \texttt{proto}: \ell;\ \texttt{m}: \mathcal{M}\}}$$

Figure C.10. Heap and Value Translation Rules in RSC

Structural Constraints $\boxed{\Gamma \vdash_{\mathcal{K}} P}$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\mathcal{F}, \mathcal{M}\} \in \mathcal{K}}{\Gamma \vdash_{\mathcal{K}} \mathsf{class}(C)}\ [\text{SC-CLASS}] \qquad \frac{\Gamma \vdash_{\mathcal{K}} x\colon C, \mathsf{class}(C)}{\Gamma \vdash_{\mathcal{K}} \mathsf{inv}(C, x)}\ [\text{SC-INV}]$$

$$\frac{\Gamma \vdash_{\mathcal{K}} \mathsf{fields}(x) = \circ\, \overline{f_i}\colon \overline{T_i}, \square\, \overline{g_i}\colon \overline{T_i'}}{\begin{array}{l}\Gamma \vdash_{\mathcal{K}} x\ \mathsf{hasImm}\ f_i\colon T_i \\ \Gamma \vdash_{\mathcal{K}} x\ \mathsf{hasMut}\ g_i\colon T_i'\end{array}}\ [\text{SC-FIELD}] \qquad \frac{}{x\colon \texttt{Object} \vdash_{\mathcal{K}} \mathsf{fields}(x) = \varnothing}\ [\text{SC-OBJECT}]$$

$$\frac{\Gamma, x\colon D \vdash_{\mathcal{K}} \mathsf{fields}(x) = \widehat{F} \qquad C \triangleleft D :: P \left\{\widehat{F}', \widehat{M}'\right\} \in \mathcal{K}}{\Gamma, x\colon C \vdash_{\mathcal{K}} \mathsf{fields}(x) = \widehat{F}, [x/\mathbf{this}]\, \widehat{F}'}\ [\text{SC-FIELD-I}]$$

$$\frac{\Gamma, x\colon C \vdash_{\mathcal{K}} \mathsf{fields}(x) = \widehat{F}}{\Gamma, x\colon \{\nu\colon C \mid P\} \vdash_{\mathcal{K}} \mathsf{fields}(x) = \mathcal{F} \uplus [x/\nu]\, P}\ [\text{SC-FIELD-C}]$$

$$\frac{\Gamma \vdash_{\mathcal{K}} \mathsf{class}(C) \qquad \theta = [x/\mathbf{this}] \qquad m(\overline{x}\colon \overline{T})\colon T \in \widehat{K}}{\Gamma, x\colon C \vdash_{\mathcal{K}} x\ \mathsf{has}\ (m(\overline{x}\colon \overline{\theta\, T})\colon \theta\, T)}\ [\text{SC-METH-B}]$$

$$\frac{\Gamma, x\colon D \vdash_{\mathcal{K}} x\ \mathsf{has}\ (m(\overline{x}\colon \overline{T})\colon T) \qquad C \triangleleft D :: P \left\{\widehat{F}, \widehat{M}\right\} \in \mathcal{K} \qquad m \notin \widehat{M}}{\Gamma, x\colon C \vdash_{\mathcal{K}} x\ \mathsf{has}\ (m(\overline{x}\colon \overline{T})\colon T)}\ [\text{SC-METH-I}]$$

$$\frac{\Gamma, x\colon C \vdash_{\mathcal{K}} x\ \mathsf{has}\ (m(\overline{x}\colon \overline{T})\colon T)}{\Gamma, x\colon \{\nu\colon C \mid P\} \vdash_{\mathcal{K}} x\ \mathsf{has}\ (m(\overline{x}\colon \overline{T})\colon T \uplus [x/\mathbf{this}]\, P)}\ [\text{SC-METH-C}]$$

Figure C.11. Structural Constraints in RSC (adapted from [76])

Well-Formed Predicates $\boxed{\Gamma \vdash P}$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}\ [\text{WP-AND}] \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \neg P}\ [\text{WP-NOT}] \qquad \frac{\Gamma \vdash t : \mathsf{Bool}}{\Gamma \vdash t}\ [\text{WP-TERM}]$$

Well-Formed Terms $\boxed{\Gamma \vdash t : N}$

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x : \lfloor T \rfloor}\ [\text{WF-VAR}] \qquad \frac{}{\Gamma \vdash n : \lfloor b_n \rfloor}\ [\text{WF-CONST}]$$

$$\frac{\Gamma \vdash t : N \qquad \Gamma, x\colon N \vdash x\ \mathsf{hasImm}\ f_i : T_i}{\Gamma \vdash t.f_i : \lfloor T_i \rfloor}\ [\text{WF-FIELD}] \qquad \frac{\Gamma \vdash f : \overline{N} \to N' \qquad \Gamma \vdash_q \bar{t}}{\Gamma \vdash f(\bar{t}) : N'}\ [\text{WF-FUN}]$$

Well-Formed Types $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma, v\colon N \vdash P}{\Gamma \vdash \{v\colon N \mid P\}}\ [\text{WT-BASE}] \qquad \frac{\Gamma \vdash T_1 \qquad \Gamma, x\colon T_1 \vdash T_2}{\Gamma \vdash \exists x\colon T_1 . T_2}\ [\text{WT-EXISTS}]$$

Well-Formed Heaps $\boxed{\Sigma \vdash \mathcal{H}}$

$$\frac{}{\Sigma \vdash \cdot}\ [\text{WF-HEAP-EMP}]$$

$$\frac{\begin{array}{c}\mathcal{O} \doteq \{\mathtt{proto}\colon \ell'; \mathtt{f}\colon \vec{\mathcal{F}}; \ldots\} \qquad \vec{\mathcal{F}} \doteq \circ\,\bar{f} := \bar{v}_\circ, \square\,\bar{g} := \bar{v}_\square \qquad \lfloor \Sigma(\ell) \rfloor = C \\ \Gamma, y\colon C \vdash \mathsf{fields}(y) = \circ\,\bar{f}\colon \overline{T'}, \square\,\bar{g}\colon \overline{T''} \qquad \Sigma \vdash \bar{v}_\circ : \bar{T}_\circ \qquad \Sigma \vdash \bar{v}_\square : \bar{T}_\square \\ \Gamma, y\colon C, \bar{y}_\circ\colon \mathsf{sngl}\left(\bar{T}_\circ, y.\vec{f}\right) \vdash \bar{T}_\circ \le \overline{T'}, \bar{T}_\square \le \overline{T''}, \mathsf{inv}(C, y)\end{array}}{\Sigma \vdash \ell \mapsto \mathcal{O}}\ [\text{WF-HEAP-INST}]$$

$$\frac{\Sigma \vdash \mathcal{H}_1 \qquad \Sigma \vdash \mathcal{H}_2}{\Sigma \vdash \mathcal{H}_1, \mathcal{H}_2}\ [\text{WF-HEAP-CONS}]$$

Figure C.12. Well-Formedness Rules in RSC

Subtyping $\boxed{\Gamma \vdash \mathsf{T} \leq \mathsf{T}'}$

$$\frac{}{\Gamma \vdash \mathsf{T} \leq \mathsf{T}} \text{ [Sub-Refl]} \qquad \frac{\Gamma \vdash \mathsf{T}_1 \leq \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{T}_2 \leq \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 \leq \mathsf{T}_3} \text{ [Sub-Trans]}$$

$$\frac{\mathsf{C} \triangleleft \mathsf{D} :: \mathsf{P}\left\{\widehat{\mathsf{F}}, \widehat{\mathsf{M}}\right\}}{\Gamma \vdash \mathsf{C} \leq \mathsf{D}} \text{ [Sub-Extends]} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{N} \leq \mathsf{N}' \\ \mathsf{Valid}([\![\Gamma]\!] \Rightarrow ([\![\mathsf{P}]\!] \Rightarrow [\![\mathsf{P}']\!]))\end{array}}{\Gamma \vdash \{\nu\colon \mathsf{N} \mid \mathsf{P}\} \leq \{\nu\colon \mathsf{N}' \mid \mathsf{P}'\}} \text{ [Sub-Base]}$$

$$\frac{\Gamma \vdash e : \mathsf{T} \qquad \Gamma \vdash \mathsf{T}_1 \leq [e/x](\mathsf{T}_2)}{\Gamma \vdash \mathsf{T}_1 \leq \exists x\colon \mathsf{T}.\, \mathsf{T}_2} \text{ [Sub-Witness]} \qquad \frac{\Gamma, x\colon \mathsf{T} \vdash \mathsf{T}_1 \leq \mathsf{T}_2 \qquad x \notin FV(\mathsf{T}_2)}{\Gamma \vdash \exists x\colon \mathsf{T}.\, \mathsf{T}_1 \leq \mathsf{T}_2} \text{ [Sub-Bind]}$$

Figure C.13. Subtyping Rules in RSC

Runtime Typing Rules $\boxed{\Sigma \vdash \nu : \mathsf{T}}$ $\boxed{\Sigma \vdash_{\mathcal{H}} \mathcal{O} : \mathsf{T}}$

$$\frac{\Sigma(\ell) = \mathsf{T}}{\Sigma \vdash \ell : \mathsf{T}} \text{ [RT-T-Loc]} \qquad \frac{}{\Sigma \vdash \nu : \mathsf{b}_n} \text{ [RT-T-Const]}$$

$$\frac{\lfloor \Sigma(\ell) \rfloor = \mathsf{C} \qquad \mathsf{fieldDefs}(\mathcal{H}, \ell) = \circ\, \overline{\mathsf{f}} := \overline{\nu}_\circ, \square\, \overline{\mathsf{g}} := \overline{\nu}_\square \qquad \Sigma \vdash \overline{\nu}_\circ : \overline{\mathsf{T}}_\circ}{\Sigma \vdash_{\mathcal{H}} \{\mathtt{proto}\colon \ell;\ \mathtt{f}\colon \vec{\mathcal{F}};\ \ldots\} : \exists \overline{\mathsf{y}}_\circ\colon \overline{\mathsf{T}}_\circ.\{\nu\colon \mathsf{C} \mid \nu.\overline{\mathsf{f}} = \overline{\mathsf{y}}_\circ \wedge \mathsf{inv}(\mathsf{C}, \nu)\}} \text{ [RT-T-Obj]}$$

Figure C.14. Typing Runtime Configurations for λ_{rsc}

C.2 Proofs

The main results in this section are:

- Program Consistency Lemma (Lemma C.13, page 213)
- Forward Simulation Theorem (Theorem C.14, page 218)
- Subject Reduction Theorem (Theorem C.27, page 220)
- Progress Theorem (Theorem C.28, page 231)

C.2.1 SSA Translation

Definition C.2.1 (Environment Substitution)**.**

$$[\delta_1 / \delta_2] \doteq [\overline{x}_1 / \overline{x}_2] \quad \textit{where} \quad (\overline{x}; \overline{x}_1; \overline{x}_2) = \delta_1 \bowtie \delta_2$$

Definition C.2.2 (Valid Configuration)**.**

$$\mathsf{validConf}(S, M) \doteq \begin{cases} \textit{true} & \textit{if } (S.X = \cdot) \implies \exists\, B \textit{ s.t. } M \equiv B \\ \textit{false} & \textit{otherwise} \end{cases}$$

Assumption C.2.1 (Stack Form)**.** *Let stack* $X = X_0$, L.E. *Evaluation context* E *is of one of the following forms:*

(a) E_0 ; `return` e

(b) `return` E_0

Lemma C.1 (Global Environment Substitution)**.** *If* $L, e \xhookrightarrow{H,\Delta} e$, *then* $L, e \xhookrightarrow{H,\Delta'} [\Delta'(e)/\Delta(e)]\,(e)$

Lemma C.2 (Evaluation Context)**.** *If*

$$L, M \xhookrightarrow{H,\Delta} \mathcal{E}\langle e \rangle$$

then there exist E *and* e *s.t.:*

(a) $M \equiv E\langle e \rangle$

(b) $L; E \xhookrightarrow{H,\Delta} \mathcal{E}$

(c) $L, e \xhookrightarrow{H,\Delta} e$

Proof. By induction on the derivation of the input transformation. □

Lemma C.3 (Translation under Store)**.** *If*

$$\cdot, B \xhookrightarrow{\cdot,\Delta} e$$

then

$$L,\ B \xhookrightarrow{H,\Delta} \theta\,(e)$$

where $\theta = \mathsf{toSubst}(\Delta(B), L, H)$.

Proof. By induction on the structure of the input translation. □

Lemma C.4 (Canonical Forms)**.**

(a) If $L,\ M \xhookrightarrow{H,\Delta} n$, *then* $M \equiv n$

(b) If $L,\ M \xhookrightarrow{H,\Delta} \ell.m(\overline{v})$, *then* $M \equiv \ell\,.\,m(\overline{v})$

(c) If $L,\ M \xhookrightarrow{H,\Delta}$ **if** $[\,\overline{\phi}\,]\ e$ **then** u_1 **else** u_2, *then* $M \equiv$ `if (e) {s1} else {s2}`

(d) If $M \hookrightarrow$ **def** $m(\overline{x}) = e_0$, *then* $M \equiv m(\overline{x}) : B$

Lemma C.5 (Translation Closed under Evaluation Context Composition)**.** *If*

(a) $L; E_0 \xhookrightarrow{H,\Delta} \mathcal{E}_0$

(b) $L'; (L,\ E_1);\ B \xhookrightarrow{H,\Delta} e$

then $L'; (L,\ E_0\langle E_1\rangle);\ B \xhookrightarrow{H,\Delta} \mathcal{E}_0\langle e\rangle$

Lemma C.6 (Heap and Store Weakening)**.** *If*

$$L;\ X;\ E \xhookrightarrow{H,\Delta} \mathcal{W}$$

then $\forall\ H', L'$ *s.t.* $H' \supseteq H$ *and* $L' \supseteq L$, *it holds that* $L';\ X;\ E \xhookrightarrow{H',\Delta} \mathcal{W}$

Lemma C.7 (Translation Closed under Stack Extension)**.** *If*

(a) $L_0;\ X_0;\ E_0 \xhookrightarrow{H,\Delta} \mathcal{E}_0$

(b) $L_1;\ X_1;\ B_1 \xhookrightarrow{H,\Delta} e_1$

then $L_1;\ (X_0,\ L_0.E_0,\ X_1);\ B_1 \xhookrightarrow{H,\Delta} \mathcal{E}_0\langle e_1\rangle$

Proof. We proceed by induction on the structure of derivation (b):

- S-STACK-E: Fact (b) has the form:

$$L_1;\ \cdot;\ B_1 \xhookrightarrow{H,\Delta} e_1 \tag{C.7.1}$$

  By applying Rule S-STACK-C on C.7.1 and (a):

$$L_1;\ (X_0,\ L_0.E_0);\ B_1 \xhookrightarrow{H,\Delta} \mathcal{E}_0\langle e_1\rangle \tag{C.7.2}$$

  Which proves the wanted result.

- S-STACK-C: Fact (b) has the form:

$$\mathsf{L}_1;\ (\mathsf{X},\ \mathsf{L}.\mathsf{E});\ \mathsf{B}_1 \xhookrightarrow{\mathsf{H},\Delta} \mathcal{E}\langle e_{1.1}\rangle \tag{C.7.3}$$

By inverting Rule S-STACK-C on C.7.3:

$$\mathsf{L}_1;\ \cdot;\ \mathsf{B}_1 \xhookrightarrow{\mathsf{H},\Delta} e_{1.1} \tag{C.7.4}$$

$$\mathsf{L};\ \mathsf{X};\ \mathsf{E} \xhookrightarrow{\mathsf{H},\Delta} \mathcal{E} \tag{C.7.5}$$

By induction hypothesis on (a) and C.7.5 (the lemma can easily be extended to evaluation contexts):

$$\mathsf{L};\ (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0,\ \mathsf{X});\ \mathsf{E} \xhookrightarrow{\mathsf{H},\Delta} \mathcal{E}_0\langle\mathcal{E}\rangle \tag{C.7.6}$$

By applying Rule S-EC-STACK-C on C.7.4 and C.7.6:

$$\mathsf{L}_1;\ (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0,\ \mathsf{X},\ \mathsf{L}.\mathsf{E});\ \mathsf{B}_1 \xhookrightarrow{\mathsf{H},\Delta} \mathcal{E}_0\langle\mathcal{E}\langle e_{1.1}\rangle\rangle \tag{C.7.7}$$

Which proves the wanted result.

□

Lemma C.8 (Translation Closed under Evaluation Context Application)**.** *If*

(i) $\mathsf{L};\ \mathsf{X};\ \mathsf{E} \xhookrightarrow{\mathsf{H},\Delta} \mathcal{W}$

(ii) $\mathsf{L},\ e \xhookrightarrow{\mathsf{H},\Delta} e$

then $\mathsf{L};\ \mathsf{X};\ \mathsf{E}\langle e\rangle \xhookrightarrow{\mathsf{H},\Delta} \mathcal{W}\langle e\rangle$

Proof. By induction on the derivation of (i). □

Lemma C.9 (Method Resolution)**.** *If*

(i) $\mathsf{S};\mathsf{H} \hookrightarrow \mathcal{H}$

(ii) $\mathsf{H};\ell \hookrightarrow \ell$

(iii) $\mathsf{resolveMethod}(\mathcal{H},\ \ell) = \mathcal{M}$

then:

(a) $\mathsf{resolveMethod}(\mathsf{H},\ \ell) = \mathsf{M}$

(b) $\mathsf{M} \hookrightarrow \mathcal{M}$

Lemma C.10 (Value Monotonicity)**.** *If*

(i) $\mathsf{validConf}(S, M)$

(ii) $S, M \overset{\Delta}{\hookrightarrow} \mathcal{S}, v$

then there exist L' *and* M' *s.t.:*

(a) $S; M \longrightarrow^* S'; M'$

(b) $S', M' \overset{\Delta}{\hookrightarrow} \mathcal{S}, v$

(c) $M' \equiv \begin{cases} \texttt{return}\, v & \text{if} \quad M \equiv B \\ v & \text{otherwise} \end{cases}$

(d) If $S.X = \cdot$ *then* $S'.L = S.L$

where $S' \equiv S.K; L'; \cdot; S.H$

Proof. By induction on the structure of the derivation (ii). □

Lemma C.11 (Top-Level Reduction)**.** *If*

$$K; L; X; H;\ M \longrightarrow K; L'; X'; H';\ M'$$

then for a stack X_0 *it holds that:*

$$K; L; (X_0, X); H;\ M \longrightarrow K; L'; (X_0, X'); H';\ M'$$

Proof. By induction on the structure of the input reduction. □

Lemma C.12 (Empty Stack Consistency)**.** *If*

(i) $S, M \overset{\Delta}{\hookrightarrow} \mathcal{S}, e$

(ii) $S.X = \cdot$

(iii) $\mathcal{S}; w \longrightarrow \mathcal{S}'; w'$

then there exist S' *and* M' *s.t.:*

(a) $S; M \longrightarrow^* S'; M'$,

(b) $S', M' \overset{\Delta}{\hookrightarrow} \mathcal{S}', e'$

(c) (A) If $M \equiv E\langle \ell.\, m(\overline{v}) \rangle$ *then:*

(1) $S'.X = S.L.E$

(2) $S'.H = S.H$

(3) $\exists B'$ *s.t.* $M' \equiv B'$

(4) $\mathcal{S}' = \mathcal{S}$

(B) Otherwise:

(1) $S'.X = \cdot$

(2) $S'.H \supseteq S.H$

(3) $S'.L \supseteq S.L$

(4) If $\exists e$ *s.t.* $M \equiv e$ *then* $\exists e'$ *s.t.* $M' \equiv e'$

(5) If $\exists B$ *s.t.* $M \equiv B$ *then* $\exists B'$ *s.t.* $M' \equiv B'$

Proof. Fact (i) has the form:

$$S,\ M \overset{\Delta}{\hookrightarrow} \mathcal{K},\ \mathcal{H},\ e \tag{C.12.1}$$

Because of fact (ii):

$$S \equiv K; L; \cdot; H \tag{C.12.2}$$

By inverting Rule S-EXP-RTCONF on C.12.1:

$$K \overset{\Delta}{\hookrightarrow} \mathcal{K} \tag{C.12.3}$$

$$S; H \hookrightarrow \mathcal{H} \tag{C.12.4}$$

$$L; \cdot;\ M \overset{H,\Delta}{\longhookrightarrow} e \tag{C.12.5}$$

By inverting S-STACK-E on C.12.10:

$$L,\ M \overset{H,\Delta}{\longhookrightarrow} e \tag{C.12.6}$$

Suppose M is a value. By Rules S-CONST and S-LOC, e is also a value: a contradiction because of (iii). Hence:

$$M \text{ is not a value} \tag{C.12.7}$$

We proceed by induction on the structure of reduction (iii):

- RC-ECTX

$$\mathcal{S};\ \mathcal{E}_0\langle e_0\rangle \longrightarrow \mathcal{S}';\ \mathcal{E}_0\langle e_0'\rangle \tag{C.12.8}$$

By inverting RC-ECTX on C.12.8:

$$\mathcal{S};\ e_0 \longrightarrow \mathcal{S}';\ e_0' \tag{C.12.9}$$

Fact C.12.6 is of the form:

$$\mathsf{L},\ \mathsf{M} \overset{\mathsf{H},\Delta}{\longleftarrow\!\!\!\shortmid} \mathcal{E}_0\langle e_0\rangle \tag{C.12.10}$$

By Lemma C.2 on C.12.10:

$$\mathsf{M} \equiv \mathsf{E}_0\langle e_0\rangle \tag{C.12.11}$$

$$\mathsf{L};\mathsf{E}_0 \overset{\mathsf{H},\Delta}{\longleftarrow\!\!\!\shortmid} \mathcal{E}_0 \tag{C.12.12}$$

$$\mathsf{L},\ e_0 \overset{\mathsf{H},\Delta}{\longleftarrow\!\!\!\shortmid} e_0 \tag{C.12.13}$$

By Rule S-STACK-E on C.12.13:

$$\mathsf{L};\ \cdot;\ e_0 \overset{\mathsf{H},\Delta}{\longleftarrow\!\!\!\shortmid} e_0 \tag{C.12.14}$$

By Rule S-EXP-RTCONF on C.12.3, C.12.4 and C.12.14:

$$\mathsf{S},\ e_0 \overset{\Delta}{\hookrightarrow} \mathcal{S},\ e_0 \tag{C.12.15}$$

By induction hypothesis using C.12.15, (ii) and C.12.9:

$$\mathsf{K};\mathsf{L};\cdot;\mathsf{H};\ e_0 \longrightarrow \mathsf{K};\mathsf{L}';\mathsf{X}';\mathsf{H}';\ \mathsf{M}_0' \tag{C.12.16}$$

$$\mathsf{K};\mathsf{L}';\mathsf{X}';\mathsf{H}',\ \mathsf{M}_0' \overset{\Delta}{\hookrightarrow} \mathcal{S}',\ e_0' \tag{C.12.17}$$

We examine cases on the form of e_0:

- Case $e_0 \equiv \mathsf{E}_1\langle \ell.\,\mathfrak{m}(\overline{\nu})\rangle$:

$$\mathsf{X}' = \mathsf{L},\ \mathsf{E}_1 \tag{C.12.18}$$

$$\mathsf{H}' = \mathsf{H} \tag{C.12.19}$$

$$\mathsf{M}_0' = \mathsf{B}' \tag{C.12.20}$$

$$\mathcal{S}' = \mathcal{S} \tag{C.12.21}$$

  For some method body B'. So C.12.17 becomes:

$$\mathsf{K};\mathsf{L}';(\mathsf{L},\ \mathsf{E}_1);\mathsf{H},\ \mathsf{B}' \overset{\Delta}{\hookrightarrow} \mathcal{S},\ e_0' \tag{C.12.22}$$

By inverting rule R-CALL on C.12.16:

$$\mathsf{resolveMethod}(H,\, \ell) = m(\overline{x}) : B' \tag{C.12.23}$$

$$L' = \overline{x} \mapsto \overline{v},\, \mathtt{this} \mapsto \ell \tag{C.12.24}$$

$$X'_0 = L.E_1 \tag{C.12.25}$$

Let:

$$S,\, M \equiv K; L; \cdot; H,\, (E_0\langle E_1\rangle)\langle \ell\,.\, m(\overline{v})\rangle$$

By rule R-CALL using C.12.23, C.12.24 and $X' = L.E_0\langle E_1\rangle$ on S, M:

$$K; L; \cdot; H;\, (E_0\langle E_1\rangle)\langle \ell\,.\, m(\overline{v})\rangle \longrightarrow K; L'; (L,\, E_0\langle E_1\rangle); H;\, B' \tag{C.12.26}$$

Which proves (a). By inverting Rule S-EXP-RTCONF on C.12.22:

$$S'; H \hookrightarrow \mathcal{H} \tag{C.12.27}$$

$$L';\, (L,\, E_1);\, B' \overset{H,\Delta}{\hookleftarrow} e'_0 \tag{C.12.28}$$

From Lemma C.5 on C.12.12 and C.12.28:

$$L';\, (L,\, E_0\langle E_1\rangle);\, B' \overset{H,\Delta}{\hookleftarrow} \mathcal{E}_0\langle e'_0\rangle \tag{C.12.29}$$

By Rule S-EXP-RTCONF using C.12.3, C.12.27 and C.12.29:

$$K; L'; (L,\, E_0\langle E_1\rangle); H,\, B' \overset{\Delta}{\hookrightarrow} \mathcal{S},\, \mathcal{E}_0\langle e'_0\rangle \tag{C.12.30}$$

Which proves (b). By C.12.11 and the current case:

$$M \equiv (E_0\langle E_1\rangle)\langle \ell\,.\, m(\overline{v})\rangle \tag{C.12.31}$$

By C.12.26 and C.12.30:

$$S'.X = L,\, E_0\langle E_1\rangle \tag{C.12.32}$$

$$M' = B' \tag{C.12.33}$$

$$\mathcal{S}' = \mathcal{S} \tag{C.12.34}$$

By C.12.32, C.12.19, C.12.33 and C.12.34 we prove (c).

- All remaining cases:

$$X' \equiv \cdot \tag{C.12.35}$$
$$H' \supseteq H \tag{C.12.36}$$
$$L' \supseteq L \tag{C.12.37}$$
$$M'_0 \equiv e'_0 \tag{C.12.38}$$

So C.12.16 and C.12.17 become:

$$K; L; \cdot; H;\ e_0 \longrightarrow K; L'; \cdot; H';\ e'_0 \tag{C.12.39}$$
$$K; L'; \cdot; H',\ e'_0 \overset{\Delta}{\hookrightarrow} \mathcal{S}',\ e'_0 \tag{C.12.40}$$

By Rule R-EVALCTX using C.12.39:

$$K; L; \cdot; H;\ E_0\langle e_0\rangle \longrightarrow K; L'; \cdot; H';\ E_0\langle e'_0\rangle \tag{C.12.41}$$

Which proves (a) and (c). By inverting Rules S-EXP-RTCONF and S-STACK-E on C.12.40:

$$L',\ e_0 \xhookrightarrow{H',\Delta} e_0 \tag{C.12.42}$$

From Lemma C.6 using C.12.12, C.12.36 and C.12.37:

$$L'; E_0 \xhookrightarrow{H',\Delta} \mathcal{E}_0 \tag{C.12.43}$$

From Lemma C.8 on C.12.42 and C.12.43:

$$L',\ E_0\langle e_0\rangle \xhookrightarrow{H',\Delta} \mathcal{E}_0\langle e_0\rangle \tag{C.12.44}$$

By inverting Rule S-EXP-RTCONF on C.12.40:

$$S'; H' \hookrightarrow \mathcal{H}' \tag{C.12.45}$$

By Rule S-EXP-RTCONF using C.12.3, C.12.44 and C.12.45:

$$K; L'; \cdot; H',\ E_0\langle e'_0\rangle \overset{\Delta}{\hookrightarrow} \mathcal{K},\ \mathcal{H}',\ \mathcal{E}_0\langle e'_0\rangle \tag{C.12.46}$$

Which proves (b).

- R-CALL:

$$\mathcal{S};\, \ell.\mathrm{m}(\overline{v}) \longrightarrow \mathcal{S};\, [\overline{v}/\overline{x},\, \ell/\texttt{this}]\,(e_0) \tag{C.12.47}$$

Where by inverting R-CALL on C.12.47:

$$\mathsf{resolveMethod}(\mathcal{H},\, \ell) = (\mathbf{def}\ \mathrm{m}(\overline{x}) = e_0) \tag{C.12.48}$$

Fact C.12.5 is of the form:

$$\mathrm{L};\, \cdot;\, \mathrm{M} \xhookrightarrow{\mathrm{H},\Delta} \ell.\mathrm{m}(\overline{v}) \tag{C.12.49}$$

By Lemma C.4(b) on C.12.49:

$$\mathrm{M} \equiv \ell\,.\,\mathrm{m}\,(\overline{v}) \tag{C.12.50}$$

So C.12.49 becomes:

$$\mathrm{L};\, \cdot;\, \ell\,.\,\mathrm{m}\,(\overline{v}) \xhookrightarrow{\mathrm{H},\Delta} \ell.\mathrm{m}(\overline{v}) \tag{C.12.51}$$

By inverting Rule S-STACK-E on C.12.51:

$$\mathrm{L},\, \ell\,.\,\mathrm{m}\,(\overline{v}) \xhookrightarrow{\mathrm{H},\Delta} \ell.\mathrm{m}(\overline{v}) \tag{C.12.52}$$

By inverting Rule SR-CALL on C.12.52:

$$\mathrm{L},\, \ell \xhookrightarrow{\mathrm{H},\Delta} \ell \tag{C.12.53}$$

$$\mathrm{L},\, \overline{v} \xhookrightarrow{\mathrm{H},\Delta} \overline{v} \tag{C.12.54}$$

By inverting SR-VAL on C.12.53 and C.12.54:

$$\mathrm{H};\ell \hookrightarrow \ell \tag{C.12.55}$$

$$\mathrm{H};\overline{v} \hookrightarrow \overline{v} \tag{C.12.56}$$

By Lemma C.9 on C.12.4, C.12.55 and C.12.48:

$$\mathsf{resolveMethod}(\mathrm{H},\, \ell) = \mathrm{M} \tag{C.12.57}$$

$$\mathrm{M} \xhookrightarrow{\Delta} \mathbf{def}\ \mathrm{m}(\overline{x}) = e_0 \tag{C.12.58}$$

By Lemma C.4(d) on C.12.58:

$$M \equiv m(\overline{x}) : B \tag{C.12.59}$$

By Rule R-CALL using C.12.57, C.12.62, C.12.63 and $E \equiv \langle\rangle$:

$$K; L; X; H; \ell.m(\overline{v}) \longrightarrow K; L'; X'; H; B \tag{C.12.60}$$

Which proves (a). By inverting rule SR-METH on C.12.58:

$$\cdot, B \overset{\cdot,\Delta}{\hookleftarrow} e \tag{C.12.61}$$

Let a store L' and a stack X' *s.t.*:

$$L' \equiv \overline{x} \mapsto \overline{v}, \texttt{this} \mapsto \ell \tag{C.12.62}$$

$$X' \equiv L.\langle\rangle \tag{C.12.63}$$

By Lemma C.3 on C.12.61

$$L', B \overset{H,\Delta}{\hookleftarrow} \theta(e_0) \tag{C.12.64}$$

Where:

$$\begin{aligned} \theta &\doteq \mathsf{toSubst}(\Delta(B), L', H) \\ &= \left\{[v/x] \mid x \mapsto x \in \Delta(B),\, x \mapsto v \in L',\, H; v \hookrightarrow v\right\} \\ &= [\overline{v}/\overline{x},\, \ell/\texttt{this}] \end{aligned} \tag{C.12.65}$$

We pick:

$$M' \equiv B \tag{C.12.66}$$

By Rule S-STACK-E using C.12.64:

$$L'; \cdot; B \overset{H,\Delta}{\hookleftarrow} \theta(e_0) \tag{C.12.67}$$

It holds that:

$$L; \cdot; \langle\rangle \overset{H,\Delta}{\hookleftarrow} \langle\rangle \tag{C.12.68}$$

By Rule S-STACK-C on C.12.67 and C.12.68:

$$L'; (L.\langle\rangle);\ B \xhookrightarrow{H,\Delta} \theta\,(e_0) \tag{C.12.69}$$

By Rule S-EXP-RTCONF using C.12.3, C.12.4 and C.12.69:

$$K; L'; X'; H,\ B \xhookrightarrow{\Delta} \mathcal{K},\ \mathcal{H},\ \theta\,(e_0) \tag{C.12.70}$$

Which proves (b). From C.12.63, C.12.60, C.12.66 and C.12.55 we prove (c).

- R-LIF:

$$\mathcal{S};\ \textbf{if}\,[\,\overline{x},\ \overline{x}_1\ \overline{x}_2\,]\,n\ \textbf{then}\ u_1\langle e_0\rangle\ \textbf{else}\ u_2\langle e_0\rangle \longrightarrow \mathcal{S};\ u_i\langle[\overline{x_i}/\overline{x}]\ e_0\rangle \tag{C.12.71}$$

$$n = \textbf{true} \implies i = 1 \tag{C.12.72}$$

$$n = \textbf{false} \implies i = 2 \tag{C.12.73}$$

Let:

$$n = \textbf{true} \tag{C.12.74}$$

The case for **false** is symmetrical. Facts C.12.71 and C.12.6 become:

$$\mathcal{S};\ \textbf{if}\,[\,\overline{x},\ \overline{x}_1,\ \overline{x}_2\,]\ \textbf{true then}\ u_1\ \textbf{else}\ u_2 \longrightarrow \mathcal{S};\ u_1\langle[\overline{x}_1/\overline{x}]\,(e_0)\rangle \tag{C.12.75}$$

$$L,\ M \xhookrightarrow{H,\Delta} \textbf{if}\,[\,\overline{x},\ \overline{x}_1,\ \overline{x}_2\,]\ \textbf{true then}\ u_1\langle e_0\rangle\ \textbf{else}\ u_2\langle e_0\rangle \tag{C.12.76}$$

By Lemma C.4(c) on C.12.76:

$$M \equiv \texttt{if}\ (e_c)\ \{s_1\}\ \texttt{else}\ \{s_2\};\ \texttt{return}\ e_0 \tag{C.12.77}$$

So C.12.76 becomes:

$$L,\ \texttt{if}\ (e_c)\ \{s_1\}\ \texttt{else}\ \{s_2\};\ \texttt{return}\ e_0 \xhookrightarrow{H,\Delta} \textbf{if}\,[\,\overline{x},\ \overline{x}_1,\ \overline{x}_2\,]\ \textbf{true then}\ u_1\langle e_0\rangle\ \textbf{else}\ u_2\langle e_0\rangle \tag{C.12.78}$$

By inverting Rule SR-BODY on C.12.78:

$$L,\ \texttt{if}\ (e_c)\ \{s_1\}\ \texttt{else}\ \{s_2\} \xhookrightarrow{H,\Delta} \textbf{if}\,[\,\overline{x},\ \overline{x}_1,\ \overline{x}_2\,]\ \textbf{true then}\ u_1\ \textbf{else}\ u_2 \tag{C.12.79}$$

$$\Delta' = \Delta[\,e_0 \mapsto \Delta\lfloor\texttt{if}\ (e_c)\ \{s_1\}\ \texttt{else}\ \{s_2\}\rfloor\,] \tag{C.12.80}$$

$$L,\ e_0 \xhookrightarrow{H,\Delta'} e_0 \tag{C.12.81}$$

By inverting Rule SR-ITE on C.12.79:

$$\mathrm{L},\ e_c \xhookrightarrow{\mathrm{H},\Delta} \textbf{true} \tag{C.12.82}$$

$$\mathrm{L},\ s_1 \xhookrightarrow{\mathrm{H},\Delta} u_1 \tag{C.12.83}$$

$$\mathrm{L},\ s_2 \xhookrightarrow{\mathrm{H},\Delta} u_2 \tag{C.12.84}$$

$$(\overline{x}; \overline{x}_1; \overline{x}_2) = \Delta\lfloor s_1 \rfloor \bowtie \Delta\lfloor s_2 \rfloor \tag{C.12.85}$$

$$\overline{x} = \Delta\lfloor \texttt{if (}e_c\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rfloor(\overline{x}) \tag{C.12.86}$$

By Lemma C.4 on C.12.82 we get:

$$e_c \equiv \texttt{true} \tag{C.12.87}$$

By Rules R-EVALCTX and R-ITE we get:

$$S;\ \texttt{if (true) \{}s_1\texttt{\} else \{}s_2\texttt{\}; return }e_0 \longrightarrow S;\ s_1\texttt{; return }e_0 \tag{C.12.88}$$

Which proves (a). Let:

$$\Delta'' \equiv \Delta'[\, e_0 \mapsto \Delta\lfloor s_1 \rfloor\,] \tag{C.12.89}$$

By Lemma C.1 on C.12.81 using C.12.89:

$$\mathrm{L},\ e_0 \xhookrightarrow{\mathrm{H},\Delta''} [\Delta''(e_0)/\Delta'(e_0)]\,(e_0) \tag{C.12.90}$$

From C.12.80 and C.12.89 it holds that:

$$\Delta'(e_0) = \Delta\lfloor \texttt{if (true) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rfloor \tag{C.12.91}$$

$$\Delta''(e_0) = \Delta\lfloor s_1 \rfloor \tag{C.12.92}$$

So:

$$\Delta'(e_0) \bowtie \Delta''(e_0) = (\overline{x}; \overline{x}_1; \overline{x}) \tag{C.12.93}$$

By Definition C.2.1:

$$[\Delta''(e_0)/\Delta'(e_0)] = [\overline{x}_1/\overline{x}] \tag{C.12.94}$$

So C.12.90 becomes:

$$\mathsf{L},\ e_0 \xhookrightarrow{\mathsf{H},\Delta''} [\overline{x}_1/\overline{x}]\,(e_0) \tag{C.12.95}$$

By Rule SR-BODY on C.12.83, C.12.92 and C.12.95, using C.12.94:

$$\mathsf{L},\ s_1\,;\,\texttt{return}\ e_0 \xhookrightarrow{\mathsf{H},\Delta} u_1\langle[\overline{x}_1/\overline{x}]\,(e_0)\rangle \tag{C.12.96}$$

Which, using S-EXP-RTCONF and S-STACK-E, prove (b) and (c).

- R-CAST, R-NEW, R-LETIN, R-DOTASGN, R-FIELD: *Cases handled in similar fashion as before.*

□

Corollary C.2.1 (Empty Stack Valid Configuration)**.** *If*

(a) $\mathsf{S},\ \mathsf{M} \xhookrightarrow{\Delta} \mathcal{S},\ e$

(b) $\mathsf{S.X} = \cdot$

(c) $\mathcal{S};\ w \longrightarrow \mathcal{S}';\ w'$

then $\mathsf{S};\mathsf{M} \longrightarrow^{*} \mathsf{S}';\mathsf{M}'$ *with* $\mathsf{validConf}(\mathsf{S}',\ \mathsf{M}')$.

Proof. Examine all cases of result (c) of Lemma C.12. □

Lemma C.13 (Consistency)**.** *If*

(i) $\mathsf{S},\ \mathsf{M} \xhookrightarrow{\Delta} \mathcal{S},\ e$

(ii) $\mathcal{S};\ w \longrightarrow \mathcal{S}';\ w'$

(iii) $\mathsf{validConf}(S,\ \mathsf{M})$

then there exist S' *and* M' *s.t.:*

(a) $\mathsf{S};\mathsf{M} \longrightarrow^{*} \mathsf{S}';\mathsf{M}'$,

(b) $\mathsf{S}',\ \mathsf{M}' \xhookrightarrow{\Delta} \mathcal{S}',\ e'$

(c) $\mathsf{validConf}(\mathsf{S}',\ \mathsf{M}')$

Proof. Let:

$$\mathsf{S} \equiv \mathsf{K};\mathsf{L};\mathsf{X};\mathsf{H} \tag{C.13.1}$$

By inverting Rule S-EXP-RTCONF on (i):

$$\mathsf{K} \overset{\Delta}{\hookrightarrow} \mathcal{K} \tag{C.13.2}$$

$$S; \mathsf{H} \hookrightarrow \mathcal{H} \tag{C.13.3}$$

$$\mathsf{L};\ \mathsf{X};\ \mathsf{M} \overset{\mathsf{H},\Delta}{\hookrightarrow} e \tag{C.13.4}$$

We proceed by induction on the derivation C.13.4:

- S-STACK-E:

$$\mathsf{L};\ \cdot;\ \mathsf{M} \overset{\mathsf{H},\Delta}{\hookrightarrow} e \tag{C.13.5}$$

By Lemma C.12 using (i) and (ii) there exist M' and S' *s.t.*:

$$S; \mathsf{M} \longrightarrow^{*} S'; \mathsf{M}' \tag{C.13.6}$$

$$S',\ \mathsf{M}' \overset{\Delta}{\hookrightarrow} \mathcal{S}',\ e' \tag{C.13.7}$$

From Corollary C.2.1 using (i), (ii) and (iii) we get:

$$\mathsf{validConf}(S',\ \mathsf{M}') \tag{C.13.8}$$

We prove (a), (b) and (c) by C.13.6, C.13.7 and C.13.8, respectively.

- S-STACK-C:

$$\mathsf{L};\ (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0);\ \mathsf{M} \overset{\mathsf{H},\Delta}{\hookrightarrow} \mathcal{E}_0\langle e_0 \rangle \tag{C.13.9}$$

Where:

$$\mathsf{X} \equiv \mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0 \tag{C.13.10}$$

By (iii) and the definition of a *valid configuration*, there exists a B_0 *s.t.*:

$$\mathsf{M} \equiv \mathsf{B}_0 \tag{C.13.11}$$

By inverting Rule S-STACK-C on C.13.9 using C.13.11:

$$\mathsf{L};\ \cdot;\ \mathsf{B}_0 \overset{\mathsf{H},\Delta}{\hookrightarrow} e_0 \tag{C.13.12}$$

$$\mathsf{L}_0;\ \mathsf{X}_0;\ \mathsf{E}_0 \overset{\mathsf{H},\Delta}{\hookrightarrow} \mathcal{E}_0 \tag{C.13.13}$$

By applying Rule S-EXP-RTCONF on C.13.2, C.13.3 and C.13.12:

$$\mathsf{K}; \mathsf{L}; \cdot; \mathsf{H},\ \mathsf{B}_0 \overset{\Delta}{\hookrightarrow} \mathcal{K},\ \mathcal{H},\ e_0 \tag{C.13.14}$$

We examine cases on the configuration of $\mathcal{S},\ e_0$:

- Case $\mathcal{S},\ e_0$ is a *terminal* configuration, so there exists v *s.t.*:

$$e_0 \equiv v \tag{C.13.15}$$

  Fact C.13.14 becomes:

$$\mathsf{K}; \mathsf{L}; \cdot; \mathsf{H},\ \mathsf{B}_0 \overset{\Delta}{\hookrightarrow} \mathcal{K},\ \mathcal{H},\ v \tag{C.13.16}$$

  By Lemma C.10 on C.13.16:

$$\mathsf{K}; \mathsf{L}; \cdot; \mathsf{H}; \mathsf{B}_0 \longrightarrow^* \mathsf{K}; \mathsf{L}; \cdot; \mathsf{H}; \texttt{return}\ v \tag{C.13.17}$$

$$\mathsf{K}; \mathsf{L}; \cdot; \mathsf{H},\ \texttt{return}\ v \overset{\Delta}{\hookrightarrow} \mathcal{S},\ v \tag{C.13.18}$$

  By Lemma C.11 on C.13.17:

$$\mathsf{K}; \mathsf{L}; \mathsf{X}; \mathsf{H}; \mathsf{B}_0 \longrightarrow^* \mathsf{K}; \mathsf{L}; \mathsf{X}; \mathsf{H}; \texttt{return}\ v \tag{C.13.19}$$

  By inverting Rule S-EXP-RTCONF on C.13.18:

$$\mathsf{L};\ \cdot;\ \texttt{return}\ v \overset{\mathsf{H},\Delta}{\longhookrightarrow} v \tag{C.13.20}$$

  By applying Rule S-STACK-C on C.13.20 and C.13.13:

$$\mathsf{L};\ (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0);\ \texttt{return}\ v \overset{\mathsf{H},\Delta}{\longhookrightarrow} \mathcal{E}_0\langle v \rangle \tag{C.13.21}$$

  By applying Rule S-EXP-RTCONF on C.13.2, C.13.3 and C.13.21:

$$\mathsf{K}; \mathsf{L}; (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0); \mathsf{H},\ \texttt{return}\ v \overset{\Delta}{\hookrightarrow} \mathcal{K},\ \mathcal{H},\ \mathcal{E}_0\langle v \rangle \tag{C.13.22}$$

  By applying Rule R-RET on on THe left-hand side of C.13.22:

$$\mathsf{K}; \mathsf{L}; (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0); \mathsf{H};\ \texttt{return}\ v \longrightarrow \mathsf{K}; \mathsf{L}_0; \mathsf{X}_0; \mathsf{H};\ \mathsf{E}_0\langle v \rangle \tag{C.13.23}$$

By inverting S-STACK-E and SR-BODY on C.13.20:

$$\mathsf{L},\, v \xhookrightarrow{\mathsf{H},\Delta} v \tag{C.13.24}$$

By inverting Rule SR-VAL on C.13.24:

$$\mathsf{H}; v \hookrightarrow v \tag{C.13.25}$$

By applying Rule SR-VAL on C.13.25 using L_0:

$$\mathsf{L}_0,\, v \xhookrightarrow{\mathsf{H},\Delta} v \tag{C.13.26}$$

By applying Lemma C.8 on C.13.13 and C.13.26:

$$\mathsf{L}_0;\, \mathsf{X}_0;\, \mathsf{E}_0\langle v\rangle \xhookrightarrow{\mathsf{H},\Delta} \mathcal{E}_0\langle v\rangle \tag{C.13.27}$$

By applying Rule S-EXP-RTCONF on C.13.2, C.13.3 and C.13.27:

$$\mathsf{K}; \mathsf{L}_0; \mathsf{X}_0; \mathsf{H},\, \mathsf{E}_0\langle v\rangle \xhookrightarrow{\Delta} \mathcal{K},\, \mathcal{H},\, \mathcal{E}_0\langle v\rangle \tag{C.13.28}$$

Because of C.13.11:

$$\mathsf{validConf}(\mathsf{K}; \mathsf{L}_0; \mathsf{X}_0; \mathsf{H},\, \mathsf{E}_0\langle v\rangle) \tag{C.13.29}$$

By induction hypothesis using C.13.28, (ii) and C.13.29:

$$\mathsf{K}; \mathsf{L}_0; \mathsf{X}_0; \mathsf{H}; \mathsf{E}_0\langle v\rangle \longrightarrow^* \mathsf{S}'; \mathsf{M}' \tag{C.13.30}$$

$$\mathsf{S}',\, \mathsf{M}' \xhookrightarrow{\Delta} \mathcal{S}',\, e' \tag{C.13.31}$$

$$\mathsf{validConf}(\mathsf{S}',\, \mathsf{M}') \tag{C.13.32}$$

We prove (a) by C.13.19, C.13.23 and C.13.33; (b) by C.13.31; and (c) by C.13.32.

♦ Case $\mathcal{S},\, e_0$ is a *non-terminal* configuration, so there exists e_0' *s.t.*:

$$\mathcal{S};\, e_0 \longrightarrow \mathcal{S}';\, e_0' \tag{C.13.33}$$

By Rule RC-ECTX using C.13.33:

$$\mathcal{S};\, \mathcal{E}_0\langle e_0\rangle \longrightarrow \mathcal{S}';\, \mathcal{E}_0\langle e_0'\rangle \tag{C.13.34}$$

By Lemma C.12 using C.13.14 and C.13.33:

$$\mathsf{K}; \mathsf{L}; \cdot; \mathsf{H}; \mathsf{B}_0 \longrightarrow^* \mathsf{S}'; \mathsf{M}' \tag{C.13.35}$$

$$\mathsf{S}', \mathsf{M}' \overset{\Delta}{\hookrightarrow} \mathcal{S}', e'_0 \tag{C.13.36}$$

And we examine cases on the form of B_0 for the last result of the above lemma:

- Case $\mathsf{B}_0 \equiv \mathsf{E}\langle \ell.\, \mathsf{m}(\overline{\mathsf{v}}) \rangle$. It holds that:

$$\mathsf{S}', \mathsf{M}' \equiv \mathsf{K}; \mathsf{L}_1; (\mathsf{L}.\mathsf{E}); \mathsf{H}, \mathsf{B}_1 \tag{C.13.37}$$

  So C.13.36 becomes:

$$\mathsf{K}; \mathsf{L}_1; (\mathsf{L}.\mathsf{E}); \mathsf{H}, \mathsf{B}_1 \overset{\Delta}{\hookrightarrow} \mathcal{S}', e'_0 \tag{C.13.38}$$

  By inverting S-EXP-RTCONF on C.13.38:

$$\mathsf{L}_1;\ (\mathsf{L}.\mathsf{E});\ \mathsf{B}_1 \overset{\mathsf{H},\Delta}{\longhookrightarrow} e'_0 \tag{C.13.39}$$

  By Lemma C.7 using C.13.13 and C.13.39:

$$\mathsf{L}_1;\ (\mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0,\ \mathsf{L}.\mathsf{E});\ \mathsf{B}_1 \overset{\mathsf{H},\Delta}{\longhookrightarrow} \mathcal{E}_0\langle e'_0 \rangle \tag{C.13.40}$$

  Let:

$$\mathsf{X}' \equiv \mathsf{X}_0,\ \mathsf{L}_0.\mathsf{E}_0,\ \mathsf{L}.\mathsf{E} \tag{C.13.41}$$

  By applying Rule S-EXP-RTCONF on C.13.2, C.13.3 and C.13.40:

$$\mathsf{K}; \mathsf{L}_1; \mathsf{X}'; \mathsf{H},\ \mathsf{B}_1 \overset{\Delta}{\hookrightarrow} \mathcal{S}',\ \mathcal{E}_0\langle e'_0 \rangle \tag{C.13.42}$$

  By Lemma C.11 on C.13.35:

$$\mathsf{K}; \mathsf{L}; \mathsf{X}; \mathsf{H}; \mathsf{B}_0 \longrightarrow^* \mathsf{K}; \mathsf{L}_1; \mathsf{X}'; \mathsf{H}; \mathsf{B}_1 \tag{C.13.43}$$

  We prove (a), (b) and (c) by C.13.43, C.13.42 and C.13.37, respectively.

- For all remaining cases on B_0:

$$\mathsf{H}' \supseteq \mathsf{H} \tag{C.13.44}$$

$$\mathsf{L}' \supseteq \mathsf{L} \tag{C.13.45}$$

Because of C.13.11, it holds that:

$$S', M' \equiv K; L'; \cdot; H', B' \tag{C.13.46}$$

By inverting Rule S-EXP-RTCONF on C.13.36:

$$S'; H' \hookrightarrow \mathcal{H}' \tag{C.13.47}$$

By Lemma C.11 on C.13.35:

$$K; L; X; H; B_0 \longrightarrow^* K; L'; X; H'; B' \tag{C.13.48}$$

Fact C.13.36 becomes:

$$K; L'; \cdot; H', B' \overset{\Delta}{\hookrightarrow} \mathcal{S}', e'_0 \tag{C.13.49}$$

By inverting S-EXP-RTCONF on C.13.49:

$$L'; \cdot; B' \xhookrightarrow{H',\Delta} e'_0 \tag{C.13.50}$$

By applying Lemma C.6 on C.13.13 using C.13.44:

$$L_0; X_0; E_0 \xhookrightarrow{H',\Delta} \mathcal{E}_0 \tag{C.13.51}$$

By applying rule S-STACK-C on C.13.13 and C.13.50:

$$L'; (X_0, L_0.E_0); B' \xhookrightarrow{H',\Delta} \mathcal{E}_0\langle e'_0\rangle \tag{C.13.52}$$

By applying rule S-EXP-RTCONF on C.13.2, C.13.47 and C.13.52:

$$K; L'; X; H', B' \overset{\Delta}{\hookrightarrow} \mathcal{S}', \mathcal{E}_0\langle e'_0\rangle \tag{C.13.53}$$

We prove (a), (b) and (c) by C.13.48, C.13.53 and C.13.46, respectively.

□

Theorem C.14 (Forward Simulation)**.** *If* $R \overset{\Delta}{\hookrightarrow} \mathcal{R}$*, then:*

(a) if $\mathcal{R}$ *is terminal, then there exists* R' *s.t.* $R \longrightarrow^* R'$ *and* $R' \overset{\Delta}{\hookrightarrow} \mathcal{R}$

(b) if $\mathcal{R} \longrightarrow \mathcal{R}'$*, then there exists* R' *s.t.* $R \longrightarrow^* R'$ *and* $R' \overset{\Delta}{\hookrightarrow} \mathcal{R}'$

Proof. Part (a) is proven by use of by Lemma C.10, and part (b) by Lemma C.13.

□

C.2.2 Type Safety

Lemma C.15 (Substitution Lemma)**.** *If*

(i) $\Gamma \vdash \overline{w}_0 : \overline{\mathsf{T}_0}$

(ii) $\Gamma, \overline{x}\colon \overline{\mathsf{T}_0} \vdash \overline{\mathsf{T}_0} \leq \overline{\mathsf{T}_0}'$

(iii) $\Gamma, \overline{x}\colon \overline{\mathsf{T}_0}' \vdash w : \mathsf{T}$

then $\Gamma \vdash [\overline{w}_0/\overline{x}]\,(w) : \mathsf{T}_1,\ \mathsf{T}_1 \leq \mathsf{T}$

Proof. By induction on the derivation of the statement $\Gamma, \overline{x}\colon \overline{\mathsf{T}_0} \vdash w : \mathsf{T}$. □

Lemma C.16 (Environment Substitution)**.** *If*

$$\Gamma_1,\ x\colon \mathsf{T},\ \Gamma_2 \vdash w : \mathsf{T}'$$

then

$$\Gamma_1,\ x\colon \mathsf{T},\ [y/x]\,(\Gamma_2) \vdash [y/x]\,(w) : [y/x]\,(\mathsf{T}')$$

Lemma C.17 (Weakening Subtyping)**.** *If* $\Gamma \vdash \mathsf{T}' \leq \mathsf{T}$*, then* $\Gamma, x\colon \mathsf{T}_1 \vdash \mathsf{T}' \leq \mathsf{T}$.

Lemma C.18 (Weakening Typing)**.** *If* $\Gamma \vdash w : \mathsf{T}$*, then for* $\Gamma' \supseteq \Gamma$*, it holds that* $\Gamma' \vdash w : \mathsf{T}$.

Lemma C.19 (Store Type)**.** *If*

(i) $\Sigma \vdash \mathcal{H}$

(ii) $\mathcal{H}(\ell) = \mathcal{O}$

(iii) $\Sigma(\ell) = \mathsf{T}$

then $\Sigma \vdash_{\mathcal{H}} \mathcal{O} : \mathsf{T}', \mathsf{T} \leq \mathsf{T}'$.

Lemma C.20 (Method Body Type – Lemma A.3 [76])**.** *If*

(i) $\Gamma, y\colon \mathsf{T} \vdash y\ \mathsf{has}\ (\mathbf{def}\ m(\overline{y}\colon \overline{\mathsf{T}}_2) :\ \mathsf{T}_1 = w)$

(ii) $\Gamma, y\colon \mathsf{T}, \overline{y}\colon \overline{\mathsf{T}}_2' \vdash \overline{\mathsf{T}}_2' \leq \overline{\mathsf{T}}_2$

then for some type T_1' *it holds that*

$$\Gamma, y\colon \mathsf{T}, \overline{y}\colon \overline{\mathsf{T}}_2' \vdash w\colon \mathsf{T}_1',\ \mathsf{T}_1' \leq \mathsf{T}_1$$

Lemma C.21 (Cast)**.** *If*

(i) $\Sigma \vdash \mathcal{H}$

(ii) $\Gamma; \Sigma \vdash \ell\colon \mathsf{T}', \mathsf{T}' \lesssim \mathsf{T}$

then $\Gamma; \Sigma \vdash \mathcal{H}(\ell)\colon \mathsf{T}_1, \mathsf{T}_1 \leq \mathsf{T}$

Lemma C.22 (Evaluation Context Typing)**.** *If* $\Gamma \vdash \mathcal{E}\langle w \rangle : \mathsf{T}$*, then for some type* T' *it holds that* $\Gamma \vdash w : \mathsf{T}'$*.*

Proof. By induction on the structure of the evaluation context $\mathcal{E}$. □

Lemma C.23 (Evaluation Context Step Typing)**.** *If*

$$\Gamma; \Sigma \vdash \mathcal{E}\langle w \rangle : \mathsf{T}, w : \mathsf{T}_0$$

and for some expression w' *and heap typing* $\Sigma' \supseteq \Sigma$ *it holds that*

$$\Gamma; \Sigma' \vdash w' \colon \mathsf{T}_0', \mathsf{T}_0' \lesssim \mathsf{T}_0$$

then $\Gamma; \Sigma' \vdash \mathcal{E}\langle w' \rangle \colon \mathsf{T}', \mathsf{T}' \lesssim \mathsf{T}$

Proof. By induction on the structure of the evaluation context $\mathcal{E}$. □

Lemma C.24 (Selfification)**.** *If* $\Gamma, x \colon \mathsf{T}' \vdash \mathsf{T}' \leq \mathsf{T}$ *then* $\Gamma, x \colon \mathsf{T}' \vdash \mathsf{T}' \leq \mathsf{sngl}\,(\mathsf{T}, x)$*.*

Lemma C.25 (Existential Weakening)**.** *If* $\Gamma \vdash \mathsf{T}_1 \leq \mathsf{T}_1'$ *then* $\Gamma \vdash \exists\, x \colon \mathsf{T}_1 \,.\, \mathsf{T} \leq \exists\, x \colon \mathsf{T}_1' \,.\, \mathsf{T}$*.*

Lemma C.26 (Boolean Facts)**.** *If*

(i) $\Gamma \vdash x : \mathsf{T}, \mathsf{T} \leq \{\, \nu \colon \mathsf{Bool} \mid \nu = \mathbf{true} \,\}$

(ii) $\Gamma, x \vdash w : \mathsf{T}', \mathsf{T}' \leq \mathsf{T}$

then $\Gamma \vdash w : \mathsf{T}', \mathsf{T}' \leq \mathsf{T}$

Theorem C.27 (Subject Reduction)**.** *If*

(i) $\Gamma;\, \Sigma \vdash w \colon \mathsf{T}$

(ii) $\mathcal{S}; w \longrightarrow \mathcal{S}'; w'$

(iii) $\Sigma \vdash \mathcal{S}.\mathcal{H}$

then for some T' *and* $\Sigma' \supseteq \Sigma$*:*

(a) $\Gamma;\, \Sigma' \vdash w' \colon \mathsf{T}'$

(b) $\Gamma \vdash \mathsf{T}' \lesssim \mathsf{T}$

(c) $\Sigma' \vdash \mathcal{H}'$*.*

Proof. We proceed by induction on the structure of fact (ii):

$$\mathcal{S}; w \longrightarrow \mathcal{S}'; w'$$

We have the following cases:

- RC-ECTX: Fact (ii) has the form:

$$\mathcal{S}; \mathcal{E}\langle w_0 \rangle \longrightarrow \mathcal{S}'; \mathcal{E}\langle w_0' \rangle \tag{C.27.1}$$

From (i):

$$\Gamma; \Sigma \vdash \mathcal{E}\langle w_0 \rangle : \mathsf{T} \tag{C.27.2}$$

By Lemma C.22 on C.27.2:

$$\Gamma; \Sigma \vdash w_0 : \mathsf{T}_0 \tag{C.27.3}$$

By inverting Rule RC-ECTX on C.27.1:

$$\mathcal{S}; w_0 \longrightarrow \mathcal{S}'; w_0' \tag{C.27.4}$$

By induction hypothesis, using C.27.3, C.27.4 and (iii) we get:

$$\Gamma; \Sigma' \vdash w_0' : \mathsf{T}_0' \tag{C.27.5}$$
$$\Gamma; \Sigma' \vdash \mathsf{T}_0' \lesssim \mathsf{T}_0 \tag{C.27.6}$$
$$\Sigma' \vdash \mathcal{S}'.\mathcal{H} \tag{C.27.7}$$
$$\Sigma' \supseteq \Sigma \tag{C.27.8}$$

For some type T_0' and heap $\mathcal{S}'.\mathcal{H}$.

From C.27.7 we prove (c).

By Lemma C.23 using C.27.2, C.27.3, C.27.5, C.27.6 and C.27.8:

$$\Gamma; \Sigma' \vdash \mathcal{E}\langle w_0' \rangle : \mathsf{T}', \mathsf{T}' \lesssim \mathsf{T} \tag{C.27.9}$$

From C.27.9 we prove (a) and (b).

- R-FIELD: Fact (ii) has the form:

$$\mathcal{S}; \ell.\mathfrak{h} \longrightarrow \mathcal{S}; v \tag{C.27.10}$$

By Fact (i) for $w \equiv \ell.\mathfrak{h}$ we have:

$$\Gamma; \Sigma \vdash \ell.\mathfrak{h} : \mathsf{T} \tag{C.27.11}$$

By inverting R-FIELD on C.27.10:

$$\mathcal{S}.\mathcal{H}(\ell) \equiv \mathcal{O} = \{\texttt{proto}: \ell'; \texttt{f}: \vec{\mathcal{F}}; \ldots\} \tag{C.27.12}$$

$$\texttt{f} := \nu \in \vec{\mathcal{F}} \tag{C.27.13}$$

By inverting WF-HEAP-INST on (iii) for location ℓ:

$$\vec{\mathcal{F}} \doteq \circ\,\overline{f} := \overline{\nu}_\circ, \square\,\overline{g} := \overline{\nu}_\square \tag{C.27.14}$$

$$\lfloor \Sigma(\ell) \rfloor = C \tag{C.27.15}$$

$$\Gamma, y: C \vdash \mathsf{fields}(y) = \circ\,\overline{f}: \overline{T_2}, \square\,\overline{g}: \overline{T_3} \tag{C.27.16}$$

$$\Sigma \vdash \overline{\nu}_\circ : \overline{T}_\circ \tag{C.27.17}$$

$$\Sigma \vdash \overline{\nu}_\square : \overline{T}_\square \tag{C.27.18}$$

$$\Gamma, y: C, \overline{y}_\circ: \mathsf{sngl}\left(\overline{T}_\circ, y.\overline{f}\right) \vdash \overline{T}_\circ \leq \overline{T_2}, \overline{T}_\square \leq \overline{T_3}, \mathsf{inv}(C, y) \tag{C.27.19}$$

By applying RT-T-OBJ on C.27.15, C.27.14 and C.27.17:

$$\Gamma; \Sigma \vdash \mathcal{O}: T_1' \tag{C.27.20}$$

Where:

$$T_1' \equiv \exists\, \overline{y}_\circ: \overline{T}_\circ . \{\, \nu: C \mid \nu.\overline{f} = \overline{y}_\circ \wedge \mathsf{inv}(C, \nu)\,\} \tag{C.27.21}$$

By Lemma C.19 using (iii), C.27.12 and C.27.15:

$$\Gamma \vdash T_1 \leq T_1' \tag{C.27.22}$$

Where:

$$\Sigma(\ell) = T_1 \tag{C.27.23}$$

We examine cases on the typing statement C.27.11:

- T-FLD-I: Field h is an immutable field f_i, so fact C.27.11 becomes:

$$\Gamma; \Sigma \vdash \ell.f_i: \exists\, y: T_1 \,.\, \mathsf{sngl}\,(T_{2.i}, y.f_i) \tag{C.27.24}$$

  By inverting T-FLD-I on C.27.24:

$$\Sigma \vdash \ell : T_1 \tag{C.27.25}$$

$$\Gamma, y: T_1; \Sigma \vdash y\ \mathsf{hasImm}\ f_i: T_{2.i} \tag{C.27.26}$$

For a fresh y.

Keeping only the relevant part of C.27.17 and C.27.19:

$$\Gamma;\, \Sigma \vdash \nu_i \colon T_i \tag{C.27.27}$$

$$\Gamma,\, y \colon C, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right); \Sigma \vdash T_i \leq T_{2.i} \tag{C.27.28}$$

By C.27.27 we prove (a).

By Lemma C.24 using C.27.28 and picking y_i as the selfification variable:

$$\Gamma,\, y \colon C, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right); \Sigma \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y_i\right) \tag{C.27.29}$$

For the above environment it holds that:

$$[\![\Gamma,\, y \colon C, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right); \Sigma]\!] \implies y_i = y.f_i \tag{C.27.30}$$

By SUB-REFL and By Lemma C.24 using C.27.30:

$$\Gamma,\, y \colon C, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right); \Sigma \vdash \mathsf{sngl}\left(T_{2.i},\, y_i\right) \leq \mathsf{sngl}\left(\mathsf{sngl}\left(T_{2.i},\, y_i\right),\, y.f_i\right) \tag{C.27.31}$$

By simplifying C.27.31 using SUB-TRANS on C.27.29 and C.27.31 we get:

$$\Gamma,\, y \colon C, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right); \Sigma \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y.f_i\right) \tag{C.27.32}$$

By C.27.32 it also holds that:

$$\Gamma, y \colon \exists\, \overline{y}_\circ \colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right)\, .\, C \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y.f_i\right) \tag{C.27.33}$$

By C.27.33 it also holds that:

$$\Gamma, y \colon \exists\, \overline{y}_\circ \colon \overline{T}_\circ\, .\, \mathsf{sngl}\left(C,\, \overline{y}_\circ\right) \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y.f_i\right) \tag{C.27.34}$$

By expanding C.27.34 and C.27.19:

$$\Gamma, y \colon \exists\, \overline{y}_\circ \colon \overline{T}_\circ\, .\, \{\, \nu \colon C \mid \nu.\overline{f} = \overline{y}_\circ \wedge \mathsf{inv}(C,\, \nu)\,\} \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y.f_i\right) \tag{C.27.35}$$

By using C.27.21 on C.27.35:

$$\Gamma, y \colon T_1' \vdash T_i \leq \mathsf{sngl}\left(T_{2.i},\, y.f_i\right) \tag{C.27.36}$$

By Lemma C.17 using C.27.36 and C.27.22:

$$\Gamma, y\colon T_1 \vdash T_i \leq \mathsf{sngl}\,(T_{2.i},\, y.f_i) \tag{C.27.37}$$

From Rule SUB-WITNESS using C.27.37:

$$\Gamma \vdash T_i \leq \exists\, y\colon T_1\,.\,\mathsf{sngl}\,(T_{2.i},\, y.f_i) \tag{C.27.38}$$

Using C.27.24, C.27.17 and C.27.38 we prove (b).

Heap $\mathcal{S}.\mathcal{H}$ does not evolve so (c) holds trivially.

♦ T-FLD-M: Field h is a mutable field g_i, so fact (i) becomes:

$$\Gamma;\, \Sigma \vdash \ell.g_i\colon \exists\, y\colon T_1\,.\,T_{5.i} \tag{C.27.39}$$

By inverting T-FLD-M on C.27.39:

$$\Gamma \vdash \ell : T_1 \tag{C.27.40}$$

$$\Gamma, \ell\colon T_1 \vdash y\ \mathsf{hasMut}\ g_i : T_{3i} \tag{C.27.41}$$

For a fresh y.

Keeping only the relevant parts of C.27.17 and C.27.19:

$$\Gamma \vdash v_i : T_i \tag{C.27.42}$$

$$\Gamma,\, y\colon C, \overline{y}_\circ\colon \mathsf{sngl}\left(\overline{T}_\circ,\, y.\overline{f}\right) \vdash T_i \leq T_{3i} \tag{C.27.43}$$

By C.27.42 we prove (a).

By similar reasoning as before and using C.27.43 we get:

$$\Gamma, y\colon T_1' \vdash T_i \leq T_{3i} \tag{C.27.44}$$

By Lemma C.17 using C.27.44 and C.27.22:

$$\Gamma, y\colon T_1 \vdash T_i \leq T_{3i} \tag{C.27.45}$$

By Rule SUB-WITNESS using C.27.45:

$$\Gamma \vdash T_i \leq \exists\, y\colon T_1\,.\,T_{3i} \tag{C.27.46}$$

Using C.27.39, C.27.17 and C.27.46 we prove (b).

Heap $\mathcal{S}.\mathcal{H}$ does not evolve so (c) holds trivially.

- R-CALL: Fact (ii) has the form:

$$\mathcal{S};\, \ell.\mathrm{m}(\overline{v}) \longrightarrow \mathcal{S};\, [\overline{v}/\overline{y},\, \ell/\mathtt{this}]\,(w') \tag{C.27.47}$$

By (i) for $w \equiv \ell.\mathrm{m}(\overline{v})$ we have:

$$\Gamma;\, \Sigma \vdash \ell.\mathrm{m}(\overline{v})\colon \exists\, y\colon T\,.\,\exists\, \overline{y}\colon \overline{T}\,.\,T_1 \tag{C.27.48}$$

By inverting T-MTH-CALL on C.27.48:

$$\Gamma; \Sigma \vdash \ell : T, \overline{v} : \overline{T} \tag{C.27.49}$$

$$\Gamma, y\colon T, \overline{y}\colon \overline{T} \vdash y \text{ has } (\mathbf{def}\ \mathrm{m}(\overline{y} :\ \overline{T_2})\,\{P\}\colon\ T_1 = w') \tag{C.27.50}$$

$$\Gamma, y\colon T, \overline{y}\colon \overline{T} \vdash \overline{T} \leq \overline{T_2} \tag{C.27.51}$$

$$\Gamma, y\colon T, \overline{y}\colon \overline{T} \vdash P \tag{C.27.52}$$

With fresh y and $\overline{y}$.

By inverting R-CALL on C.27.47:

$$\mathsf{resolveMethod}(\mathcal{H},\, \ell) = (\mathbf{def}\ \mathrm{m}(\overline{y} :\ \overline{T_2})\,\{P\}\colon\ T_1 = w) \tag{C.27.53}$$

$$\mathsf{eval}(P) = \mathbf{true} \tag{C.27.54}$$

Note that **this** has already been substituted by ℓ in T_1 and P.
By Lemma C.20 using C.27.50 and C.27.51:

$$\Gamma, y\colon T, \overline{y}\colon \overline{T} \vdash w'\colon T_1',\, T_1' \leq T_1 \tag{C.27.55}$$

By C.27.55 we prove (a).

By Rule SUB-WITNESS using C.27.55:

$$\Gamma \vdash T_1' \leq \exists\, y\colon T\,.\,\exists\, \overline{y}\colon \overline{T}\,.\,T_1 \tag{C.27.56}$$

By Lemma C.15 using C.27.49, C.27.51 and C.27.55:

$$\Gamma \vdash [\overline{v}/\overline{y},\, \ell/\mathtt{this}]\,(w') : T_3,\, T_3 \leq T_1' \tag{C.27.57}$$

By Rule SUB-TRANS on C.27.55 and C.27.57:

$$\Gamma \vdash \mathsf{T}_3 \leq \exists\, y\colon \mathsf{T}\,.\, \exists\, \overline{y}\colon \overline{\mathsf{T}}\,.\, \mathsf{T}_1 \tag{C.27.58}$$

By C.27.58 we prove (b).

Heap $\mathcal{S}.\mathcal{H}$ does not evolve so (c) holds trivially.

- R-CAST: Fact (ii) has the form:

$$\mathcal{S};\, \ell \;\mathbf{as}\; \mathsf{T} \longrightarrow \mathcal{S};\, \ell$$

By (i) for $w \equiv \ell \;\mathbf{as}\; \mathsf{T}$ we have:

$$\Gamma;\, \Sigma \vdash \ell \;\mathbf{as}\; \mathsf{T}\colon \mathsf{T} \tag{C.27.59}$$

By inverting T-CAST on C.27.59:

$$\Gamma;\, \Sigma \vdash \ell\colon \mathsf{T}_1 \tag{C.27.60}$$

$$\Gamma \vdash \mathsf{T} \tag{C.27.61}$$

$$\Gamma \vdash \mathsf{T}_1 \lesssim \mathsf{T} \tag{C.27.62}$$

By C.27.60 and C.27.62 we get (a) and (b), respectively.

$\mathcal{S}.\mathcal{H}$ does not evolve, which proves (c), given (ii).

- R-NEW: Fact (iii) has the form:

$$\mathcal{S};\, \mathbf{new}\; \mathsf{C}(\overline{v}) \longrightarrow \mathcal{S}';\, \ell \tag{C.27.63}$$

By inverting R-NEW on C.27.63:

$$\mathsf{H}(\ell_0) = \{\mathtt{name}\colon \mathsf{C}\,;\; \mathtt{proto}\colon \ell_0'\,;\; \mathtt{m}\colon \mathcal{M}\} \tag{C.27.64}$$

$$\mathsf{fields}(\mathcal{K},\, \mathsf{C}) = \overline{\mathsf{f}}\colon \overline{\mathsf{T}} \tag{C.27.65}$$

$$\mathcal{O} = \{\mathtt{proto}\colon \ell_0\,;\; \mathtt{f}\colon \overline{\mathsf{f}}{:=}\overline{v}\,;\; \ldots\} \tag{C.27.66}$$

$$\mathcal{H}' = \mathcal{H}[\,\ell \mapsto \mathcal{O}\,] \tag{C.27.67}$$

By (i) for $w \equiv \mathbf{new}\; \mathsf{C}(\overline{v})$ we have:

$$\Gamma;\, \Sigma \vdash \mathbf{new}\; \mathsf{C}(\overline{v})\colon \mathsf{T}_{2.0} \tag{C.27.68}$$

Where:

$$\mathsf{T}_{2.0} \equiv \exists\, \overline{y}_{\circ} \colon \overline{\mathsf{T}}_{\circ} \,.\, \{\, \nu \colon \mathsf{C} \mid \nu.\overline{f} = \overline{y}_{\circ} \wedge \mathsf{inv}(\mathsf{C}, \nu) \,\} \tag{C.27.69}$$

By inverting T-NEW on C.27.68:

$$\Gamma \vdash \overline{\nu} : (\overline{\mathsf{T}}_{\circ}, \overline{\mathsf{T}}_{\square}) \tag{C.27.70}$$

$$\vdash \mathsf{class}(\mathsf{C}) \tag{C.27.71}$$

$$\Gamma,\, y \colon \mathsf{C} \vdash \mathsf{fields}(y) = \circ\, \overline{f} \colon \overline{\mathsf{T}_2},\, \square\, \overline{g} \colon \overline{\mathsf{T}_3} \tag{C.27.72}$$

$$\Gamma,\, y \colon \mathsf{C},\, \overline{y} \colon \overline{\mathsf{T}}, y.\overline{f} = \overline{y}_{\circ} \vdash \overline{\mathsf{T}}_{\circ} \leq \overline{\mathsf{T}_2},\, \overline{\mathsf{T}}_{\square} \leq \overline{\mathsf{T}_3},\, \mathsf{inv}(\mathsf{C}, y) \tag{C.27.73}$$

For fresh y and $\overline{y}$.

We choose a heap typing Σ', such that:

$$\Sigma' = \Sigma[\, \ell \mapsto \mathsf{T}_{2.0} \,]$$

Hence:

$$\Sigma'(\ell) = \mathsf{T}_{2.0} \tag{C.27.74}$$

By applying Rule RT-T-LOC using C.27.74:

$$\Gamma;\, \Sigma' \vdash \ell \colon \mathsf{T}_{2.0}$$

Which proves (a).

By applying Rule RT-T-OBJ using C.27.74, C.27.66 and C.27.70:

$$\mathcal{S} \vdash_{\Sigma} \mathcal{O} : \mathsf{T}_{2.0} \tag{C.27.75}$$

By $\leq$-ID we trivially get:

$$\Gamma \vdash \mathsf{T}_{2.0} \leq \mathsf{T}_{2.0} \tag{C.27.76}$$

Which proves (b).

By applying Rule WF-HEAP-INST on C.27.66, C.27.64, C.27.74, C.27.72, C.27.70 and C.27.73:

$$\Sigma' \vdash \mathcal{S}'.\mathcal{H}$$

Which proves (c).

- R-LETIN *Similar approach to case* R-CALL.
- R-DOTASGN: Fact (ii) has the form:

$$\mathcal{S};\, \ell.\mathsf{g}_\mathsf{i} \leftarrow v' \longrightarrow \mathcal{S}';\, v' \tag{C.27.77}$$

By inverting Rule R-DOTASGN on C.27.77:

$$\mathcal{H}' = \mathcal{S}.\mathcal{H}[\,\ell \mapsto \mathcal{S}.\mathcal{H}(\ell)[\,\mathsf{g}_\mathsf{i} \mapsto v'\,]\,] \tag{C.27.78}$$

From (i) for $w \equiv \ell.\mathsf{g}_\mathsf{i} \leftarrow v'$:

$$\Gamma;\, \Sigma \vdash \ell.\mathsf{g}_\mathsf{i} \leftarrow v' \colon \mathsf{T}' \tag{C.27.79}$$

By inverting Rule T-DOTASGN on C.27.79:

$$\Gamma; \Sigma \vdash \ell : \mathsf{T}_\ell, v' : \mathsf{T}' \tag{C.27.80}$$

$$\Gamma,\, \mathsf{y} \colon \lfloor \mathsf{T}_\ell \rfloor; \Sigma \vdash \mathsf{y}\ \mathsf{hasMut}\ \mathsf{g}_\mathsf{i} \colon \mathsf{T}_{3\mathsf{i}},\, \mathsf{T}' \leq \mathsf{T}_{3\mathsf{i}} \tag{C.27.81}$$

For a fresh y.

By C.27.80 and SUB-REFL we prove (a) and (b).

By inverting RT-T-LOC on C.27.80:

$$\Sigma(\ell) = \mathsf{T}_\ell \tag{C.27.82}$$

By inverting WF-HEAP-INST on (iii) for location ℓ and using C.27.82:

$$\mathcal{O} \doteq \{\mathtt{proto}\colon \ell';\ \mathtt{f}\colon \vec{\mathcal{F}};\ \ldots\} \tag{C.27.83}$$

$$\vec{\mathcal{F}} \doteq \circ\,\overline{\mathsf{f}} := \overline{v}_\circ,\ \square\,\overline{\mathsf{g}} := \overline{v}_\square \tag{C.27.84}$$

$$\lfloor \Sigma(\ell) \rfloor = \mathsf{C} \tag{C.27.85}$$

$$\Gamma,\, \mathsf{y} \colon \mathsf{C} \vdash \mathsf{fields}(\mathsf{y}) = \circ\,\overline{\mathsf{f}} \colon \overline{\mathsf{T}_2},\ \square\,\overline{\mathsf{g}} \colon \overline{\mathsf{T}_3} \tag{C.27.86}$$

$$\Sigma \vdash \overline{v}_\circ : \overline{\mathsf{T}}_\circ \tag{C.27.87}$$

$$\Sigma \vdash \overline{v}_\square : \overline{\mathsf{T}}_\square \tag{C.27.88}$$

$$\Gamma,\, \mathsf{y} \colon \mathsf{C}, \overline{\mathsf{y}}_\circ \colon \mathsf{sngl}\left(\overline{\mathsf{T}}_\circ,\, \mathsf{y}.\overline{\mathsf{f}}\right) \vdash \overline{\mathsf{T}}_\circ \leq \overline{\mathsf{T}_2},\ \overline{\mathsf{T}}_\square \leq \overline{\mathsf{T}_3},\ \mathsf{inv}(\mathsf{C},\, \mathsf{y}) \tag{C.27.89}$$

Fact C.27.78 becomes:

$$\mathcal{H}' = \mathcal{S}.\mathcal{H}[\,\ell \mapsto \mathcal{O}'\,] \tag{C.27.90}$$
$$\mathcal{O}' = \{\texttt{proto}\colon \ell'\,;\ \texttt{f}\colon \vec{\mathcal{F}}'\,;\ \ldots\} \tag{C.27.91}$$
$$\vec{\mathcal{F}}' = \circ\,\overline{\texttt{f}} := \overline{v}_{\circ},\ \square\,\overline{\texttt{g}} := \overline{v}'_{\square} \tag{C.27.92}$$
$$\overline{v}'_{\square} = \overline{v}_{\square,..\texttt{i}-1}, v'_{\square,\texttt{i}}, \overline{v}_{\square,\texttt{i}+1..} \tag{C.27.93}$$

Also by C.27.80 and C.27.88 it holds that:

$$\Sigma \vdash \overline{v}'_{\square} : (\overline{T}_{\square,..\texttt{i}-1}, T', \overline{T}_{\square,\texttt{i}+1..}) \tag{C.27.94}$$

By Lemma C.17 on C.27.81:

$$\Gamma,\, y\colon C, \overline{y}_{\circ}\colon \mathsf{sngl}\left(\overline{T}_{\circ},\, y.\overline{f}\right); \Sigma \vdash T' \leq T_{3\texttt{i}} \tag{C.27.95}$$

By applying Rule WF-HEAP-INST on C.27.91, C.27.92, C.27.85, C.27.86, C.27.87, C.27.94, C.27.89 and C.27.95:

$$\Sigma \vdash \mathcal{H}'$$

Which proves (c).

- R-LIF: Assume $n \equiv$ **true** (the case for **false** is symmetric).

Fact (ii) has the form:

$$\mathcal{S};\ \textbf{if}\,[\,\overline{x},\,\overline{x}_1,\,\overline{x}_2\,]\,\textbf{true then}\ u_1\langle w\rangle\ \textbf{else}\ u_2\langle w\rangle \longrightarrow \mathcal{S};\ u_1\langle[\overline{x}_1/\overline{x}]\,(w)\rangle \tag{C.27.96}$$

By Rule T-CTX fact (i) has the form:

$$\Gamma \vdash \textbf{if}\,[\,\overline{x},\,\overline{x}_1,\,\overline{x}_2\,]\,\textbf{true then}\ u_1\langle w\rangle\ \textbf{else}\ u_2\langle w\rangle : \exists\,\overline{x}\colon \overline{T}_1\,.\,T_2 \tag{C.27.97}$$

So type T has the form:

$$T \equiv \exists\,\overline{x}\colon \overline{T}_1\,.\,T_2 \tag{C.27.98}$$

By inverting Rule T-CTX on (i):

$$\Gamma \vdash \textbf{if}\,[\,\overline{x},\,\overline{x}_1,\,\overline{x}_2\,]\,\textbf{true then}\ u_1\langle w\rangle\ \textbf{else}\ u_2\langle w\rangle \triangleright \overline{x}\colon \overline{T}_1 \tag{C.27.99}$$
$$\Gamma, \overline{x}\colon \overline{T}_1 \vdash w : T_2 \tag{C.27.100}$$

By inverting Ryle T-LetIf on C.27.99:

$$\Gamma \vdash \textbf{true} : T_1,\ T_1 \ \leq\ \mathsf{Bool} \tag{C.27.101}$$
$$\Gamma,\ y\colon T_1,\ y \vdash u_1 \triangleright \Gamma_1 \tag{C.27.102}$$
$$\Gamma,\ y\colon T_1,\ \neg y \vdash u_2 \triangleright \Gamma_2 \tag{C.27.103}$$
$$\Gamma,\ \Gamma_1 \vdash \Gamma_1(\overline{x}_1) \leq \overline{T}_1 \tag{C.27.104}$$
$$\Gamma,\ \Gamma_2 \vdash \Gamma_2(\overline{x}_2) \leq \overline{T}_1 \tag{C.27.105}$$
$$\Gamma \vdash \overline{T}_1 \tag{C.27.106}$$

By Rule T-Cst on **true**:

$$\Gamma \vdash \textbf{true} : \{\, v\colon \mathsf{Bool} \mid v = \textbf{true} \,\} \tag{C.27.107}$$

By Lemma C.26 on C.27.101 and C.27.102:

$$\Gamma \vdash u_1 \triangleright \Gamma_1 \tag{C.27.108}$$

Environment Γ_1 has the form:

$$\Gamma_1 \equiv \overline{x}_1 : \Gamma_1(\overline{x}_1), \overline{x}_1' : \Gamma_1(\overline{x}_1') \tag{C.27.109}$$

For some $\overline{x}_1'$.

By Lemma C.16 using C.27.100:

$$\Gamma, \overline{x}_1 : \overline{T}_1 \vdash [\overline{x}_1/\overline{x}]\,(w) : [\overline{x}_1/\overline{x}]\,(T_2) \tag{C.27.110}$$

By Lemma C.18 using C.27.110:

$$\Gamma, \overline{x}_1 : \overline{T}_1, \overline{x}_1' : \Gamma_1(\overline{x}_1') \vdash [\overline{x}_1/\overline{x}]\,(w) : [\overline{x}_1/\overline{x}]\,(T_2) \tag{C.27.111}$$

By applying rule T-Ctx on C.27.108 and C.27.111:

$$\Gamma \vdash u\langle [\overline{x}_1/\overline{x}]\,(w)\rangle : \exists\, \overline{x}_1 : \Gamma_1(\overline{x}_1)\,.\,\exists\, \overline{x}_1' : \Gamma_1(\overline{x}_1')\,.\,[\overline{x}_1/\overline{x}]\,(T_2) \tag{C.27.112}$$

Which proves (a).

Fact C.27.112 can be rewritten as:

$$\Gamma \vdash u\langle [\overline{x}_1/\overline{x}]\,(w)\rangle : \exists\, \overline{x}\colon \Gamma_1(\overline{x})\,.\,\exists\, \overline{x}_1' : \Gamma_1(\overline{x}_1')\,.\,T_2 \tag{C.27.113}$$

Applying Rule SUB-BIND using C.27.113:

$$\Gamma \vdash \exists \overline{x}\colon \Gamma_1(\overline{x})\,.\,\exists \overline{x}_1'\colon \Gamma_1(\overline{x}_1')\,.\,T_2 \leq \exists \overline{x}\colon \Gamma_1(\overline{x})\,.\,T_2 \tag{C.27.114}$$

By Lemma C.25 on the right-hand side of C.27.114:

$$\Gamma \vdash \exists \overline{x}\colon \Gamma_1(\overline{x})\,.\,T_2 \leq \exists \overline{x}\colon T_1\,.\,T_2 \tag{C.27.115}$$

By C.27.113, C.27.114 and C.27.115, and using Rule SUB-TRANS we prove (b).

Heap $\mathcal{S}.\mathcal{H}$ does not evolve so (c) holds trivially.

□

Theorem C.28 (Progress)**.** *If*

(i) $\Gamma; \Sigma \vdash e\colon T$,

(ii) $\Sigma \vdash \mathcal{H}$

then one of the following holds:

(a) e *is a value,*

(b) there exist e', $\mathcal{H}'$ *and* $\Sigma' \supseteq \Sigma$ *s.t.* $\Sigma' \vdash \mathcal{H}'$ *and* $\mathcal{H}; e \longrightarrow \mathcal{H}'; e'$.

Proof. We proceed by induction on the structure of derivation (i):

- T-FLD-I:

$$\Gamma; \Sigma \vdash e_0.f_i\colon \exists y\colon T_0\,.\,\mathsf{sngl}\,(T, y.f_i) \tag{C.28.1}$$

By inverting T-FLD-I on C.28.1:

$$\Gamma; \Sigma \vdash e_0\colon T_0 \tag{C.28.2}$$

$$\Gamma, y\colon T_0; \Sigma \vdash y\ \mathsf{hasImm}\ f_i\colon T \tag{C.28.3}$$

By i.h. using C.28.2 and (ii) there are two possible cases on e_0:

- ♦ $e_0 \equiv \ell_0$ Statement C.28.2 becomes:

$$\Gamma; \Sigma \vdash \ell_0\colon T_0 \tag{C.28.4}$$

By (ii) for location ℓ_0:

$$\Sigma \vdash \mathcal{H}[\,\ell_0 \mapsto \mathcal{O}\,] \tag{C.28.5}$$

Where:

$$\mathcal{O} \equiv \{\texttt{proto}: \ell_0'; \texttt{f}: \vec{\mathcal{F}}; \ldots\} \tag{C.28.6}$$

By Lemma C.19 using (ii) and C.28.5:

$$\Sigma(\ell_0) = \mathsf{T}_0 \tag{C.28.7}$$

$$\Gamma; \Sigma \vdash \mathcal{O}: \mathsf{T}_0', \mathsf{T}_0' \leq \mathsf{T}_0 \tag{C.28.8}$$

By Lemma A.6 in [76] using C.28.3 and C.28.8:

$$\Gamma, \mathtt{y}: \mathsf{T}_0'; \Sigma \vdash \mathtt{y} \ \mathsf{hasImm} \ \mathtt{f_i}: \mathsf{T} \tag{C.28.9}$$

By applying Rule R-FIELD using C.28.5, C.28.6 and C.28.9:

$$\mathcal{H}; \ell_0.\mathtt{f_i} \longrightarrow \mathcal{H}; v_i$$

♦ $\exists e_0'$ *s.t.* $\mathcal{H}; e_0 \longrightarrow \mathcal{H}'; e_0'$ By applying Rule RC-ECTX:

$$\mathcal{H}; e_0.\mathtt{f_i} \longrightarrow \mathcal{H}'; e_0'.\mathtt{f_i}$$

- T-FLD-M: *Similar to previous case.*
- T-MTH-CALL, T-NEW: *Similar to the respective case of CFJ [76].*
- T-CAST:

$$\Gamma; \Sigma \vdash e_0 \ \textbf{as} \ \mathsf{T}: \mathsf{T} \tag{C.28.10}$$

By inverting T-CAST on C.28.10:

$$\Gamma \vdash e_0 : \mathsf{T}_0' \tag{C.28.11}$$

$$\Gamma; \Sigma \vdash \mathsf{T} \tag{C.28.12}$$

$$\Gamma; \Sigma \vdash \mathsf{T}_0' \lesssim \mathsf{T} \tag{C.28.13}$$

By i.h. using C.28.11 and (ii) there are two possible cases on e_0:

♦ $e_0 \equiv \ell_0$ Statement C.28.11 becomes:

$$\Gamma; \Sigma \vdash \ell_0: \mathsf{T}_0' \tag{C.28.14}$$

By Lemma C.21 using (ii) and C.28.13:

$$\Gamma; \Sigma \vdash \mathcal{H}(\ell_0) \colon \mathsf{T}_0'', \mathsf{T}_0'' \leq \mathsf{T} \tag{C.28.15}$$

From R-CAST using C.28.15:

$$\mathcal{H};\ \ell_0\ \textbf{as}\ \mathsf{T} \longrightarrow \mathcal{H};\ \ell_0$$

♦ $\exists e_0'$ *s.t.* $\mathcal{H};\ e_0 \longrightarrow \mathcal{H}';\ e_0'$ By rule RC-ECTX:

$$\mathcal{H};\ e_0\ \textbf{as}\ \mathsf{T} \longrightarrow \mathcal{H}';\ e_0'\ \textbf{as}\ \mathsf{T}$$

- T-LET, T-DOTASGN, T-IF *These cases are handled in a similar manner.*

□

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[2] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507.

[3] A. Aiken. Introduction to Set Constraint-based Program Analysis. *Sci. Comput. Program.*, 35(2-3):79–111, Nov. 1999.

[4] A. Aiken and E. L. Wimmers. Solving systems of set constraints. *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[5] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 163–173.

[6] D. Ancona and A. Corradi. Semantic subtyping for imperative object-oriented languages. pages 568–587.

[7] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 428–452.

[8] E. Andreasen and A. Møller. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 17–31.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems*, 33(2), Feb. 2011.

[10] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 257–281.

[11] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 105–116.

[12] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt,

and G. Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100.

[13] M. Bostock, V. Ogievetsky, and J. Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[14] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3), June 2005.

[15] R. Cartwright and M. Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292.

[16] G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 182–192.

[17] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 378–391.

[18] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 410–429.

[19] W. Choi, S. Chandra, G. C. Necula, and K. Sen. SJS: A Type System for JavaScript with Fixed Object Layout. In *SAS*, volume 9291 of *Lecture Notes in Computer Science*, pages 181–198.

[20] R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606.

[21] R. Chugh, P. M. Rondon, and R. Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 231–244.

[22] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71.

[23] Closure. Closure Compiler. https://developers.google.com/closure/compiler/, 2009. Accessed: 2016-11-15.

[24] Cognitect Labs. https://github.com/cognitect-labs/transducers-js.

[25] P. Cousot. Types As Abstract Interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 316–331.

[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static

Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252.

[27] L. Damas and R. Milner. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212.

[28] Dart. Dart Language Specification. https://www.dartlang.org/guides/language/spec, 2011. Accessed: 2016-11-15.

[29] S. Dolan and A. Mycroft. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 60–72.

[30] J. Dunfield. Untangling Typechecking of Intersections and Unions. In *Proceedings of the Workshop on Intersection Types and Related Systems (ITRS '10)*, volume 45 of *EPTCS*, pages 59–70. arXiv:1101.4428[cs.PL].

[31] J. Dunfield. Elaborating intersection and union types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, pages 17–28.

[32] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *ACM SIGPLAN Notices*, volume 30, pages 169–184.

[33] M. Fähndrich and A. Aiken. Making Set-Constraint Program Analyses Scale. Technical report, Berkeley, CA, USA, 1996.

[34] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 85–96.

[35] A. Feldthaus and A. Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 1–16.

[36] R. B. Findler and M. Felleisen. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59.

[37] C. Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[38] C. Flanagan and M. Felleisen. Componential Set-based Analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, Mar. 1999.

[39] C. Flanagan, R. Joshi, and K. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2):97 – 108, 2001.

[40] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming*

Language Design and Implementation, PLDI '02, pages 234–245.

[41] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.

[42] T. Freeman and F. Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277.

[43] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19:1–19:64, Sept. 2008.

[44] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866.

[45] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44.

[46] Google Developers. https://developers.google.com/octane/.

[47] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 21–40.

[48] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee References for Refinement Types over Aliased Mutable Data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 73–84.

[49] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, pages 151–168.

[50] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, pages 256–275.

[51] C. Haack and E. Poll. Type-Based Object Immutability with Flexible Initialization. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 520–545.

[52] Hack. Hack Language Specification. https://github.com/hhvm/hack-langspec, 2014. Accessed: 2016-11-15.

[53] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6183 LNCS:200–224, 2010.

[54] T. S. Heinze, A. Møller, and F. Strocco. Type Safety Analysis for Dart. In *Proceedings of the*

12th Symposium on Dynamic Languages, pages 1–12.

[55] F. Henglein and J. Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 192–203.

[56] D. Herman and C. Flanagan. Status Report: Specifying JavaScript with ML. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 47–52.

[57] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[58] K. A. Jafery and J. Dunfield. Sums of uncertainty: Refinements go gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 804–817.

[59] D. Jang and K.-M. Choe. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1930–1937.

[60] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5673 LNCS:238–255, 2009.

[61] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 320–339.

[62] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69.

[63] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132.

[64] A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 296–309.

[65] K. Knowles and C. Flanagan. Compositional Reasoning and Decidable Checking for Dependent Contract Types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 27–38.

[66] K. Knowles and C. Flanagan. Hybrid Type Checking. *ACM Transactions on Programming Languages and Systems*, 32(2):6:1–6:34, Feb. 2010.

[67] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, volume 10.

[68] N. Lehmann and E. Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 775–788.

[69] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. pages 1–16.

[70] S. Maffeis, J. C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325.

[71] Microsoft Corporation. TypeScript v1.4. http://www.typescriptlang.org/.

[72] F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 334–359.

[73] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[74] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[75] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 139–152.

[76] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained Types for Object-oriented Languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 457–474.

[77] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 146–161.

[78] C. Park and S. Ryu. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 735–756.

[79] D. Park, A. Stefănescu, and G. Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356.

[80] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-16028.

[81] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1): 1–44, Jan. 2000.

[82] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 1–16.

[83] F. Pottier. Type Inference in the Presence of Subtyping: from Theory to Practice. Research Report 3483, INRIA, Sept. 1998.

[84] F. Pottier. Simplifying Subtyping Constraints: a Theory. *Information & Computation*, 170(2): 153–183, Nov. 2001.

[85] X. Qi and A. C. Myers. Masked Types for Sound Object Initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65.

[86] T. Racket. The Typed Racket Guide. https://docs.racket-lang.org/ts-guide/, 2011. Accessed: 2017-06-06.

[87] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 167–180.

[88] J. C. Reynolds. Automatic computation of data set definitions.

[89] J. C. Reynolds. ALGOL-like Languages, Volume 1. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-like Languages*, chapter Design of the Programming Language FORSYTHE. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[90] G. Richards, F. Z. Nardelli, and J. Vitek. Concrete Types for TypeScript. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100.

[91] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169.

[92] E. L. Seidel, N. Vazou, and R. Jhala. Type Targeted Testing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pages 812–836.

[93] T. F. Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu. The K primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304:57–80, 2014.

[94] J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*, ECOOP '07, pages 2–27.

[95] F. Smith, D. Walker, and J. G. Morrisett. Alias Types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381.

[96] SMTLIB. The Satisfiability Modulo Theories Library. http://smt-lib.org.

[97] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-to Analysis of Javascript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458.

[98] Z. Su, M. Fähndrich, and A. Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '00, pages 81–95.

[99] Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. The First-order Theory of Subtyping Constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–216.

[100] A. J. Summers and P. Mueller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1013–1032.

[101] O. Tardieu, N. Nystrom, I. Peshansky, and V. Saraswat. Constrained Kinds. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 811–830.

[102] P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, pages 408–422.

[103] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406.

[104] S. Tobin-Hochstadt and M. Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 117–128.

[105] L. Tratt. Dynamically Typed Languages. *Advances in Computers*, 77:149–184, July 2009.

[106] V. Trifonov and S. Smith. Subtyping Constrained Types. *SAS '96: Proceedings of the 3rd International Symposium on Static Analysis*, pages 349–365, 1996.

[107] TypeScript. TypeScript Design Goals. https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals, 2012. Accessed: 2016-11-15.

[108] P. Vekris, B. Cosman, and R. Jhala. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52–75.

[109] WALA. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net, 2006.

[110] J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.

[111] A. K. Wright and R. Cartwright. A Practical Soft Type System for Scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, Jan. 1997.

[112] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[113] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227.

[114] B. Yankov. http://definitelytyped.org.

[115] T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 37–50.

[116] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kie, un, and M. D. Ernst. Object and Reference Immutability Using Java Generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84.

[117] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 598–617.

[118] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object Initialization in X10. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 207–231.