University of London
Imperial College London
Department of Computing

Type Inference for JavaScript

Christopher Lyon Anderson

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, March 2006

To my family.

Abstract

Object-oriented scripting languages like JavaScript are popular, in part because of their dynamic features. These include the runtime modification of objects and classes, through addition of fields or updating of methods. These features make static typing difficult and usually dynamic typing is used. Consequently, errors such as access to non-existent members, are not detected until runtime.

We provide a static type system that can cope with dynamic features such as member addition, while providing the usual safety guarantees. Since the structure of objects may change over time, we employ a structural type system to track the changes. We show how type inference can be used to infer the structure of objects and give corresponding structural types. Therefore, the programmer can enjoy the safety offered by static typing, without having to give explicit types in their programs.

We develop JS_0, a formalisation of JavaScript with features including dynamic addition of fields and updating of methods. We give an operational semantics and static type system for JS_0 using structural types. Our types allow objects to evolve in a controlled manner by classifying members as *definite* or *potential*. A member is potential until it has been assigned to and then it becomes definite. We prove that our type system is sound.

We develop a type inference algorithm for JS_0 based on a system of constraints between type variables. We define a translation between constraints and types that allows us to generate a type annotated JS_0 program from an untype JS_0 program. We prove that the constraints are sound with respect to the type system and that our translation is deterministic. We define a well-formedness criterion on constraints and conjecture that well-formed constraints are satisfiable. Therefore, combined with the soundness of constraints and the soundness of the type system, we conjecture that programs that generate well-formed constraints will not get stuck.

Acknowledgements

There are few people that you meet in your life that you know will shape your future. Sophia Drossopoulou, my supervisor, teacher and above all friend is one such person. I am very grateful that she wanted to work with me. Her unending enthusiasm has been an inspiration to me. As is her determination, to not only solve seemingly impossible problems, but also make them easy to understand. Thank you Sophia.

I would also like to thank my co-supervisor Susan Eisenbach. Her practical and positive advice have been very helpful. It was Susan who introduced me to Sophia back in 2000. She thought we would work well together. She was right.

It is said that all PhD students go through a depression about their work in the second year of study. I was no exception. I was very fortunate to meet and work with Paola Giannini. Her great ideas put my research back on course and lifted my spirits. I very much enjoyed my trips to Italy to work with her.

I would like to thank Professor Mariangiola Dezani-Ciancaglini for creating the wonderful type system for delta. Working with her was great fun and I learnt a lot of new things. I must also say that she has excellent taste in restaurants.

I would like to thank my parents for all the love, support and understanding they have given me throughout my life. I am indebt to them always. I thank my wonderful sisters, Katie and Lucy, who are wise beyond their years. I would also like to thank my surrogate parents Auntie Nonette and Uncle Mark. It was at their house on a Sunday afternoon back in 2001 that I finally decided to do all of this. I am also very proud to be uncle of Isabella and Oliver. Last but not least I would like to thank Brenda.

I would like to thank my examiners, Dr. Simon Gay and Dr. Maribel Fernandez, for a most interesting discussion, and their detailed and helpful remarks.

I am very fortunate to have a great set of friends who have supported me throughout the course of this thesis. From living with me, working with me, to listening to my moaning and of course socialising. My thanks goes to: Marcos, Simon (Smithy), Ian (Bonus), David (Ishra), John, Mike, Dan, Jodie, Alison, Matthew, Will, Jonny, Alex A, Alex B, Rob, Ash, Alexis, Niel, Mark S, Ross, Phil (Pips), Grant, Rich, James R, Kieren, Sam W, Tom (Brada), Jason, Tim, Simon E, Paul, Scott P, Simon, Paul C.

This thesis is in part dedicated to my dear granddad, Raffaele, who passed away during the course of this work.

Statement

The research reported in this thesis is work carried out by myself, or as a collaboration between Sophia Drossopoulou, Paola Giannini and myself. The initial concept of JS_0 was a joint development between Sophia Drossopoulou and myself. The design decisions on which language features to include were my own. In Chapter 4, Sections 4.1 and 4.2 are derived from joint work with Paola Giannini and Sophia Drossopoulou, and Section 4.3 is my own work. In Chapter 5, the initial ideas were developed with Sophia Drossopoulou and Paola Giannini. The main development of the type inference algorithm and formulation of the formal properties, Sections 5.4 to 5.9 is my own work. Chapter 6 is my own work.

Contents

List of Figures

Glossary of Judgements

Judgement	Meaning	Figure/Definition	Page
$\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'$	Operational Semantics for expressions	3.6	39
$\vdash \mathsf{t} \diamond$	Well-formed types	4.4	48
$\mathsf{t} \equiv \mathsf{t}'$	Congruence	4.5	49
$\mathsf{t} \leq \mathsf{t}'$	Subtyping	4.3	50
$\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \mid\mid \Gamma'$	Typing of Expressions	4.11	53
$\vdash \mathbb{P} \diamond$	Well-formed Programs	4.12	54
$\mathsf{t} \sqcup \mathsf{t}'$	Upper Bound of types	4.15	58
$\mathbb{P}, \mathcal{T} \vdash \mathsf{v} \prec \mathsf{t}$	Weak Agreement of Values	4.16	60
$\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t}$	Strong Agreement of Values	4.16	60
$\gamma \vdash \mathsf{e} \ : \ \mathbb{e} \mid\mid \gamma' \mid\mid \mathsf{C}$	Constraint Generation for Expressions	5.4, 5.5	92, 93
$\vdash \mathsf{P} : \ \mathsf{C}$	Constraint Generation for Programs	5.6	94
$\mathsf{C} \ \longrightarrow \ \mathsf{C}'$	Constraint Closure	5.8	119
$\vdash \mathsf{C} \diamond_{cl}$	Closed Constraints	6	113
$\mathsf{C} \vdash \mathsf{V} \diamond$	Well-formed Type Variable Mapping	7	122
$\vdash \mathsf{C} \diamond$	Well-formed Constraints	5.9	121
$\mathsf{C}, \mathsf{V}, \tau \rightarrow \mathsf{tp}$	Translation from Constraints to Types	5.10	123

1 Introduction

SCRIPTING LANGUAGES such as JavaScript[29] are increasingly popular, but lack the maturity of object-oriented languages like Java[38] and C++[16]. The aim of this thesis is to offer more safety guarantees to JavaScript programmers. We believe this will lead to reduced development time and more robust JavaScript code being deployed on the Internet.

We achieve this by formalising a system of types and type inference. Types are well established in the object-oriented programming community. There are mainstream languages, like Java and C++, with advanced type systems capable of guaranteeing the safe execution of code. The programmer is responsible for introducing the types, through classes, and then explicitly annotating the code with them. Code is then submitted to the compiler for compilation. One of the phases of compilation is type checking (sometimes referred to as semantics analysis). Any violation of the type system is flagged as an error and reported to the programmer. Only code with no type errors can be run. There is a large body of theoretical work concerned with types for object-oriented programming languages.

Type inference for object-oriented programming is perhaps less well known. In languages that don't provide a mechanism for annotating code with types, type inference can be used to infer the types. This takes the burden of writing types away from the programmer. It does, however, require a type system for the underlying language. Languages that use type inference include O'Caml[43] and in the functional world: Haskell[41] and ML[52]. Type inference can also be used behind the scenes where, for example, it has been used in the optimization of compiled code. Modern development environments also make use of type inference to give developers useful information about their code, *e.g.* which fields and methods a variable of an object type can access.

Why JavaScript? JavaScript was developed by Brendan Eich at Netscape. It was originally called LiveScript but for marketing purposes was renamed to JavaScript. "Bad press" and incompatibilities between implementations led Netscape (and

Sun) to standardise the core language, which led to the language ECMAScript, the official standardisation of JavaScript. It is commonplace to refer to ECMAScript as JavaScript. We chose JavaScript because of its popularity and its support for flexible program development. With flexible program development we have malleability of object structure, with member addition and removal, and dynamic typing. This flexibility has come at the cost of safety.

Why type inference for JavaScript? One way to give safety guarantees to JavaScript programmers would be to make them annotate their code with types. The code could then be checked before being deployed. This poses two questions: what kind of types would be suitable for JavaScript, and do we expect the programmer to write types in their code.

The first part of this thesis will answer the first question. The current type system of JavaScript is not powerful enough to give the kinds of safety guarantees we are looking for. Types only distinguish the base types (integers, floats and strings) from the object type and the function type. The object type is *coarse* in that it is unaware of the fields and methods of the objects it is typing. We will need a more advanced type system that is able to distinguish the structure of one object from another. This will allow us to determine statically the validity of a field access or a method call. Note that there is a tool called `jslint`[1] that will find some of the more basic errors including syntax errors and variable misspellings.

For the second question we believe the answer is no. The reason being that programmers use JavaScript because it's easy to hack together code. Having to write types would effectively slow down development. With type inference, we can give the safety guarantees offered by types without having to annotate code with types. The second part of this thesis will show how we can infer the types we developed in the first part of this thesis.

1.1 Example JavaScript Program

Consider the example JavaScript program in Figure 1.1. The code is used to disable a button on a webpage when it is clicked. We will discuss this example in more detail in Chapter 3, but for now it serves as a taste of the language.

[1]`www.jslint.com`

```
 1  function input(value) {
 2    this.value = value;this.disabled = 0;
 3    this;
 4  }
 5
 6  function form() {
 7    this.onSubmit = onSubmit;this;
 8  }
 9
10  function onSubmit() {
11    checkform(this);
12  }
13
14  function checkform(theform) {
15    theform.submit.disabled=1;
16    theform.submit.value = "Calculating..."
17  }
18  //Main
19  htmlform = new form();
20  htmlinput = new input("Infer");
21  htmlform.submit = htmlinput;
22  htmlform.onSubmit();
```

Figure 1.1.: Web Form Example

This code demonstrates some key features of developing code in JavaScript. Lines 19 and 20 demonstrate how functions are used to create objects. Line 2 shows how objects acquire members through assignment. Line 7 demonstrates how objects acquire methods through assignment of a function identifier to a member. Line 21 demonstrates adding a member to an object *after* it has been created. Another observation is that functions are being used for different roles: `input` and `form` are being used to construct objects, `onSubmit` is being used as a method and `checkForm` is being used as a global function.

If in the body of `checkForm` line 15 were mistakenly entered as `theform.submi.disabled=1`, the field access, `theform.submi`, would return an undefined value when the code is run. The subsequent access of `disable`, on the undefined value, would result in a runtime error. This error would only be detected when the code is running. In Chapter 5 we will show how type inference can be used to catch this kind of error.

1.2 Our Approach and Contributions

We now outline our approach and the contributions of this thesis.

1.2.1 JS$_0$ - A formalism of JavaScript

We define JS$_0$ a formalism of JavaScript that supports the standard JavaScript flexible features, *e.g.* functions creating objects and dynamic addition/reassignment of fields and methods. We give a structural operational semantics for JS$_0$ that gives the runtime behaviour of programs.

1.2.2 Types for JS$_0$

We define JS$_0^{\mathsf{T}}$ an explicitly typed version of JS$_0$. As we have seen in the example above the type system has to address the following features of JavaScript:

- Object structure is determined by assignment of members.
- Objects can have members added to them after they have been created.
- Methods are created by assigning functions to members.
- Methods can be shared among objects.
- Functions can have three different roles: creating objects, methods of objects and global functions.

Types in JS$_0^{\mathsf{T}}$ comprise object types, function types, and Int (the type of integers). Object types list the methods and fields present in an object, *e.g.* $\mu\ \alpha.[\mathtt{m_1} : (\mathtt{t_1}, \psi_1) \cdots \mathtt{m_n} : (\mathtt{t_n}, \psi_\mathtt{n})]$. We use the μ-binder to allow a type to refer to itself. Our type system permits objects to evolve in a controlled manner by allowing members to be added to an object after it has been created. This is achieved by annotating with ψ each member of an object type as either *potential* (through '$\circ$') or *definite* (through '$\bullet$'). Returning to the example above, we could give the return type of the function `input` as follows:

$$[\mathtt{value} : (\mathsf{String}, \bullet), \mathtt{disabled} : (\mathsf{Int}, \bullet)]$$

This type represents an object that has two members `value` and `disabled` of type Int and String respectively. Note that JS$_0$ is expression oriented, and the last expression in the body of a function is considered the return value. Thus, as `input` is being used to construct objects, we return the receiver, `this`, which has had the members `value` and `disabled` added to it.

Function types have the form $\mathsf{t} = \mu\ \alpha.(\mathsf{O} \times \mathsf{t}_1 \to \mathsf{t}_2)$ where O is the type of the receiver, t_1 is the type of the formal parameter and t_2 is the return type. As for object types, the bound variable α allows references to t within O, t_1 and t_2. Thus, $\mu\ \alpha.(\_ \times \_ \rightarrow \alpha)$ is a function that returns a value of the same type as the function itself.

A function can be used as a *global function* if its type does not make any requirements of its receiver. The type system is rich enough to allow typing of many JavaScript programs and at the same time prevents runtime errors such as access to non-existing members of objects. We prove that our type system is sound.

1.2.3 Type Inference for JS_0

We give a type inference algorithm that given an untyped JS_0 program will infer a typed JS_0^T version of the program. The algorithm will only succeed if the JS_0 program can be typed. One of the major challenges with type inference for JS_0 is determining the structure of objects from the code. The algorithm has to track the evolution of objects as they grow through addition of members.

We use *type variables* to express the - yet unknown - types of expressions. Returning to the example above the type variable $[\![\texttt{new form()}]\!]$ expresses the type of expression `new form()`. To keep track of how an object, referenced through a variable, changes we use *labeled* type variables. This allows us to give a type to a variable at a particular point in the code. For example, the type variable $[\![\texttt{this\_1}]\!]$ gives the type of the receiver, `this`, in a function at a particular point.

Constraints between type variables express the relationship between the types of expressions, *i.e.* which members a type must have and which expressions are subtypes of other expressions. For example:

1. $[\![\texttt{this\_input}]\!] \ \leq\ [\texttt{value} : ([\![\texttt{this\_input.value}]\!], \bullet)]$
2. $[\![\texttt{this\_input.value}]\!] \ \leq\ [\![\texttt{value\_input}]\!]$
3. $[\![\text{“}\texttt{Infer}\text{”}]\!] \ \leq\ [\![\texttt{value\_input}]\!]$

Constraint (1) expresses that the receiver in function `input` should have a member `value`. The type of this member comes from the type of $[\![\texttt{this\_input.value}]\!]$. Constraint (3) expresses that the type of the formal parameter in function `input` is a supertype of the type of $[\![\text{“}\texttt{Infer}\text{”}]\!]$. Naturally, we would expect the type of $[\![\text{“}\texttt{Infer}\text{”}]\!]$ to be a String type. Inspection of constraints (1),(2) and (3) tell us that the type of member `value` should be of String type.

A solution, S, to a set of constraints is a mapping from type variables to types, *e.g.* S ($[\![$“`Infer`”$]\!]$) = String. After the constraints have been generated for a program they are *closed*. Constraint closure is a process that makes the solutions (or lack of) explicit in the constraints. We show that the constraints are sound with respect to the type system.

We define a translation between constraints and types that allows us to generate a $\mathrm{JS}_0^{\mathsf{T}}$ type annotated program from a JS_0 program. We define a well-formedness criterion on constraints, and conjecture that well-formed constraints are satisfiable. Therefore, combined with the soundness of constraints we conjecture that programs that generate well-formed constraints will not get stuck.

1.2.4 Prototype Implementation

Using Python we develop a prototype implementation of our inference algorithm. The implementation takes a more practical interpretation of JS_0 and includes features not in the formalism of JS_0 but in JavaScript. These include multiple parameters, local variables, statements and a string base type. There is a web front-end so that the user can enter a program for testing. Furthermore, we develop a Latex output mode so that constraints and types for the examples could be included in this thesis.

1.3 Earlier Work

The idea of using types for more robust JavaScript development was explored in [11, 12] with a JavaScript like language called BabyJ. The approach there was different to the one described in this thesis. Code development starts in an object based style and through a series of iterations is translated to a class based style. Mapping BabyJ to Java allows for better code reuse, and therefore nominal types were used. The type system used is a subset of the one used in Java with the addition of the type *Any*. The *Any* type acts as an escape from the type system, *i.e.* the type system ignores variables with the *Any* type. The process of going from object based to class based involves replacing occurrences of the *Any* type with new types introduced through constructor functions. This process was facilitated using type inference.

At the time this work was developed there appeared to be a move towards a more Java like style of programming in JavaScript, *c.f.* JavaScript 2 [40]. However, it became clear that expecting programmers to effectively rewrite their code to get *some* safety guarantees was not proving popular. We base this on the lack of support for JavaScript 2 in web browsers. This lead us to develop the approach described in [13, 14] and this thesis.

1.4 Thesis Organisation

This thesis is organized as follows: In Chapter 2 we give background material relevant to the thesis. In Chapter 3 we introduce JS_0 our formalisation of a subset of JavaScript. In Chapter 4 we introduce the type system for JS_0. We prove that our type system is sound. In Chapter 5 we develop a type inference algorithm for the type system developed in Chapter 4. We prove that our type inference algorithm is sound with respect to the type system. We prove that when the algorithm succeeds for a given JS_0 program that program can be translated into a well-typed $\mathrm{JS}_0^{\mathrm{T}}$ program. In Chapter 6 we detail a prototype implementation of the type inference algorithm developed in Chapter 5. Finally, in Chapter 7 we draw conclusions and discuss future work.

2 Background

In this chapter we give the background material that will be useful throughout the thesis. We divided the discussion into three parts. First, we give an outline of types for programming languages through a *taxonomy* of types. While most of the terms and explanations will be familiar to the reader with experience of programming languages, it is useful to collect the myriad of definitions together into a coherent structure. Especially, as many of the terms are confused or misattributed to a particular programming language.

Second, we look at type inference and how it has been used in object-oriented programming languages. We present some of the standard notation that has evolved around the various formalisations of type inference.

Third, we look at scripting languages, and then pull together our discussion of types by looking at attempts to add types to JavaScript.

2.1 Paradigms in Object-Oriented Programming

Object oriented programming can be classified into the object based and the class based paradigm.

2.1.1 Object Based Programming

Object based languages, *e.g.* JavaScript[29], SELF[6] and Cecil[24], were introduced in the 90s to support flexibility and prototyping. Typically, they are weakly typed, do not distinguish between fields and methods, allow the removal and addition of members, do not support classes and store all behaviour (methods and fields) in the objects themselves. The first formal models of object oriented languages were object based, [1, 34], as the object based paradigm was considered more fundamental than the class based paradigm.

2.1.2 Class Based Programming

Class based languages, *e.g.* Smalltalk[37], were introduced in the late 70s. They distinguish between objects which hold fields, and classes which hold methods and impose an immutable class hierarchy and class membership. Although Smalltalk is weakly typed most class based languages, *e.g.* Java[38], C++[16] and C♯[30], are strongly typed and support robust programs, and offer more opportunities for optimizations at compile time. Classes and a class hierarchy give structure and can be used organise a program to give a clean external interface.

2.2 Type Systems

A type system is a set of type rules for a particular programming language. It is independent of a type checking algorithm, in that a type system can have *no* feasible type checking algorithm (it can even have no algorithm at all).

2.2.1 Type Rules

Judgements form the base blocks of type rules and assert validity of a statement. A judgement has the general form:

$$\Gamma \vdash A$$

Where A is an assertion whose free variables are declared in an environment Γ. The environment may comprise more than just free variables it may also contain the program itself. This may be required for a system that allows terms to reference other parts of the program. For example, referring to a function name in a function call may require the program in order to check the validity of the actual and formal parameters.

For a programming language there will be *typing judgements* that assert that a term has a particular type. For example:

$$\Gamma \vdash 5 : \mathsf{Int}$$

A judgement can also describe the *effect* that executing a term has thus, giving a judgement of the form (where ϕ is the effect):

$$\Gamma \vdash A : \phi$$

A type system with effects is termed a *Type and Effect System*. Note that sometimes the effect is subsumed into a new environment giving a judgement of the form: $\Gamma \vdash A : \Gamma'$. In this thesis our typing judgements require the program and will have an effect on the environment. For example, the judgement for typing an expression will have the form:

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \mid\mid \Gamma'$$

Note that type rules are comprised of judgements. A type rule has the form:

$$\frac{\Gamma_1 \ \vdash \ A_1 \ldots \Gamma_n \ \vdash \ A_n}{\Gamma \ \vdash \ A} \ (rulename)$$

The judgements above the line are called premises and the judgement below the line the line is called the conclusion. The conclusion holds if all the premises are true. By convention, each rule has a name. In the example above it was $(rulename)$. We now give a real type rule from our system as an example. Naturally, all the details will become clear in later sections, but for now it serves as an example. We give the rule for sequence of expressions:

$$\frac{\begin{array}{c}\mathsf{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{t} \mid\mid \Gamma' \\ \mathsf{P}, \Gamma' \vdash \mathsf{e}_2 : \mathsf{t}' \mid\mid \Gamma''\end{array}}{\mathsf{P}, \Gamma \vdash \mathsf{e}_1; \mathsf{e}_2 : \mathsf{t}' \mid\mid \Gamma''} \ (seq)$$

Type rules are not required to have premises for example:

$$\frac{}{\mathsf{P}, \Gamma \vdash n : \mathsf{Int} \mid\mid \Gamma} \ (var)$$

There will also be type rules for well-formedness of the environment, and well-formedness of a program. A *derivation* is a tree of type judgements. We call a collection of type rules a *type system*.

2.2.2 Safety of a Type System

A safe type system guarantees that no untrapped errors will occur. Safety can be demonstrated by showing that execution of a well typed expression will return a value of that type, unless it diverges or produces a trapped error *e.g.* NullPointerException or DivisionByZero[68, 67].

Alternatively, safety can be demonstrated by a subject reduction theorem which

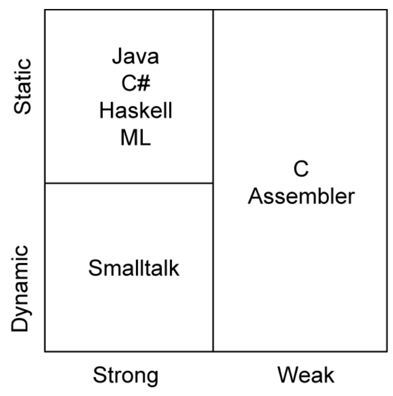

Figure 2.1.: Languages with respect to Strong vs Weak and Dynamic vs Static

shows that each execution step preserves the type of the expression [72]. In [28, 27] Drossopoulou et al. show subject reduction for a subset of Java.

2.3 Taxonomy of Types in Program Languages

There are many criteria applicable to classification of type systems for programming languages. We now discuss three of them.

2.3.1 Strong vs Weak

The amount of errors detected by a type system characterises how strong (weak) it is. In [21] Cardelli classifies errors into *trapped errors* and *untrapped errors*. Trapped errors are those that stop computation such as division by zero or accessing an illegal area of memory. Untrapped errors are those errors that go undetected such as accessing past the end of an array in C[42]. Strength ranges between *strong* and *weak*.

Strong Typing- Strong type systems prevent untrapped errors from occurring and possibly also a subset of trapped errors. For example, mixing operations between mismatched types like adding a `String` to an `int` is usually forbidden. Languages with strong typing are Java, Haskell[41], ML[52] and Smalltalk.

Weak Typing- Weak type systems have no enforcement of types, with values assigned arbitrarily. Variables can be implicitly coerced to unrelated types. At the most extreme we have assembler languages where operations are performed on regions of memory with no checking. Languages with weak typing are C and Perl[70].

2.3.2 Dynamic vs Static

The time at which errors are detected by a type system characterises whether it is static or dynamic.

Dynamic Typing- Dynamic type systems assign a type to a value at runtime and perform type checking at runtime. Languages with dynamic typing are Python[46] and Smalltalk. Note that Java has some dynamic typing with respect to casting and arrays.

Static Typing- Static type systems associate types with values at compile-time. Variables are constrained with respect to the type of the value they can refer to and checks are done at compile time. Languages with static typing are Java, C++ and Haskell.

2.3.3 Implicit vs Explicit

The amount of explicit type information required in a program characterises whether the type system is explicit or implicit.

Explicit Typing- Explicit type systems require a type annotation to be given explicitly for each declaration, *i.e.* types are ”manifest” in the source code. Types in the source can be used to enhance readability and provide documentation. For example, Java documentation can be produced through the JavaDoc program. Type declarations can provide a clean external interface allowing more interopability between languages. Languages with explicit typing are Java, C♯ and C++.

Implicit Typing- Implicit type systems require little or no type annotations in the source. Languages with no annotations are often referred to as being untyped. Two types of implicit typing exist:

- Type Inference - Types are inferred and then checked by the type system before the program is executed. An example of an object-oriented language with type inference is OCaml[43]. It is more popular in functional languages, *e.g.* Haskell and ML.

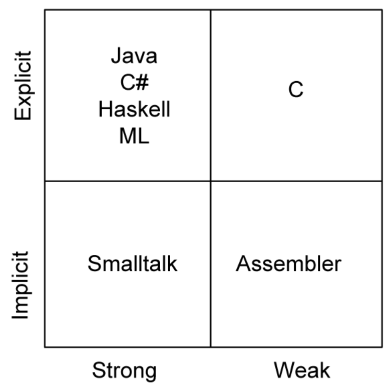

Figure 2.2.: Languages with respect to Strong vs Weak and Implicit vs Explicit

- Dynamic Typing - The runtime system knows the type of each entity at runtime. This information can then be used by the type system. For example, PHP[15] has an internal (but changeable) flag that tells if a variable is a string or number. Languages with Dynamic Typing are Lisp[39], Smalltalk, Python, Ruby[47] and PHP.

2.4 Subtyping

Subtyping describes inclusion between types in a type system. Intuitively, it characterises when a value of one type can be considered a value of another type. A type system will have a judgement for subtyping for example:

$$t \leq t'$$

One can include the subtype judgement in the type rules that would *benefit* from subtyping or include a separate type rule called a subsumption rule. By benefit, we mean cases in the programming language where using a subtype of the specified type would be practically useful. For example, consider function f:

```
function f(x:t₂) {
...
}
```

In any reasonable language with subtyping, if $t_1 \leq t_2$ then we would expect to be able to pass a value of t_1 to function `f`. Hence, the type rule for function application/call will permit the actual parameter to be a subtype of the formal parameter.

Subtyping adds great flexibility as functions can accept values of types added in the *future*.

One important issue that affects subtyping is how the type system handles equivalence between types. Next we shall discuss the two kinds of equivalence: name equivalence and structural equivalence. Type equivalence is sometimes referred to as type identity. We shall denote name equivalence by $\equiv_N$ and structural equivalence by $\equiv_S$.

2.4.1 Name Equivalence

With name equivalence two types are considered the same if they have the same name. In [23] it is noted that this definition of name equivalence is not quite correct but the one adopted in general. The real definition is that each occurrence of a type constructor produces a new type. As with [23] we go with the popular definition as it is more intuitive. Consider the following example:

$$\mathsf{t}_1 = \mathsf{Int}$$
$$\mathsf{t}_2 = \mathsf{Int}$$

Although t_1 and t_2 are both integer types, Int, they have different names and therefore, $\mathsf{t}_1 \not\equiv_N \mathsf{t}_2$. From a software engineering perspective one could argue that this is good behaviour. To allow assignment between t_1 and t_2 could be the indication of a mistake in the program. Furthermore, either of the types could be redefined at a later stage resulting in unpredictable results or code breakage. There is a tradeoff between safety and flexibility. Some languages, *e.g.* Modula-3[22], have a mixture of name and structural equivalence whereby t_1 and t_2 would be equivalent but richer types, *e.g.* object types, have to have the same name.

Programming languages that use name equivalence include Java, C++ and C♯.

In languages with name equivalence values with the same type are subtypes of each other. Some languages allow the programmer to explicitly specify which types are subtypes. A good example, is the inheritance mechanism in Java where a class can inherit from another class. At this point they become subtypes, *and* code is inherited from the supertype to the subtype. For example:

```
class A {
  int a;
  void m() { ... }
}

class B ext A {
}

x:A = new B();
x.m;
```

Here type B is a subtype of type A, and therefore, x can access any of the members defined in A. In Cecil subtyping and code inheritance are separated. Therefore, the programmer must specify which types are subtypes and/or if code is inherited.

2.4.2 Structural Equivalence

With structural equivalence two types are considered the same if their structures are the same. In the example above, t_1 and t_2 would be considered equivalent, $t_1 \equiv_S t_2$. Structural equivalence becomes interesting when applied to types with rich structure such as object types. We now borrow our notation for object types, from Chapter 4, for the following example:

$$t_3 = [\mathsf{m}_1 : \mathsf{Int}]$$
$$t_4 = [\mathsf{m}_1 : \mathsf{Int}]$$

Types t_3 and t_4 are structurally equivalent, $t_3 \equiv_S t_4$, because they both have a member m_1 with type Int. With recursive types the types must be *unfolded* to ensure that they are structurally the same, for example:

$$t_5 = \mu\, \alpha.[\mathsf{m}_1 : \alpha]$$
$$t_6 = \mu\, \alpha.[\mathsf{m}_1 : \mu\, \alpha.[\mathsf{m}_1 : \alpha]]$$

Types t_5 and t_6 are equivalent because t_5 is structurally the same as t_6 with substitution of α for t_5 in t_5. This process is called unfolding. In [8] Amadio and Cardelli give a thorough treatment of the issues surrounding equivalence of recursive structural types.

In the above example we have given names to the types, t_3, t_4..., but we are discussing structural equivalence! One should view the names as *sugar* to aid in the presentation. The names are only a way to refer to the type, they have no other semantics. This works well in a distributed environments where two equivalent types with *different* names can be used together.

In languages with structural equivalence subtyping becomes a property directly between the types. This becomes interesting when looking at object types. For example:

```
function printName(x:[name : String]) { ... x.name ... }

function checkAge(x:[age : Int]) { ... x.age ... }

var p:[name : String, age : Int]

printName(p); // Ok
checkAge(p); // Ok
```

As expected, variable `p` can be passed to both `printName` and `checkAge`. Each function specifies a formal parameter type that mentions only those members used in the function body. This has the advantage of allowing function to work on any object type that has the appropriate members. We will define a structural type system and subtypes for a JavaScript like language in Chapter 4.

2.5 Type Inference for Object-Oriented Languages

Type inference is the process of finding a type for a term within a type system. In type systems that have type annotations the type inference process may only involve checking the consistency of those annotations. More difficult, is type inference in untyped languages which is sometimes referred to as *type reconstruction*[21]. This involves taking an untyped program, P, and an expression, e, then generating an annotated version of P, $\mathbb{P}$, an environment, Γ, and finding a type, t, for e in $\mathbb{P}$. In this thesis our aim is to give a guarantee that a JavaScript program will not go *wrong* (by showing it can be typed). As the JavaScript program will be untyped our problem is one of *type reconstruction*. We chose to use the term *type inference* as in practice it covers type reconstruction too.

The type inference problem can be expressed as a constraint system between type variables. If the constraints have a solution we say that they are satisfiable. Type variables express the - yet unknown - type of expressions in a program. As in [56, 7, 57, 55] we denote the type variable for an expression, e, with $[\![\mathsf{e}]\!]$. Constraints between type variables express the relationship between the types of expressions in a program. For example, we would expect the type of an actual parameter to a function to be a subtype of the formal parameter. Returning to the function `printName` above we could have the constraint:

$$[\![\texttt{p}]\!] \leq [\![\texttt{x\_printName}]\!]$$

Intuitively, this expresses that the type of variable `p` should be subtype of the formal parameter in function `printName`.

2.5.1 Using Type Inference

Type inference has many different uses including static checking, [56, 7, 57, 55], where safety guarantees are given to the programmer before the code is run. In [71] Wang describes a tool that can statically analyse down casts in Java. This could also be used to remove the checking of casts in the bytecode. For example[1]:

```
1 Vector s = new Vector();
2 s.add(``Hello'');
3 String s = (String) s.get(); // Cast will succeed
4 s.add(new Integer(2));
5 s = (String) s.get(); // Cast will fail
```

Java performs dynamic type checking of down casts, this can lead to nasty surprises when using the collection classes. In the above example, at runtime the cast on line 3 will succeed, but the one on line 5 will fail. Using the tool in [71] this type error could be identified before the code is run.

In StarKiller[63], type inference is done for Python with the aim of increasing the performance of compiled code. By inferring the type of an expression it is possible, in many cases, to eliminate dynamic dispatch and binding along with many of the runtime type checks. Similarly, Plevyak and Chien in [61] use type inference to build a optimizing compiler for a Scheme like language called Concurrent Aggregates[25].

In Chapter 5 we will develop a type inference algorithm for the type system developed in Chapter 4.

2.6 JavaScript as a Scripting Language

We start our discussion by looking at what makes a programming language a *scripting language*. The term scripting language came out of languages that were used to connect or glue pre-existing components together to perform a task, often

[1]You can try the tool on the example by going to `http://www.cs.jhu.edu/~wtj/precise/`

for system administration purposes. Often scripting languages are designed with a specific domain in mind such as text processing (Awk) and multimedia (ActionScript). However, most modern scripting languages are sufficiently general purpose to be used for most programming tasks, *e.g.* Python, Perl and PHP. According to *WikiPedia* (`www.wikipedia.com`) scriping language have the following properties:

- Application code is not delivered as object code for the underlying processor but as source code that is interpreted.
- Rapid development is favoured over efficiency of execution
- Variables, functions and methods typically do not require type declarations. There are automated conversions or equivalence between types, particularly between strings and other types.
- Automatic memory management.
- Very powerful built-in types - typically a dictionary, a set, a literal type that can act as either numerics or strings.
- The ability to generate, load and interpret source code at run time through an `eval` function, *e.g.* Perl, Python, Tcl and JavaScript.
- Interface to the underlying operating system, in order to run other programs and communicate with them. (Support for system calls is essential for systems programming languages, but not for scripting languages, which more often need higher-level abstractions that are portable across operating systems)

It is possible to write a script in any language, including Java and C, and so the boundary between scripting languages and regular programming is becoming quite vague. In `http://merd.sourceforge.net/pixel/language-study/scripting-language/`, a study has been undertaken to determine the suitability of a language to be used as a scripting language (Script Oriented Programming). The metrics for assessing suitability include the smallest running program, the "hello world" program and testing for the existence of a file. A low scoring language, such as Java, has the following "hello world" program:

```
public class hello_world {
 public static void main(String[] args) {
   System.out.println("Hello World");
 }
}
```

A high scoring language, such as JavaScript, has the following "hello world" program:

```
System.print("Hello World\n")
```

2.7 Adding Classes and Type Annotations to JavaScript

One of the major themes in this thesis is the development of a static type system for JavaScript. We now discuss an alternative attempt to extend JavaScript with classes and type annotations. While the aim of this extension was not to provide static typing, it does allow better dynamic typing than currently available in JavaScript.

JavaScript uses dynamic typing rather than static typing. The rational for this is that many of the rapid development features found in scripting languages do not fit well with static typing. This allows code to be quickly *hacked* together and deployed to the customer. Horwat, in [40] claims that these languages must evolve to support *"Programming in the Large"* which he defines as:

- Programs written by more than one person,
- Programs assembled from components (packages),
- Programs that live in heterogeneous environments,
- Programs that use or expose evolving interfaces,
- Long-lived programs that evolve over time.

In order to support programming in the large a set of features is suggested for JavaScript 2[40] extending JavaScript to include packages, versioning, optional classes, type annotations and conditional compilation. With this in mind, we now look at how classes and type annotations can be used to enforce static checks. While the other suggestions of packages, versioning and conditional compilation are interesting, they are not directly relevant to type checking.

The ECMA TC39TG1 working group has produced the *ECMAScript 4*[54] standard of which JavaScript 2 is superset. *ECMAScript 4* introduces into JavaScript classes and optional type annotations. Figure 2.3 gives an example that uses some

```
class Shape{
  // declare variables
  private var area:Number;
  private var posx:Number;
  private var posy:Number;
  private var speed:Number;
  private var direction:String;
  // Constructor
  public function Shape(xpos:Number, ypos:Number){
    posx = xpos;
    posy = ypos;
  }
  // declare methods
  public function setArea(parea:Number):Void{
    area = parea;
  }
  public function getArea():Number{
    trace("superclass getarea method");
    return area;
  }
  public function move(speed, dir){
    if(dir=="up") posy -= speed;
    if(dir=="down") posy += speed;
    if(dir=="left") posx -= speed;
    if(dir=="right") posx += speed;
    trace("x = " + posx + ", y = "+posy);
  }
}
```

Figure 2.3.: *ECMAScript 4* example

of the new features. The example uses class `Shape` to describe shapes with an area (field `area`), position (fields `posx` and `posy`) and velocity (fields `speed` and `direction`). There are methods `setArea` and `getArea` that set and get the area respectively and a method `move` to move. There is a constructor `Shape` used when creating objects of class `Shape`.

There are two major implementations of *ECMAScript 4*: *ActionScript 2* and *JScript .NET*[51].

ActionScript 2 is Macromedia's multi-media language. *ActionScript 1* is a variant of JavaScript with multimedia libraries. It is integrated into the Flash development environment which uses type annotations to provide hints to the programmer. For example, if a variable is declared to be of type `String` (one of the predefined datatypes) any use of the variable in the development environment would furnish the programmer with a list of available fields and methods of the `String` type.

```
Account = function(name){
  this.currentOwner = name;
}
getOwner = function(){
  return this.currentOwner;
}
setOwner = function(name){
  this.currentOwner = name;
}
Account.prototype.addProperty("owner", getOwner, setOwner);
//Main code
a = new Account("Chris");
```

Figure 2.4.: *ActionScript 1* Example

```
class Account {
  private var currentOwner:String = "none";

  function Account(name:String){
    currentOwner = name;
  }

  function get owner():String {
    return currentOwner;
  }
  function set owner(name:String):Void {
    currentOwner = name;
  }
}
// Main code
var myChecking:Account = new Account("Chris");
```

Figure 2.5.: *ActionScript 2* Example

Figure 2.4 shows an *ActionScript 1* example with the equivalent *ActionScript 2* code shown in Figure 2.5[2]. Note that in Figure 2.4 functions `getOwner` and `setOwner` are added to Account by making them a property of `Account`. In Figure 2.5 they are represented as methods of the `Account` class.

JScript .NET is Microsoft's evolution of their JScript[51] language. Figure 2.6 gives a JScript example for a fictitious weather service. *JScript .NET*, unlike *JScript*, is a compiled language and integrates closely with the .NET framework [50]. The compiler offers more static type checks than are defined in *ECMAScript 4*; however, most type errors are still detected at runtime. Figure 2.7 gives the equivalent *JScript*

[2]This example comes from a tutorial that can be found at `http://www.kirupa.com/developer/oop2/AS2OOPClassStructure6.htm`

```
function getCondition(strCity) {
  var now = new Date();
  switch (strCity.toUpperCase()) {
    case "LONDON":
      if (now.getMonth() <= 7||now.getMonth() >=9){
        return "overcast"
      }
      if (now.getMonth() == 8) {
        return "partly overcast"
      }
      break;
    case "SEATTLE":
      if (now.getMonth() == 7 && now.getDay()==4){
        return "torrential rain"
      }
      else {
        return "rain"
      }
      break;
    default:
      return "partly cloudy with a chance of showers"
  }
}
```

Figure 2.6.: JScript Example

.NET version of the *JScript* example in Figure 2.6. Note the use of type annotations for function parameters and local variables.

2.8 Discussion

We have introduced type systems and type inference with the relevant notation they entail. We have given a taxonomy of type systems for object oriented programming languages.

We have discussed the proposed extension to JavaScript: *ECMAScript 4*. While this did introduce optional type annotations and classes, it falls short of providing a static type system for JavaScript. This was never the intention of this work. However, it does serve as an interesting insight into how the designers of JavaScript considered adding types to JavaScript.

```
function getConditions(strCity : String) : String
{
  var now : Date = new Date();
  switch (strCity.toUpperCase())
  {
  case "LONDON":
    if (now.getMonth() <= 7 || now.getMonth() >= 9) {
      return "overcast";
    }
    else {
      return "partly overcast and humid";
    }
    break;
  case "SEATTLE":
    if (now.getMonth() == 7 && now.getDay() == 4) {
      return "torrential rain";
    }
    else {
      return "rain";
    }
    break;
  default:
    return "partly cloudy with a chance of showers";
  }
}
```

Figure 2.7.: JScript .NET Example

3 JS$_0$- A Subset of JavaScript

IN THIS CHAPTER we look at the features of JavaScript, and extract a subset called JS$_0$. JS$_0$ will be the language used throughout the formal sections of this thesis. We give some example JS$_0$ programs followed by an operational semantics.

3.1 The Features of JavaScript

Our aim is to provide a language that is a *realistic* subset of JavaScript, but manageable with respect to formalization and static typing. We start by discussing some of the interesting features of JavaScript. We structure our discussion around the following topics: object structure, methods, inheritance, metaprogramming, name resolution, variable declarations and coercions. We also compare how other languages represent these features.

3.1.1 Object Structure

Object structure can be specified in a variety of ways, depending on the language, from classes to object literals in the source code. In class based languages it is the class that dictates the members an object will have. The class acts as a template for the creation of objects. In a nominal type system, such as with Java, the class can be used to represent a type. Furthermore, because Java is statically typed the validity of a field access or method call can be checked statically, for example:

```
class A {
  int m1;
  int m2;

  A foo() { return new A(); }
}

//Main
A a = new A();
a.m1 // OK
a.foo() // OK
a.m3 = 3 // Compile time error
```

In dynamically typed languages with classes, such as Python, although objects can be created through classes they can have members added to them afterwards.

Hence, the example above would execute without an error, because the assignment `a.m3 = 3` would create a new member `m3` in `a`. Access to non-existent members in Python will produce a runtime error (`AttributeError`). In Python, classes are represented as objects. Therefore, members can be added to the class object that will affect all objects created by that class, for example:

```
class A:
  f1 = 10

a1 = A()
a2 = A()
a1.f1 # 10
a2.f3 = 20
A.f2 = 10
a1.f2 # 10
a2.f2 # 10
a2.f3 # 20
a1.f3 # Error as only a2 has f3
```

JavaScript has no classes, instead, functions are used to create objects. These are referred to as constructor functions in the JavaScript[29] standard. This unifies classes and functions into functions. Consider the following example:

```
function A(x) {
  this.f1 = x;
}

a = new A();
a.f1
```

Unlike Python, JavaScript fields are not explicitly defined in constructor functions. Instead, assignments to members of `this` are made, *e.g.* `this.f1 = x`. Note that the same can be done with Python by making assignments to `self` in the constructor (a method named `__init__`) of a class.

Given that the structure of an object can be extended after it has been created, Python classes and JavaScript constructor functions only act as *loose* definitions of object structure. Classes provide a logical structuring of code, but they only loosely define the structure of objects; objects can be changed after being created.

JavaScript allows access to non-existent members without generating an error (see Section 3.1.6). Instead, the `undefined` value is returned. Attempting to access a member on the `undefined` value will result in a runtime error. Similarly, attempting to invoke a method on an undefined value will result in an error.

Furthermore, in JavaScript objects are mappings from strings to values. Access to a member is defined in terms of property references. For example, `e1.m` is actually interpreted at runtime as `e1[m]`, where `m` can be another expression. Thus, `e1.e2` is the more general form of member access represented as `e1[e2]`. The only requirement on `e2` is that it can be *coerced* (see Section 3.1.6) into a string.

3.1.2 Methods

In languages with classes, methods can be defined within the classes. Consider the following Python class:

```
class A:
  f1 = 1
  def m1(self,x):
    self.f1 = x
```

In JavaScript, for an object to acquire a method a function must be assigned to a member of that object, for example:

```
function A(x) {
  this.f1 = x
  this.m1 = m // m will be a method of objects created by A
}

function m(x) {
  this.f1 = x
}

a = new A(10);
a.f1; // 10
a.m1(20);
a.f1; // 20
```

Note that JavaScript does permit objects to gain methods by assigning a function definition to a member. For example, the above code could be represented as:

```
function A(x) {
  this.f1 = x
  this.m1 = function m(x) { this.f1 = x }
}

a = new A(10);
a.f1; // 10
a.m1(20);
a.f1; // 20
```

3.1.3 Inheritance

In Python, inheritance can be achieved by parameterizing the class definition with the base classes, for example:

```
class A:
  f1 = 10

class B(A):
  f2 = 20
```

Here, class `B` will inherit the fields and methods of class `A`. It is possible to have multiple inheritance by specifying a list of base classes: `class B(`$A_1 \ldots A_n$`)`. If a member is not found when looked up in an object, a search of the class occurs followed by a breadth first search of the base classes.

Code inheritance in JavaScript is achieved using delegation which allows code to be shared between objects. When an object is created it maintains a link to the function that created it. Each function has a prototype object associated with it. When a member is looked up in an object if it is not found then the prototype of the function that created it is searched. Thus, members can be shared between instances of objects. However, when attempting to update a shared member in an object, a new member will be created in the object that will *shadow* the shared one. Consider the following example:

```
function Point(x,y) {
  this.x = x;
  this.y = y;
}

function addFn(p) {
  this.x += p.x;
  this.y += p.y
}

Point.prototype.add = addFn; // Shared by all points

a = new Point(1,1);
b = new Point(2,2);
a.add(b);
```

3.1.4 Metaprogramming

Metaprogramming allows programs to write or change themselves or other programs. A program can generate another program and execute it. Consider the following JavaScript example:

```
prog = "while (true) { "++"print \"Hello\" } "
eval(prog)
```

Here, the two strings `"while (true):   "` and `"print \"Hello\" "` are concatenated together to make a make a string `prog` representing a short program that outputs ”`Hello`” forever. The command `eval` evaluates the string as a program.

Another characteristic of metaprogramming is the ability for a program to look into itself. This is often referred to as *reflection*. Consider the following example:

```
var point = new Point(2,3);
var properties = "";
for(var property in point) properties += property + " " + point[property
    ];
```

Execution of this code will display: `'x' : 2 'y' : 3`. Here, we are seeing JavaScript treating objects like arrays of properties that can be inspected and modified. Thus, one can check that a property exists in an object before attempting to access it. Therefore, the common JavaScript idiom of checking the existence of a member by accessing it:

```
if (obj.prop1) {
  // obj has a prop1 ...
  x = obj.prop1
}
```

is really sugar for looking up the member in the object:

```
for (var prop in obj) {
  if (prop == 'prop1') {
    // obj has a prop1
    x = obj.prop1
    break;
  }
}
```

Testing the presence of a member is primarily used in the web browser environment where differences in the Document Object Model (DOM)[32] structure need to be determined without generating an error. In [26] Damiani et al. give a formalism of JavaScript which includes a member test feature. There the aim was to show an equivalence between JavaScript programs running in different web browsers. This work is covered in more detail in Section 4.4.

3.1.5 Variable Declarations

Variable declarations refer to how and when a programmer must declare variables (if at all). With JavaScript and Python you are required to declare a variable before using it. A declaration is either an explicit declaration: `var count`, or the assignment of a value to a new name: `count = 0`. Using a variable without declaration will generate an error. In Perl and PHP, the approach is more relaxed with references to undeclared variables creating the variable on-the-fly. Note that both PHP and Perl offer a *strict* mode that disallows variable use without declaration.

3.1.6 Coercions

Coercions refer to how a language converts values from one type to another. The standard set of coercions are between numerical representations, for example:

```
i = 2 // i is an integer
j = i + 2.4 // i is coerced to a float
```

JavaScript will coerce a variable to an object by *wrapping* it when necessary. This can create some very subtle behaviour; as highlighted in [66] with this example:

```
1  var str = "danger"
2  str.m1 = 23
3  str.m1 // Nothing!
```

In this example, `str` has a runtime type `string` (a base type). When `str.m1` is executed JavaScript wraps `str` in an object and adds the member `m1`. However, the new wrapper object is *not* referred to by `str`, and so it is discarded. The access `str.m1` on line 3 creates *another* wrapper object which has *no* member `m1`[1]. Consider the same example, but this time we have initialized `str`:

```
1  var str = new String("danger")
2  str.m1 = 23
3  str.m1 // 23
```

The addition of member `m1` to `str` on line (2), does not require `str` to be wrapped. This is because `str` is already referring to a `String` object. Therefore, the access to `m1` on line (3) returns `23`.

3.2 Introduction to JS_0

We introduce the language JS_0, a subset of JavaScript. We discuss each of the language features above and how it is represented in JS_0.

[1]Remember that access to non-existent members is not an error in JavaScript.

Object Structure

JS$_0$ allows objects to be created through constructor functions. We don't have object literals as these are only syntactic sugar. Objects can acquire new members through assignment to members. We do not allow the removal of a member. Furthermore, we don't provide a mechanism for testing the presence of a member.

We don't represent the members of an object through a dictionary, or any other data structure. Therefore, the structure of an object can only be changed directly and *not* by manipulating a special member containing the list of members.

Finally, access to non-existent members of an object will produce a runtime error. We shall see more about this when we explain the operational semantics of JS$_0$ in Section 3.5.

Methods

Methods are defined in the JavaScript way by assigning function identifiers to members of objects. This will allow us to add new methods to objects and reassign existing ones. We don't allow assignment of a function definition to a member; this is easily represented by first defining the function and then assigning its identifier. In JS$_0$, when a function is called as a method of an object the receiver is implicitly bound to `this`: there is no need to explicitly have it as a parameter of a method.

Inheritance

We don't provide JavaScript's prototyping and delegation mechanism in JS$_0$. We discuss the possibility of adding delegation in future work on JS$_0$ in Section 7.2. Although prototyping and delegation are certainly useful when implementing larger systems, we believe that currently it is an underused and misunderstood feature of JavaScript.

Coercions

We chose not to have coercions as they are not suited to the level of abstraction of JS$_0$, *i.e.* we only have integers and objects. We also disallow coercion between integers and objects, i.e. wrapping integers into objects. Again, wrapping of base types (in JS$_0$'s case integers) is just syntactic sugar.

Metaprogramming

The style of metaprogramming supported in JavaScript allows code to be represented as strings. This allows them to be composed together in a *potentially* arbitrary order and executed. In [64] Sheard outlines how the lack of internal structure in strings makes metaprogramming difficult, *i.e.* one can easily create syntactically incorrect and type incorrect programs. Therefore, to be type safe the programmer has to create code to check the constructed program. A better approach is to provide a data structure that captures the syntax of the underlying language as in Haskell[41]; when a correctly typed data structure representing a program means a syntactically correct program. Given JavaScript does not provide a *separate* metaprogramming language we choose not to support metaprogramming. Furthermore, metaprogramming with strings would complicate type inference. The inference would have to *look inside* strings that *could* represent programs and analyse them.

Variable Declarations

JS$_0$ has no local variables and one function parameter. When we discuss the implementation, in Chapter 6, we will allow local variable declarations at the beginning of a function and more than one function parameter. Allowing variable declarations anywhere in a function is a convenience rather than a necessity. It is better programming style to declare variables at the beginning of a function or block[48]. We loose no expressivity as we can model multiple parameters and local variables as objects representing a block of variables (parameter block).

Other Features

JS$_0$ does not include the following features: libraries of functions, native calls, global `this` (through a global object) and functions as objects. We omitted these features because they are not central to the paradigm.

3.3 Syntax of JS$_0$

The syntax of JS$_0$ is given in Figure 3.1. A program is a sequence of function declarations. In JS$_0$, functions may have only one formal parameter. For a program P we use P(f) as a shorthand for looking up the definition of function f in P, defined as follows:

$$\mathsf{P}(\mathsf{f}) \;=\; \begin{cases} \texttt{function}\,\mathsf{f}(\mathtt{x})\{\,\mathsf{e}\} & \text{if } \mathsf{P} = \mathsf{F}_1\ \texttt{function}\,\mathsf{f}(\mathtt{x})\{\,\mathsf{e}\}\ \mathsf{F}_2 \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

As we will be presenting a *large step* semantics in Section 3.5, we are not required to have addresses as part of our syntax.

$\mathsf{P} \in$ *Program*	::=	F^*	
$\mathsf{F} \in$ *FuncDecl*	::=	`function` f (x) { e }	
$\mathsf{e} \in$ *Exp*	::=	var	locals
		f	function identifier
		new f(e)	object creation
		e; e	sequence
		e.m(e)	member call
		e.m	member select
		f(e)	global call
		lhs = e	assignment
		$\mathsf{e}_1 ?\, \mathsf{e}_2 : \mathsf{e}_3$	conditional
		`null`	null
		n	integer
var $\in$ *EnvVars*	::=	`this` | x	
lhs $\in$ *LeftHandSide*	::=	x | e.m	

Identifiers

$\mathsf{f} \in$ *FuncID*	::=	$\mathsf{f} \mid \mathsf{f}' \mid \ldots$
$\mathsf{m} \in$ *MemberID*	::=	$\mathsf{m} \mid \mathsf{m}' \mid \ldots$

Figure 3.1.: Syntax of JS_0

3.4 Examples

Figure 3.2 is an example JS_0 program that describes an implementation of the JavaScript `Date` object[2]. We define functions `Date` and `addFn` where we assume the standard addition operations, but omit the definition in JS_0 to ease the presentation. Again, to aid presentation we shall assume that the code after the comment `//Main` is the entry point into the program. By convention we could insist that each JS_0 program has a `main` function that is the entry point. This would compli-

[2]For more information on the `Date` object see [35]. We give a simplified version, and allow the adding of one date to another with `add`.

```
1  function Date(x) {
2   this.mSec = x;
3   this.add = addFn;
4   this
5  }
6
7  function addFn(x) {
8   this.mSec = this.mSec + x.mSec; this
9  }
10 //Main
11 x = new Date(1000);
12 y = new Date(100);
13 x.add(y);
```

Figure 3.2.: Date Example

cate the presentation, and not add any value to the formalism. For completeness, Figure 3.3 gives the `Date` example of Figure 3.2 in JS_0 syntax[3]. We have moved the main code (shown below the comment `//Main` in Figure 3.2) into a function `main`. We use a *parameter block* object, created through function `parameterBlock`, with fields `param1` and `param2` to represent the local variables `x` and `y` respectively. The `Date` example of Figure 3.2 demonstrates the following features:

1. Creating objects using functions (line 11 and 12),
2. Implicit creation of members in objects through assignment (lines 2 and 3) and
3. Acquiring methods through assignment of a function to a member (line 3).

Figure 3.4 presents a HTML page containing some JavaScript code[4]. Function `checkform` is typical of the JavaScript code found in webpages. Web browsers allow code to be *attached* to elements in a page. One can think of a webpage as a tree of elements where each element corresponds to a *tag* in the page. When JavaScript code is placed in a webpage, it can access the elements in the page through the global `document` object. Any code that is attached to a tag has a receiver, `this`, corresponding to the object representing that tag. In Figure 3.4 the element `form` has code associated with its `onSubmit` event. Events occur in response to user interactions within a webpage, *e.g.* clicking on a button.

[3]As in Figure 3.2 we use numbers and the operator '+'. While it is possible to define them using objects it would greatly complicate the presentation.

[4]In fact, the page is derived from the web front of the implementation.

```
function Date(x) {
   this.mSec = x;
   this.add = addFn;
   this;
}

function addFn(x) {
   this.mSec = this.mSec + x.mSec; this;
}

function main(x) {
  x = new parameterBlock(new Date(1000));
  x.param2 = new Date(100);
  x.param1.add(x.param2);
}

function parameterBlock(x) {
  this.param1 = x;this;
}
```

Figure 3.3.: Date Example In JS_0 Syntax

```
<html>
<body>
<script language="javascript" type="text/javascript">
function checkform(theform) {
  theform.submit.disabled=1;
  theform.submit.value = "Calculating..."
}
</script>
</body>
<form method=post onSubmit="return checkform(this);">
<input type=submit name=submit value="Infer">
</form>
</html>
```

Figure 3.4.: Web Form Example HTML

Without concerning ourselves too much with the details of webpages, this example demonstrates the interaction between JavaScript code and the webpage. The form will display a *submit* button with text "Infer":

When the button is clicked; the web browser will invoke the `onSubmit` method of the `form`. As can be seen in Figure 3.4 the code associated with `onSubmit` makes a call to global function `checkform` through `return checkform(this`. Note that the passing of `this` to `checkform` passes a reference to the JavaScript object representing the `form`. Once inside the body of `checkform` (the code for which is the same in Figures 3.4 and 3.5) the `submit` member of the `form` has its `disabled` member set to `1` and `value` member set to `Calculating....` This has the effect on the button of disabling it (by greying out), so that multiple clicks cannot occur, and to set the text to "Calculating":

Another point of interest is how the members of objects representing the elements of a page get their names. In function `checkform` the `submit` member of the `form` is accessed. The name and value of this member come directly from the webpage. Note that the `input` tag is nested inside the `form` tag and is given name `submit` through `name=submit`. This nesting and naming give rise to the object structure that is relied on in the code `checkform`. Hence, if the name were to change, *e.g.* `name=newsubmit`, the code in `checkform` would *fail*. We shall study this issue further in Chapters 4 and 5.

Figure 3.5 represents the webpage in 3.4 as JS_0 code. Note that we have been liberal with the syntax and allowed functions with no parameters or parameters with a name other than `x`. We represent the HTML elements `form` and `input` as JS_0 functions `form` and `input` respectively. The code in the `form` tag in Figure 3.4 is represented as a method of `form` in Figure 3.5 through `this.onSubmit = onSubmit`. The code following `//Main` creates a `form` object and `input` object. The `input` object is added as a new member of the `form` object through

```
function input(value) {
  this.value = value;this.disabled = 0;
  this;
}
function form() {
  this.onSubmit = onSubmit;this;
}

function onSubmit() {
  checkform(this);
}

function checkform(theform) {
  theform.submit.disabled=1;
  theform.submit.value = "Calculating..."
}
//Main
htmlform = new form();
htmlinput = new input("Infer");
htmlform.submit = htmlinput;
htmlform.onSubmit();
```

Figure 3.5.: Web Form Example

`htmlform.submit = htmlinput`. This represents in Figure 3.4 the nesting of the `input` element inside the `form` element.

3.5 Operational Semantics

We have a structural operational semantics for JS_0 that rewrites tuples of expressions, heaps and stacks into tuples of values, heaps and stacks in the context of a program. The signature of the rewriting relation $\twoheadrightarrow$ is:

$$\twoheadrightarrow : \mathit{Program} \rightarrow \mathit{Exp} \times \mathit{Heap} \times \mathit{Stack} \rightarrow (\mathit{Val} \cup \mathit{Dev}) \times \mathit{Heap} \times \mathit{Stack}$$

where:

$$\begin{array}{lcl}
\mathsf{H} \in \mathit{Heap} & = & \mathit{Addr} \rightarrow_{\text{fin}} \mathit{Obj} \\
\chi \in \mathit{Stack} & = & \{\texttt{this}, \texttt{x}\} \rightarrow \mathit{Val} \;\; \text{such that} \;\; \chi(\texttt{this}) \in \mathit{Addr} \\
\mathsf{v} \in \mathit{Val} & = & \{\texttt{null}\} \cup \mathit{FuncID} \cup \mathit{Addr} \cup \mathit{Int} \\
\mathsf{dv} \in \mathit{Dev} & = & \{\texttt{nullPntrExc}, \texttt{stuckErr}\} \\
\mathsf{o} \in \mathit{Obj} & = & \mathit{MemberID} \rightarrow_{\text{fin}} \mathit{Val}
\end{array}$$

The heap maps addresses to objects, where addresses, *Addr*, are $\iota_0, ..\iota_n...$ We use $\rightarrow_{\text{fin}}$ to indicate a finite mapping. As usual, the notation $\mathsf{f}[\mathsf{x} \mapsto \mathsf{y}]$ denotes updating function f to map $\mathtt{x}$ to $\mathtt{y}$. Thus, the meaning of heap update $\mathsf{H}[\iota \mapsto \mathsf{v}]$ and stack update $\chi[\mathtt{x} \mapsto \mathsf{v}]$ is clear. The stack maps `this` to an address and `x` to a value, where values, *Val*, are function identifiers (denoting functions), addresses (denoting objects), `null` , or integers. Finally, objects are finite mappings from member identifiers to values. With $\ll\mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_n : \mathsf{v}_n\gg$ we denote the object mapping m_i to v_i for $i \in 1 \cdots n$.

The rules are given in Figure 3.6. We now discuss the most interesting rules: (var), (val), $(memSel)$, $(paramAss)$, (new), $(memCall)$ and $(funcCall)$.

In (var) the receiver (`this`) or parameter (`x`) are looked up in the stack. The heap and stack are unmodified.

In (val) a function name, f, integer, or null rewrites to itself as they are values. The heap and stack are unmodified.

In $(memSel)$ member m is looked up in the receiver, ι (obtained by evaluation of e), in the heap. We will discuss what happens when m is not found in ι in Section 3.5.1.

In $(paramAss)$ we replace (with the stack update operation) the value of `x` in the stack with the value obtained by execution of e.

In (new) we execute the body of function f (looked up in P) with a stack that maps `this` to a fresh address (pointing to an empty object) and maps `x` to the value obtained by execution of e.

In $(memCall)$ we obtain the function definition by looking up the value of member m in the receiver (obtained by evaluation of e) in P. We execute the body with a stack similar to that for (new) except that `this` points to the receiver. For example, returning to the example of Figure 3.2, executing the main body of code `// Main` in the presence of an empty heap, H_0, and stack, χ_0, mapping `x` and `y` to `null` will result in stack χ_1:

$$\chi_1(\mathtt{x}) = \iota_0,\ \chi_1(\mathtt{y}) = \iota_1$$

and updated heap H_1:

$$\mathsf{H}_1(\iota_0) = \ll\mathtt{mSec} : \mathtt{1100}, \mathtt{add} : \mathtt{addFn}\gg$$
$$\mathsf{H}_1(\iota_1) = \ll\mathtt{mSec} : \mathtt{100}, \mathtt{add} : \mathtt{addFn}\gg$$

Note that member `add` of ι_0 and ι_1 has value `addFn`; this indicates that it is an alias of function `addFn` (which was invoked when `x.add(y)` was executed). For the purpose of this example we have extended the stacks χ_0 and χ_1 to include the variable `y`.

In $(funcCall)$, as with (new), we execute the body of function f (looked up in P) with a stack that maps `x` to the value obtained by execution of e, but maps `this` to `null`. This represents there being no receiver in a global function call. As we shall see in Section 3.5.1 any access to `this` in a global function will result in an error.

3.5.1 Runtime Errors

The operational semantics given in the previous section are only concerned with the *correct* execution of JS$_0$ programs. We now discuss what happens when things go wrong. Our choice of behaviour was dictated by our aim of providing a static type system for JS$_0$. That is, we have classified errors into those that the type system *should* prevent and those it will not. A line is drawn that dictates what is an acceptable runtime error and what is not.

As we discussed in Section 3.2; access to an non-existent member in JavaScript does not result in a runtime error. In JS$_0$ we have made this a runtime error, and it will be the job of the type system to ensure that this does not happen. However, as in Java, there are occasions where attempting to access a member will generate a runtime error, even when the program is well typed. This arises because Java (and JavaScript) have a `null` value. Consider the following example where we assume that objects created by `A` have a member m:

```
x = new A(5);
x = null;
x.m; // Runtime error!
```

While the above is statically type correct, in Java a runtime error is still generated. This is because the type system does *not* track the assignment of `null` to variables. There are languages, [49], that attempt to prevent null pointer exceptions by statically tracking where null is assigned. However, in Java the line was drawn, and it was decided that this was an acceptable runtime error. Of course, they could have opted not to have `null`, but this would have been far too restrictive. Returning to the example, if objects created by `A` do *not* have a member m then Java will disallow the program:

```
x = new A(5);
```

$$\frac{}{\begin{array}{l}\mathtt{this}, \mathsf{H}, \chi \twoheadrightarrow \chi(\mathtt{this}), \mathsf{H}, \chi \\ \mathtt{x}, \mathsf{H}, \chi \twoheadrightarrow \chi(\mathtt{x}), \mathsf{H}, \chi\end{array}}\ (var) \qquad \frac{}{\begin{array}{l}\mathsf{f}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{f}, \mathsf{H}, \chi \\ \mathtt{n}, \mathsf{H}, \chi \twoheadrightarrow \mathtt{n}, \mathsf{H}, \chi \\ \mathtt{null}, \mathsf{H}, \chi \twoheadrightarrow \mathtt{null}, \mathsf{H}, \chi\end{array}}\ (val)$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'\end{array}}{\mathsf{e}_1; \mathsf{e}_2, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (seq) \qquad \frac{\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}', \chi'}{\mathsf{e}.\mathsf{m}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{H}'(\iota)(\mathsf{m}), \mathsf{H}', \chi'}\ (memSel)$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}, \mathsf{H}_2, \chi' \\ \mathsf{H}' = \mathsf{H}_2[\iota \mapsto \mathsf{H}_2(\iota)[\mathsf{m} \mapsto \mathsf{v}]]\end{array}}{\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (memAss) \qquad \frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'' \\ \chi' = \chi[\mathtt{x} \mapsto \mathsf{v}]\end{array}}{\mathtt{x} = \mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (paramAss)$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}'', \chi'' \\ \mathsf{v}' > 0 \\ \mathsf{e}_2, \mathsf{H}'', \chi'' \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'\end{array}}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (condTrue) \qquad \frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow 0, \mathsf{H}'', \chi'' \\ \mathsf{e}_3, \mathsf{H}'', \chi'' \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'\end{array}}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (condFalse)$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}_1, \chi' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x})\{\ \mathsf{e}'\} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto \ll\gg] \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi''\end{array}}{\mathtt{new}\, \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}', \chi'}\ (new)$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}_2, \chi' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x})\ \{\ \mathsf{e}'\} \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (memCall)$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}_1, \chi' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x})\ \{\ \mathsf{e}'\} \\ \mathsf{e}', \mathsf{H}_1, \{\mathtt{this} \mapsto \mathtt{null}, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi''\end{array}}{\mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (funcCall)$$

Figure 3.6.: Operational Semantics of JS_0- correct execution

```
x.m; // Static error.
```

If an attempt was made to run the above program (with no member m in A) then we say that the program *gets stuck*. This error falls within the line of unacceptable runtime errors and we *expect* the type system to prevent such errors.

We must have operational semantics to cover *both* cases. From this point, we shall refer to errors preventable by the type system as *exceptions* and to those that are not preventable as *errors*. When we discuss the static typing of JS_0, in Chapter 4, we shall prove that a well typed program never generates an error, but may generate an exception. Figure 3.7 gives the rules for generation of exceptions and errors in JS_0. The only exception present in our system is null pointer exception, `nullPntrExc`, and the only error is stuck error, `stuckErr`.

Returning to the example above, consider a function A such that:

```
function A(x) {
  this.m = x
}
```

If we have an empty heap, H_0, and stack, χ_0, mapping x to `null`then execution of `x = new A(5)` will result in a stack χ_1 where $\chi_1(\mathtt{x}) = \iota_0$ and updated heap H_1 where $\mathsf{H}_1(\iota_0) = \ll \mathtt{m} : 5 \gg$. Execution of `x = null` will result in a stack χ_2 where $\chi_2(\mathtt{x}) = \mathtt{null}$. Finally, execution of `x.m` will result in rule $(nullExc)$ being applied because $\chi_2(\mathtt{x}) = \mathtt{null}$.

When objects created by A do not contain a member m:

```
function A(x) {
}
```

Execution of `x.m` will result in rule $(noMem)$ being applied because $\chi_2(\mathtt{x})(\mathsf{m}) = \mathcal{U}df$.

If an exception or error does occur while executing an expression then it must be propagated *upwards*. The rules for propagation of exceptions and errors are given in Figure 3.8. For example, if a series of nested calls `e.m(1).n(2).o(3)` where to encounter a problem, say with the call to method m (member no found), then the

$$\frac{\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \texttt{null}, \mathsf{H}', \chi'}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \chi \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \chi' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \chi' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \chi \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \chi'\end{array}} \; (nullExc)$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi' \\ \mathsf{v} \neq \texttt{null} \\ \mathsf{v} \notin \mathit{Addr} \text{ or } (\mathsf{v} = \iota \text{ and } \mathsf{H}(\iota) = \mathcal{U}\mathit{df})\end{array}}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi'\end{array}} \; (noAddr)$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}', \chi' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathcal{U}\mathit{df}\end{array}}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi'\end{array}} \; (noMem)$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}', \chi' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathcal{U}\mathit{df}\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi'} \; (noMeth)$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}', \chi' \\ \mathsf{P}(\mathsf{f}) = \mathcal{U}\mathit{df}\end{array}}{\begin{array}{l}\mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' \\ \texttt{new}\ \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi'\end{array}} \; (noFun)$$

Figure 3.7.: Operational Semantics - Generation of Exceptions

following would happen:

$$\begin{array}{lr} \texttt{e.m(1).n(2).o(3)}, \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' & (prop1) \\ \texttt{e.m(1).n(2)}, \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' & (prop1) \\ \texttt{e.m(1)}, \mathsf{H}, \chi \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \chi' & (noMem) \end{array}$$

We see that the `stuckErr` error is generated by the expression `e.m(1)`, and is propagated through the whole expression via rule $(prop1)$.

3.6 Discussion

We have described JavaScript with its interesting behaviour and features. We have decided on a subset of JavaScript, JS$_0$, which captures the core of the language. We have given an operational semantics that describes the runtime behaviour of JS$_0$ including those cases when things go wrong. In the next chapter we turn our attention to statically typing JS$_0$.

$$\frac{\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi'}{\begin{array}{l} \mathtt{x} = \mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathtt{new}\, \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{e.m}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{e}; \mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \end{array}} \;(prop1)
\qquad
\frac{\begin{array}{l} \mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \end{array}}{\begin{array}{l} \mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \end{array}} \;(prop2)$$

$$\frac{\begin{array}{l} \mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \end{array}}{\mathsf{e}_1; \mathsf{e}_2, \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi'} \;(prop3)
\qquad
\frac{\begin{array}{l} \mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}_2, \chi' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x}) : \mathsf{G}\{\ \mathsf{e}'\} \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi'' \end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi'} \;(prop4)$$

$$\frac{\begin{array}{l} \mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}', \mathsf{H}_1, \chi' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x}) : \mathsf{G}\{\ \mathsf{e}'\} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto \ll\gg] \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi'' \end{array}}{\begin{array}{l} \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \\ \mathtt{new}\, \mathsf{f}(\mathsf{e}), \mathsf{H}, \chi \twoheadrightarrow \mathsf{dv}, \mathsf{H}', \chi' \end{array}} \;(prop5)$$

Figure 3.8.: Operational Semantics - Propagation of errors and exceptions

4 Types for JS$_0$: JS$_0^{\top}$

IN THIS CHAPTER we introduce a typed version of JS$_0$: JS$_0^{\top}$. We introduce the types and how they are added to JS$_0$. We then prove the soundness of the type system.

4.1 Structural Types for JS$_0$

As was described in Section 2.4, structural types with structural subtyping provide a solid foundation for adding types to a scripting language such as JS$_0$. We have developed such a type system for JS$_0$. We start by describing the types and how they work, and then we will discuss how to add the types to JS$_0$.

4.1.1 Syntax of Types

Types, $\mathtt{t}_1$, ..., $\mathtt{t}_n$, are classified into three different kinds: object types, function types and Int (the type of integers). The syntax of types is given in Figure 4.1.

Object Types

Object types are a named collection of types much like a record type [21]. We use O to denote object types. Each component of an object type is identified by a label which we shall refer to as a member name. We use M to denote the list of members (and their respective types) of an object type. Consider the following object type[1]:

$$\mathtt{t}_1 = [\mathtt{mSec} : (\mathsf{Int}, \circ),\ \mathtt{add} : ((\mathtt{t}_2 \times \mathtt{t}_2 \to \mathtt{t}_2), \circ)]$$

Note that we shall use the above definition of $\mathtt{t}_1$ throughout Chapter 4. We will define the type $\mathtt{t}_2$ shortly and discuss the meaning of the annotation $\circ$ in Section 4.1.2.

We allow recursion in object types by specifying a binding to a variable which represents the overall type of the object type. This allows object types to refer to themselves. We represent the bound variable by placing a μ-binding with a type variable, $\alpha, \alpha', \alpha''$..., before the list of members. We use tp to denote types and type variables. Consider the following type:

[1]Here we use '=' to denote syntactic equality.

$$
\begin{array}{lll}
\mathsf{t} \in \mathit{Type} & ::= & \mathsf{O} \mid \mathsf{G} \mid \mathsf{Int} \\
\mathsf{tp} \in \mathit{PreType} & ::= & \alpha \mid \mathsf{t} \\
\mathsf{O} \in \mathit{ObjType} & ::= & \mu\,\alpha.\mathsf{M} \mid \mathsf{M} \\
\mathsf{G} \in \mathit{FuncType} & ::= & \mu\,\alpha.\mathsf{R} \mid \mathsf{R} \\
\mathsf{M} \in \mathit{ObjMembers} & ::= & [(\mathsf{m} : \mathsf{tm})^*] \\
\mathsf{tm} \in \mathit{MemberType} & ::= & (\mathsf{tp}, \psi) \\
\mathsf{R} \in \mathit{FuncRow} & ::= & (\mathsf{O} \times \mathsf{tp} \rightarrow \mathsf{tp}) \\
 & & \\
\psi \in \mathit{Annotation} & ::= & \circ \mid \bullet \\
\alpha \in \mathit{ObjVar} & ::= & \alpha \mid \alpha' \mid \alpha'' \ldots
\end{array}
$$

Figure 4.1.: Syntax of Types

$$\mathsf{t}_2 = \mu\,\alpha.[\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\alpha \times \alpha \rightarrow \alpha), \bullet)]$$

Note that we shall use the above definition of t_2 throughout Chapter 4. We will discuss the meaning of the annotation $\bullet$ in Section 4.1.2.

If the type of member m is an object type, or Int, the member represents a field. If the type of m is a function type (defined next), then m represents a method. If the type of m is a variable, α, then if α is bound in an object type the member is a field, whereas if it is bound in a function type it is a method.

Function Types

In an object-oriented setting one can think of a method as a function that takes its actual parameter(s) *as well as* the receiver as a parameter. We use G to denote function types. Function types comprise three components: $(\mathsf{O} \times \mathsf{t}_1 \rightarrow \mathsf{t}_2)$. There is the receiver, O, which is an object type, the type of the parameter, t_1, and the type of the return value of the function, t_2. We use R to denote this triple of types. We require that the receiver be an object type because the receiver of any method call will always be an object.

As with object types, we allow a function type to refer to itself by placing a μ-binding before the R.

Integer Types

We have one *base* type, namely Int.

4.1.2 Object Type Annotations

Our type system permits objects to evolve in a controlled manner by 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* '$\circ$' or *definite* '$\bullet$'. We use ψ to denote an annotation. Consider the following type:

$$\mathsf{t}_3 = [\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\mathsf{t}_2 \times \mathsf{t}_2 \to \mathsf{t}_2), \circ)]$$

A definite member in an object type represents an object that has that member present with an appropriate type, *e.g.* member `mSec` in t_3. A potential member in an object type represents a member that *may* be added to an object with an appropriate type, *e.g.* member `add` in t_3. We shall see in Section 4.2 how the type system manages the annotations, and in particular how an annotation changes from potential to definite.

4.1.3 Well-formed Types

We now define our notion of well-formed types. Figure 4.2 gives the definition of free type variables for types. We say that a type is *closed* if it does not contain any free type variables. We say that a type is *bound* if it does have a μ-binding and *unbound* otherwise. Naturally, only object and function types can be bound or unbound. A bound type can be turned into an unbound type by substituting occurrences of the bound variable by the overall type. With $\mathsf{t}_1[\alpha/\mathsf{t}_2]$ we denote the substitution of the free occurrences of α in t_1 with t_2. Figure 4.3 gives the definition of substitutions for types.

An object type is well-formed if it is closed and contains unique member definitions that are themselves well-formed. A function type is well-formed if the receiver, parameter and return type are well-formed. Figure 4.4 gives the definition of well-formed types.

For a well-formed object type O we define $\mathsf{O}(\mathsf{m})$ which returns (if it is defined) the type of member m in O. We first define selection from an unbound object type $\mathsf{M} = [\mathsf{m}_1 : (\mathsf{t}_1, \psi_1) \cdots \mathsf{m}_n : (\mathsf{t}_n, \psi_n)]$ as:

$$\mathsf{M}(\mathsf{m}) = \begin{cases} (\mathsf{t}_i, \psi_i) & \text{if } \mathsf{m} = \mathsf{m}_i \text{ for some } i,\ 1 \leq i \leq n \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
\mathcal{FV}(\mu\,\alpha.\mathsf{M}) &= \mathcal{FV}(\mathsf{M})/\{\alpha\}\\
\mathcal{FV}(\mu\,\alpha.\mathsf{R}) &= \mathcal{FV}(\mathsf{R})/\{\alpha\}\\
\mathcal{FV}([\mathsf{m_1} : (\mathsf{tp}_1, \psi_1)...\mathsf{m_n} : (\mathsf{tp_n}, \psi_n)]) &= \mathcal{FV}(\mathsf{tp}_1) \cup \,...\cup\, \mathcal{FV}(\mathsf{tp}_n)\\
\mathcal{FV}(\mathsf{O} \times \mathsf{tp} \to \mathsf{tp}') &= \mathcal{FV}(\mathsf{O}) \,\cup\, \mathcal{FV}(\mathsf{tp}) \,\cup\, \mathcal{FV}(\mathsf{tp}')\\
\mathcal{FV}(\alpha) &= \{\,\alpha\}\\
\mathcal{FV}(\mathsf{Int}) &= \emptyset
\end{aligned}
$$

Figure 4.2.: Free Variables

$$
\begin{aligned}
(\mu\,\alpha.\mathsf{M})[\alpha_1/\mathsf{tp}] &= \mu\,\alpha.\mathsf{M} && (\text{if } \alpha_1 = \alpha)\\
(\mu\,\alpha.\mathsf{R})[\alpha_1/\mathsf{tp}] &= \mu\,\alpha.\mathsf{R} && (\text{if } \alpha_1 = \alpha)\\
(\mu\,\alpha.\mathsf{M})[\alpha_1/\mathsf{tp}] &= \mu\,\alpha.(\mathsf{M}[\alpha_1/\mathsf{tp}]) && (\text{if } \alpha_1 \neq \alpha)\\
(\mu\,\alpha.\mathsf{R})[\alpha_1/\mathsf{tp}] &= \mu\,\alpha.(\mathsf{R}[\alpha_1/\mathsf{tp}]) && (\text{if } \alpha_1 \neq \alpha)\\
[\mathsf{m_1} : (\mathsf{tp}_1, \psi_1)...\mathsf{m_n} : (\mathsf{tp_n}, \psi_n)][\alpha_1/\mathsf{tp}] &= [\mathsf{m_1} : (\mathsf{tp}_1[\alpha_1/\mathsf{tp}], \psi_1)...\mathsf{m_n} : (\mathsf{tp_n}[\alpha_1/\mathsf{tp}], \psi_n)]\\
(\mathsf{O} \times \mathsf{tp}_1 \to \mathsf{tp}_2)[\alpha_1/\mathsf{tp}] &= (\mathsf{O}[\alpha_1/\mathsf{tp}] \times \mathsf{tp}_1[\alpha_1/\mathsf{tp}] \to \mathsf{tp}_2[\alpha_1/\mathsf{tp}])\\
\alpha_1[\alpha_1/\mathsf{tp}] &= \mathsf{tp}\\
\alpha[\alpha_1/\mathsf{tp}] &= \alpha \quad \text{if } \alpha \neq \alpha_1\\
\mathsf{Int}[\alpha_1/\mathsf{tp}] &= \mathsf{Int}
\end{aligned}
$$

Figure 4.3.: Substitutions

$$\frac{}{\vdash \alpha \diamond}\ (wlfVar) \qquad \frac{}{\vdash \mathsf{Int} \diamond}\ (wlfInt) \qquad \frac{\vdash \mathsf{O} \diamond \quad \vdash \mathsf{t}_2 \diamond \quad \vdash \mathsf{t}_3 \diamond}{\vdash (\mathsf{O} \times \mathsf{t}_2 \to \mathsf{t}_3) \diamond}\ (wlfFunc)$$

$$\frac{\begin{array}{l}\mathcal{FV}(\mathsf{M}) = \emptyset \\ \mathsf{M} = [\mathsf{m}_1 : (\mathsf{t}_1, \psi_1)...\mathsf{m}_n : (\mathsf{t}_n, \psi_n)] \\ \vdash \mathsf{t}_i \ \diamond \ \text{for } i \in 1...n \\ \forall\, \mathsf{m} : (\mathsf{M} = \mathsf{M}_1\ \mathsf{m} : (\mathsf{tp}, \psi)\ \mathsf{M}_2 \wedge \\ \qquad\qquad \mathsf{M} = \mathsf{M}'_1\ \mathsf{m} : (\mathsf{tp}', \psi')\ \mathsf{M}'_2) \implies \mathsf{M}_1 = \mathsf{M}_2 \wedge \mathsf{M}'_1 = \mathsf{M}'_2\end{array}}{\vdash \mathsf{M} \diamond}\ (wlfObj)$$

Figure 4.4.: Well-formed Types

For a bound object type, $\mathsf{O} = \mu\ \alpha.\mathsf{M}$, we have:

$$\mathsf{O}(\mathsf{m}) = \mathsf{M}[\alpha/\mathsf{O}](\mathsf{m})$$

That is, the type is closed by substituting occurrences of α with the enclosing type. Therefore, if O is well-formed then $\mathsf{O}(\mathsf{m})$ is well-formed.

We use the notation: $\mathsf{G}(\mathtt{this})$, $\mathsf{G}(\mathtt{x})$ and $\mathsf{G}(\mathtt{ret})$ to denote the type of the receiver, parameter and return value respectively of G. We define selection from an unbound function $\mathsf{R} = (\mathsf{O} \times \mathsf{t}_1 \to \mathsf{t}_2)$ type as:

$$\mathsf{R}(\mathtt{this}) = \mathsf{O} \qquad \mathsf{R}(\mathtt{x}) = \mathsf{t}_1 \qquad \mathsf{R}(\mathtt{ret}) = \mathsf{t}_2$$

For bound function types, $\mathsf{G} = \mu\ \alpha.\mathsf{R}$, we have: $\mathsf{G}(\mathtt{z}) = (\mathsf{R}[\alpha/\mathsf{G}])(\mathtt{z})$ where $\mathtt{z} \in \{\mathtt{x}, \mathtt{this}, \mathtt{ret}\}$.

With $\mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \psi)]$ we denote the *updating of the member* m *to type* t *with annotation* ψ *in* O. If O and t are well-formed then $\mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \psi)]$ is also well-formed.

4.1.4 Congruence and Subtyping

Congruence between types is defined in Figure 4.5. Object types are congruent up to α-conversion, permutation of their members and unfolding of the bound variable. Function types are congruent up to α-conversion and unfolding of the bound variable. The most interesting rule is $(\equiv fix)$ which asserts that if two types have the same *fixed point* then they are congruent. That is, types t and t$'$ are congruent to

$$\frac{}{\mathsf{tp} \equiv \mathsf{tp}}\ (\equiv\ \mathit{reflex}) \qquad \frac{\mathsf{t}_1 \equiv \mathsf{t}_2 \quad \mathsf{t}_2 \equiv \mathsf{t}_3}{\mathsf{t}_1 \equiv \mathsf{t}_3}\ (\equiv\ \mathit{trans}) \qquad \frac{\alpha \notin \mathcal{FV}(\mathsf{M})}{\mathsf{M} \equiv \mu\,\alpha.\mathsf{M}}\ (\equiv\ \mathit{intro})$$

$$\frac{}{\begin{array}{l}\mu\,\alpha.\mathsf{M} \equiv \mathsf{M}[\alpha/\mu\,\alpha.\mathsf{M}]\\ \mu\,\alpha.\mathsf{R} \equiv \mathsf{R}[\alpha/\mu\,\alpha.\mathsf{R}]\end{array}}\ (\equiv\ \mathit{unfold}) \qquad \frac{}{\begin{array}{l}\mu\,\alpha.\mathsf{M} \equiv \mu\,\alpha'.\mathsf{M}[\alpha/\alpha']\\ \mu\,\alpha.\mathsf{R} \equiv \mu\,\alpha'.\mathsf{R}[\alpha/\alpha']\end{array}}\ (\equiv\ \alpha\mathit{Conv})$$

$$\frac{\forall\,\mathsf{m} : \mathsf{M}(\mathsf{m}) \equiv \mathsf{M}'(\mathsf{m})}{\mu\,\alpha.\mathsf{M} \equiv \mu\,\alpha.\mathsf{M}'}\ (\equiv\ \mathit{reorder}) \qquad \frac{\mathsf{t} \equiv \mathsf{t}'}{(\mathsf{t},\psi) \equiv (\mathsf{t}',\psi)}\ (\equiv\ \mathit{member})$$

$$\frac{\mathsf{M} \equiv \mathsf{M}' \quad \mathsf{t}_1 \equiv \mathsf{t}_1' \quad \mathsf{t}_2 \equiv \mathsf{t}_2'}{\mu\,\alpha.(\mathsf{M} \times \mathsf{t}_1 \rightarrow \mathsf{t}_2) \equiv \mu\,\alpha.(\mathsf{M}' \times \mathsf{t}_1' \rightarrow \mathsf{t}_2')}\ (\equiv\ \mathit{func})$$

$$\frac{\mathsf{t}''[\alpha/\mathsf{t}] \equiv \mathsf{t} \quad \mathsf{t}''[\alpha/\mathsf{t}'] \equiv \mathsf{t}'}{\mathsf{t} \equiv \mathsf{t}'}\ (\equiv\ \mathit{fix})$$

Figure 4.5.: Congruence for Types

the type operator t$''$ with α substituted by t and t$'$ respectively. For a detailed discussion of congruence for recursive types see [8]. Consider the following example:

$$\begin{array}{lcl}
\mathsf{t} & = & \mu\,\alpha.[\mathsf{m} : ([\mathsf{m} : (\alpha,\bullet)],\bullet)]\\
\mathsf{t}' & = & \mu\,\alpha.[\mathsf{m} : (\alpha,\bullet)]\\
\mathsf{t}'' & = & [\mathsf{m} : ([\mathsf{m} : (\alpha,\bullet)],\bullet)]
\end{array}$$

$$\begin{array}{lcl}
\mathsf{t} & \equiv & [\mathsf{m} : ([\mathsf{m} : (\mathsf{t},\bullet)],\bullet)]\ \ (\equiv\ \mathit{unfold})\\
\mathsf{t}' & \equiv & [\mathsf{m} : (\mathsf{t}',\bullet)]\ \ (\equiv\ \mathit{unfold})\\
 & \equiv & [\mathsf{m} : ([\mathsf{m} : (\mathsf{t}',\bullet)],\bullet)]\ \ (\equiv\ \mathit{unfold})
\end{array}$$

$$\begin{array}{lcl}
\mathsf{t}''[\alpha/\mathsf{t}] & \equiv & \mathsf{t}\\
\mathsf{t}''[\alpha/\mathsf{t}'] & \equiv & \mathsf{t}'
\end{array}$$

The *subtyping* judgement t $\leq$ t$'$, defined in Figure 4.6, means that an object or function of type t can be used whenever one of type t$'$ is required. For object types (see rule ($\leq$ *obj*)) we have subtyping in width. If O $\leq$ O$'$ then all definite members of O$'$ must be present and congruent with those in O, and all potential members

$$\frac{\psi' = \bullet \quad \Longrightarrow \quad \psi = \bullet}{\psi \leq \psi'} \; (\leq \mathit{ano})$$

$$\frac{\mathsf{t} \equiv \mathsf{t}' \quad \psi \leq \psi'}{(\mathsf{t}, \psi) \leq (\mathsf{t}', \psi')} \; (\leq \mathit{mem}) \qquad \frac{\mathsf{t} \equiv \mathsf{t}'}{\mathsf{t} \leq \mathsf{t}'} \; (\leq \mathit{cong}) \qquad \frac{\mathsf{t} \leq \mathsf{t}' \quad \mathsf{t}' \leq \mathsf{t}''}{\mathsf{t} \leq \mathsf{t}''} \; (\leq \mathit{trans})$$

$$\frac{\forall\, \mathsf{m} \,:\, \mathsf{O}'(\mathsf{m}) = (\mathsf{t}', \psi') \quad \Longrightarrow \quad (\mathsf{O}(\mathsf{m}) = (\mathsf{t}, \psi) \;\wedge\; (\mathsf{t}, \psi) \leq (\mathsf{t}', \psi'))}{\mathsf{O} \leq \mathsf{O}'} \; (\leq \mathit{obj})$$

Figure 4.6.: Subtyping

```
x:$\mu\ \alpha.[\mathsf{m1} : (\mathsf{Int}, \circ)]$
y:$\mu\ \alpha.[\mathsf{m1} : (\alpha, \circ)]$
z:$\mu\ \alpha.[]$
z = x;
x.m1 = 5;
y = z;
y.m1 = null
```

Figure 4.7.: Dangers of Ignoring Potential Members Example

of O' must be present as potential or definite members of O with congruent types. This condition is needed to insure that the addition of a new member to an object does not break compatibility.

For example, consider the code fragment in Figure 4.7 (a similar example could be given using the syntax of JS$_0^{\mathsf{T}}$). Our subtyping for object types is such that $\mu\,\alpha.[\mathsf{m1} : (\mathsf{Int}, \circ)]$ and $\mu\,\alpha.[\mathsf{m1} : (\alpha, \circ)]$ are subtypes of $\mu\,\alpha.[]$. Hence, the assignment `z = x` is correct.

Subtyping does not disregard potential members; therefore, $\mu\,\alpha.[]$ is not a subtype of $\mu\,\alpha.[\mathsf{m1} : (\mathsf{Int}, \circ)]$, and the assignment `y = z` is type incorrect. However, if `y = z` were allowed, then at the end of the code fragment the field `m1` of `x` would contain `null` instead of an integer.

For function types and integers subtyping coincides with congruence (see rule ($\leq$ *cong*)). All annotations are a subtype of $\circ$ (see rule ($\leq$ *mem*)). For members of

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Int} \equiv \mathsf{Int}}\,(\equiv relfex)}{(\mathsf{Int},\bullet) \leq (\mathsf{Int},\circ)}\,(\leq mem) \qquad \dfrac{\begin{array}{c}\alpha[\alpha/\mathsf{t}_2] \equiv \mathsf{t}_2\\ \alpha[\alpha/\mathsf{t}_2] \equiv \mathsf{t}_2\\ \alpha[\alpha/\mathsf{t}_2] \equiv \mathsf{t}_2\end{array}}{((\alpha \times \alpha \to \alpha)[\alpha/\mathsf{t}_2],\bullet) \leq ((\mathsf{t}_2 \times \mathsf{t}_2 \to \mathsf{t}_2),\circ)}\,(\leq func)}{\mathsf{t}_2 \leq \mathsf{t}_1}\,(\leq obj)$$

Figure 4.8.: Derivation of $\mathsf{t}_2 \leq \mathsf{t}_1$

an object type: the types must be congruent and the annotations must be subtypes (see rule $(\leq mem)$).

Returning to the examples of Section 4.1.1 we see that t_2 is a subtype of t_1, because all members of t_1 are also members of t_2 and have congruent types; furthermore, all members of t_2 are definite. We show the derivation in Figure 4.8.

4.2 Adding types to JS_0: JS_0^T

We present JS_0^T an extension of JS_0, where functions are annotated with a function type G. Figure 4.10 shows the parts of JS_0^T that differ from JS_0.

By giving a function type to each function definition we give *complete* type information for that function, *i.e.* the type of the receiver, parameter and return value. We shall see that a single annotation on each function definition is sufficient to type check a program.

Figure 4.9 gives a JS_0^T version of the `Date` example from Figure 3.2. Remember from earlier that $\mathsf{t}_1 = [\texttt{mSec} : (\mathsf{Int},\circ),\ \texttt{add} : ((\mathsf{t}_2 \times \mathsf{t}_2 \to \mathsf{t}_2),\circ)]$ and $\mathsf{t}_2 = \mu\,\alpha.[\texttt{mSec} : (\mathsf{Int},\bullet),\ \texttt{add} : ((\alpha \times \alpha \to \alpha),\bullet)]$.

4.2.1 Type Checking Expressions

Typing expression e in the context of program P and environment Γ has the form:

$$\mathbb{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \,||\, \Gamma'$$

The environment, $\Gamma = \{\texttt{this} : \mathsf{O}, \texttt{x} : \mathsf{t}\}$, maps the receiver, `this`, to a well-formed object type and the formal parameter, `x`, to a well-formed type. The environment on the right hand side of the judgement, Γ', reflects the changes to the type of the receiver or parameter while typing the expression. As we shall see in the type rules;

```
function Date(x):(t₁ × Int → t₂) {
 this.mSec = x;
 this.add = addFn;
 this
}

function addFn(x):(t₂ × t₂ → t₂) {
 this.mSec = this.mSec + x.mSec; this;
}
//Main
t₂ x = new Date(1000);
t₂ y = new Date(100);
x.add(y);
```

Figure 4.9.: Typed JS_0 Date Example.

$$\begin{array}{lll} \mathbb{P} \in \mathit{Program} & ::= & \mathbb{F}^* \\ \mathbb{F} \in \mathit{FuncDecl} & ::= & \texttt{function}\ \mathsf{f}(\mathtt{x}) : \mathsf{G}\ \{\ \mathtt{e}\} \end{array}$$

Figure 4.10.: Syntax of JS_0^T

the only change possible is a change from potential to definite of the annotation of a member. With $\Gamma[\mathsf{var} \mapsto \mathsf{t}]$ we denote the *updating of* var *to type* t *in* Γ. We define environment lookup, Γ (var), in the standard way:

$$\Gamma(\mathsf{var}) \quad = \quad \begin{cases} \mathsf{O} & \text{if } \mathsf{var} = \mathtt{this} \\ \mathsf{t} & \text{if } \mathsf{var} = \mathtt{x} \\ \mathit{Udf} & \text{otherwise} \end{cases}$$
$$\text{where } \Gamma = \{\mathtt{this} : \mathsf{O}, \mathtt{x} : \mathsf{t}\}$$

Definition 1 *An environment,* Γ*, is well-formed iff* $\vdash \Gamma(\mathtt{this}) \diamond$ *and* $\vdash \Gamma(\mathtt{x}) \diamond$.

The typing rules are given in Figure 4.11. Rules (var), $(func)$, $(const)$ and (seq) are straightforward. Note that `null` may have any object type. We now discuss the interesting rules in Figure 4.11.

In rule $(memAcc)$ the expression e must be of an object type in which the member m is definite, *i.e.* with annotation •.

In rule $(varAss)$ the type of e has to be a subtype of the type of x in Γ'.

In rule $(methCall)$ we check that the type of the receiver is an object type with definite member m. Member m must have a function type G. The type of the re-

$$\frac{}{\begin{array}{l}\mathbb{P},\Gamma \vdash \texttt{this} : \Gamma(\texttt{this}) \parallel \Gamma \\ \mathbb{P},\Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \parallel \Gamma\end{array}}\ (var) \qquad \frac{\mathbb{P}(\mathsf{f}) = \texttt{function}\,\mathsf{f}(\texttt{x}) : \mathsf{G}...}{\mathbb{P},\Gamma \vdash \mathsf{f} : \mathsf{G} \parallel \Gamma}\ (func)$$

$$\frac{}{\begin{array}{l}\mathbb{P},\Gamma \vdash \texttt{null} \ : \mathsf{O} \parallel \Gamma \\ \mathbb{P},\Gamma \vdash \texttt{n} : \mathsf{Int} \parallel \Gamma\end{array}}\ (const) \qquad \frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{t} \parallel \Gamma' \\ \mathbb{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma''\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{e}_1\texttt{;}\,\mathsf{e}_2 : \mathsf{t}' \parallel \Gamma''}\ (seq)$$

$$\frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e} : \mathsf{O} \parallel \Gamma' \\ \mathsf{O}(\mathsf{m}) = (\mathsf{t}', \bullet)\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{e.m} : \mathsf{t}' \parallel \Gamma'}\ (memAcc) \qquad \frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \\ \mathsf{t} \le \Gamma'(\texttt{x})\end{array}}{\mathbb{P},\Gamma \vdash \texttt{x} = \mathsf{e} : \mathsf{t} \parallel \Gamma'}\ (varAss)$$

$$\frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{O} \parallel \Gamma' \\ \mathsf{O}(\mathsf{m}) = (\mathsf{G}, \bullet) \\ \mathbb{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma'' \\ \mathsf{t}' \le \mathsf{G}(\texttt{x}) \\ \mathsf{O} \le \mathsf{G}(\texttt{this})\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2) : \mathsf{G}(\texttt{ret}) \parallel \Gamma''}\ (methCall) \qquad \frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \\ \mathbb{P}(\mathsf{f}) = \texttt{function}\,\mathsf{f}(\texttt{x}) : \mathsf{G}\ \{...\} \\ \mathsf{t} \le \mathsf{G}(\texttt{x}) \\ \{\mathsf{t}' \mid (\mathsf{G}(\texttt{this}))(\mathsf{m}) = (\mathsf{t}', \bullet)\} = \emptyset\end{array}}{\begin{array}{l}\mathbb{P},\Gamma \vdash \texttt{new}\,\mathsf{f}(\mathsf{e}) : \mathsf{G}(\texttt{ret}) \parallel \Gamma' \\ \mathbb{P},\Gamma \vdash \mathsf{f}(\mathsf{e}) : \mathsf{G}(\texttt{ret}) \parallel \Gamma'\end{array}}\ (call)$$

$$\frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma' \\ \mathsf{var} = \mathsf{e}' \text{ is not a subexpression of } \mathsf{e}_2 \\ \Gamma'(\mathsf{var}) \ = \mathsf{O} \\ \mathsf{O}(\mathsf{m}) = (\mathsf{t}'', \psi) \\ \mathsf{t} \le \mathsf{t}'' \\ \Gamma'' = \Gamma'[\mathsf{var} \mapsto \mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}'', \bullet)]]\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{var.m} = \mathsf{e}_2 : \mathsf{t} \parallel \Gamma''}\ (assignAdd)$$

$$\frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{O} \parallel \Gamma' \\ \mathbb{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma'' \\ \mathsf{O}(\mathsf{m}) = (\mathsf{t}'', \bullet) \\ \mathsf{t} \le \mathsf{t}''\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{e}_1.\mathsf{m} = \mathsf{e}_2 : \mathsf{t} \parallel \Gamma''}\ (assignUpd) \qquad \frac{\begin{array}{l}\mathbb{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{Int} \parallel \Gamma' \\ \mathbb{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma'' \\ \mathbb{P},\Gamma' \vdash \mathsf{e}_3 : \mathsf{t}' \parallel \Gamma'''\end{array}}{\mathbb{P},\Gamma \vdash \mathsf{e}_1?\,\mathsf{e}_2 :\ \mathsf{e}_3 : \mathsf{t} \sqcup \mathsf{t}' \parallel \Gamma'' \sqcup \Gamma'''}\ (cond)$$

Figure 4.11.: Type Rules for Expressions in JS_0^T

$$\frac{\begin{array}{l}\forall\, \mathsf{f}:\ \ \mathbb{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f(x)} : \mathsf{G}\,\{\mathsf{e}\} \wedge\ \vdash \mathsf{G} \diamond \\ \qquad\qquad \Longrightarrow\ \ \mathbb{P}, \{\, \mathtt{this} : \mathsf{G}(\mathtt{this}), \mathsf{x} : \mathsf{G(x)}\, \} \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'' \ \wedge\ \mathsf{t} \leq \mathsf{G}(\mathtt{ret})\end{array}}{\vdash \mathbb{P} \diamond}\ (prog)$$

Figure 4.12.: Type Rules for Programs in JS$_0^\mathsf{T}$

ceiver and actual parameter must be a subtype of the declared type of the receiver and formal parameter in G.

In rule $(call)$ we consider global calls and constructors, and require that the type of the receiver defined in the function has no definite members. This is consistent with the operational semantics, where in the case of global call and object creation we start with an empty receiver object.

Rule $(assignAdd)$ describes assignment to a member (preceded, potentially, by addition of that member). The final environment Γ'' ensures that member m (of this or of the formal parameter) is definite. From this point onwards member m may be accessed. For example, consider the expression:

$$\mathsf{x.m_2} = \mathsf{x}$$

with environment Γ, where:

$$\Gamma(\mathsf{x}) = \mathsf{t} = \mu\, \alpha.[\mathsf{m_1} : (\mathsf{Int}, \bullet),\ \mathsf{m_2} : (\alpha, \circ)]$$

The expression is well-typed in Γ and application of $(assignAdd)$ gives:

$$\mathbb{P}, \Gamma \vdash \mathsf{x.m_2} = \mathsf{x} : \mathsf{t} \parallel \Gamma'$$

with environment Γ', where:

$$\Gamma'(\mathsf{x}) = \mathsf{t'} = [\mathsf{m_1} : (\mathsf{Int}, \bullet), \mathsf{m_2} : (\mathsf{t}, \bullet)]$$

This reflects the updating of member $\mathsf{m_2}$. Note that although member $\mathsf{m_2}$ of $\mathsf{t'}$ is definite, member $\mathsf{m_2}$ of $\mathsf{x.m_2}$ will *not* be definite. Therefore, when updating the member's annotation the type of that member is closed. In the type rule the selection of m from O gives the closed type of m (see Section 4.1.3). Recall that member $\mathsf{m_2}$ of t is α, but t (m) = t.

If the update were to only update the annotation, then the above update would be: $m_2 \mapsto (\alpha, \bullet)$. This would allow us to access `x.m2.m2`! This is clearly wrong as we only updated `x` and not `x.m2`.

The requirement that $\mathsf{var} = \mathsf{e}'$ is not a subexpression of e_2 ensures that the receiver of the update is not changed by the expression on the right hand side. If this requirement were not in place, then the type system could go out of sync with the actual object referenced by var. Consider the following example where x has a type $[\mathsf{m} : (\mathsf{t}, \circ)]$, for some t, and `A` has a return type with a potential member m:

```
x.m = (x = new A());
```

Operationally the *original* receiver of the assignment (the x from x.m) will have a member m added. But the right hand side, `x = new A()`, changes x to reference a new object. Assume that `A` returns an empty object (note that this still agrees with a return type where m is potential). After the assignment `x = new A()` the parameter x will not have a member m; only the original object referenced by x will. Thus, if the type system allowed the assignment `x = new A()` on the right hand side the type of x would have definite member m, but the real object would not! This would clearly be unsound.

We see in rule $(assignAdd)$ that the environment is changed to reflect the change in type of only `this` or x. Hence, we only track the change in the type of `this` and x in the scope of a function. To illustrate, consider the example in Figure 4.13. This program cannot be typed in JS_0^T, because member `m` of `z` would have to be definite (see line 11). Function `g` is used as a constructor (see line 9) and the result is assigned to `z`. This would require `this` in function `g` to have definite member `m`. Recall that (*call*) requires the receiver to have no definite members. Thus, we cannot type this program.

However, the type system *does* allow an object to be extended *after* it has been constructed. Consider the example in Figure 4.14. The code is identical to that of Figure 4.13 with line 11 replaced by an assignment (thus creating the new member) to member `m`. We can give `z` the type $[\mathsf{p} : (\mathsf{Int}, \bullet), \mathsf{m} : (\mathsf{Int}, \circ)]$ which must be a subtype of the return type for function `g`. Note that the parameter type for function f must be an object type with member m of type Int. This is because of the assignment on line 2 to m coupled with the passing of `z` to `f`.

```
function f(x) {
  x.m = 2;x;
}

function g(x) {
  this.p = x;this;
}

z = new g(2);
z = f(z);
z.m // Problem!
```

Figure 4.13.: JS$_0^T$ 'forgets' assignment in function f

```
function f(x) {
  x.m = 2;x;
}

function g(x) {
  this.p = x;this;
}

z = new g(2);
z = f(z);
z.m = 4 // Ok!
z.m // Ok!
```

Figure 4.14.: JS$_0^T$ Adding New Member After Object Creation

Rule $(assignUpd)$ is applicable when the receiver of the assignment (the expression before the field on the left hand side of the assignment) is not `this` or `x`. The receiver must have a definite member that is a subtype of the right hand side of the assignment.

In rule $(cond)$ the least upper bound, $\sqcup$, of the true branch, t, and false branch, t' is taken; $\mathsf{t} \sqcup \mathsf{t}'$ will only contain members common to both t and t'. The annotation of members in $\mathsf{t} \sqcup \mathsf{t}'$ is found using $\sqcup$ for annotations. A member is definite only if it is definite in both t and t', otherwise it it is potential. We lift least upper bound onto environments in the obvious way. The definition of $\sqcup$ for types, annotations and environments is shown in Figure 4.15.

We say that O' extends O, denoted by $\mathsf{O}' \sqsubseteq \mathsf{O}$, see Definition 2, if both O and O' contain the same members with congruent types, and the annotation of a member m in O' is a subtype of the annotation of member m in O.

Definition 2 $\mathsf{O}' \sqsubseteq \mathsf{O}$ *iff for all* m*:*

- $\mathsf{O}(\mathsf{m}) = \mathcal{U}df \iff \mathsf{O}'(\mathsf{m}) = \mathcal{U}df$ *and*
- $\mathsf{O}(\mathsf{m}) \neq \mathcal{U}df \implies \mathsf{O}'(\mathsf{m}) \leq \mathsf{O}(\mathsf{m})$

Consider the following examples:

$$[\mathsf{m}_1 : (\mathsf{t}_1, \bullet), \mathsf{m}_2 : (\mathsf{t}_2, \circ)] \sqsubseteq [\mathsf{m}_1 : (\mathsf{t}_1, \circ), \mathsf{m}_2 : (\mathsf{t}_2, \circ)]$$
$$\mu\, \alpha.[\mathsf{m}_1 : (\alpha, \bullet)] \sqsubseteq \mu\, \alpha.[\mathsf{m}_1 : (\alpha, \circ)]$$
$$[\mathsf{m}_1 : (\mathsf{t}_1, \bullet), \mathsf{m}_2 : (\mathsf{t}_2, \circ)] \not\sqsubseteq [\mathsf{m}_1 : (\mathsf{t}_1, \circ)]$$

Proposition 1 follows directly from Definition 2.

Proposition 1 *For types* O*,* O'*:*

1. $\mathsf{O} \sqsubseteq \mathsf{O}' \implies \mathsf{O} \leq \mathsf{O}'$

Proposition 2 states that during type checking of an expression the variables in the environment, `this` and `x`, are extended but do *not* gain or loose members.

Proposition 2 *For program,* $\mathbb{P}$*, environments* Γ, Γ'*, expression* e*, type* t*, if* $\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \,||\, \Gamma'$ *then* $\Gamma'(\mathtt{this}) \sqsubseteq \Gamma(\mathtt{this})$ *and* $\Gamma'(\mathtt{x}) \sqsubseteq \Gamma(\mathtt{x})$*.*

Proof: Follows easily from the definition of the type system. From rule $(assignAdd)$ we see that either `this` or `x` is updated using environment update; the only update

$$
\mathsf{t} \sqcup \mathsf{t}' = \begin{cases} \mathsf{t} & \text{if } \mathsf{t}' \equiv \mathsf{t} \\ \mathsf{O} & \text{if } \mathsf{t} = \mathsf{O}' \text{ and } \mathsf{t}' = \mathsf{O}'' \text{ where} \\ & \quad \mathsf{O}(\mathsf{m}) = (\mathsf{t}_1, \psi_1) \implies \\ & \quad (\mathsf{O}'(\mathsf{m}) = (\mathsf{t}_2, \psi_2) \wedge \\ & \quad \mathsf{O}''(\mathsf{m}) = (\mathsf{t}_3, \psi_3) \wedge \\ & \quad \mathsf{t}_1 \equiv \mathsf{t}_2 \equiv \mathsf{t}_3 \ \wedge \ \psi_1 = \psi_2 \sqcup \psi_3) \end{cases}
$$

$$
\psi \sqcup \psi' = \begin{cases} \bullet & \text{if } \psi = \bullet \text{ and } \psi' = \bullet \\ \circ & \text{otherwise} \end{cases}
$$

$$
\Gamma \sqcup \Gamma' = \{\mathtt{this} : \Gamma(\mathtt{this}) \sqcup \Gamma'(\mathtt{this}), \mathtt{x} : \Gamma(\mathtt{x}) \sqcup \Gamma'(\mathtt{x})\}
$$

Figure 4.15.: Upper Bound for Types, Annotations and Environments

is to a member where we make the annotation definite and close the type. For rule (*cond*) the environment least upper bound operation is used. This with the definition of least upper bound for types ensures that the types in the environment are only extended when both sides of the conditional branch agree. □

Clearly, if there is no relation between t and t' rule (*cond*) is not applicable, because $\sqcup$ will not be defined. For example, $\mathsf{Int} \sqcup [\mathsf{m}_1 : (\mathsf{t}_1, \bullet)]$ is not defined.

A program $\mathbb{P}$ is *well-formed*, see Figure 4.12, if all the function declarations in $\mathbb{P}$ are well-typed.

4.3 Formal Properties of the Type System

We now show that our type system is sound with respect to the operational semantics presented in Section 3.5. We do this in the usual way by showing that execution of an expression preserves the type. The overall idea was explained with relevant references in Section 2.2.2.

4.3.1 Agreement and Store Typings

Because we use a large step semantics the outcome of the rewrite rule is always a value, a heap and a stack. Recall the rewrite relation:

$$
\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'
$$

Hence, we are required to establish a relation between values, heaps, stacks and types. We call this relation *agreement*. In order to establish agreement we will need to give a type to each address in the store. We introduce a store typing [3] which is a function from addresses to types. We shall use this to ensure that a given address in a heap *agrees* with its address in the store typing.

We have two notions of agreement: *weak* and *strong*. Both are shown in Figure 4.16. For values that are not addresses weak and strong agreement are the same. Namely, we require that the value belongs to the set of values it is supposed to agree with *c.f.* rules $(weakInt)$, $(weakNull)$, $(weakFunc)$ and $(strongVal)$. For example, $\mathbb{P}, \mathcal{T} \vdash \texttt{5} \prec \mathsf{Int}$ holds because 5 is an integer. Note that both weak and strong agreement for non-addresses do not depend on the heap or the store typing.

For an address to be in weak agreement with a type it must be a subtype of the type given by the store typing *c.f.* rule $(weakAddr)$. For an address to be in strong agreement with a type it must be in weak agreement and all its members (found by looking up the address in the heap) must be in weak agreement with the type given to the member by the store typing *c.f.* rules $(weakAddr)$ and $(strongAddr)$.

We say that a heap and stack are well-formed with respect to a store typing and environment if:

- The heap and store typing are defined for the same addresses.
- Each address in the store is in strong agreement with its type in the store typing.
- Variables `this` and x agree with their types in the environment.

Proposition 3 states some properties that easily follow from the definitions in Figure 4.16.

Proposition 3 *For program* $\mathbb{P}$*, store typing* $\mathcal{T}$*, heap* H*, value* v*, types* t, t'*:*

1. $\mathbb{P}, \mathcal{T} \vdash \mathsf{v} \prec \mathsf{t},\ \ \mathsf{t} \leq \mathsf{t}' \implies \mathbb{P}, \mathcal{T} \vdash \mathsf{v} \prec \mathsf{t}'$
2. $\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t},\ \ \mathsf{t} \leq \mathsf{t}' \implies \mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t}'$
3. $\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \iota \lhd \mathsf{t} \implies \mathcal{T}(\iota) \leq \mathsf{t}$
4. $\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t},\ \ \mathsf{v} \notin Addr \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t}\ (for\ any\ \mathsf{H}'\ and\ \mathcal{T}')$

In order to reflect the changes to the types of addresses during the evaluation of a well-typed expression we give Definition 3 - *Store Type Extension*.

$$\frac{}{\mathbb{P}, \mathcal{T} \vdash \mathtt{n} \prec \mathsf{Int}}\ (weakInt) \qquad \frac{}{\mathbb{P}, \mathcal{T} \vdash \mathtt{null} \prec \mathsf{O}}\ (weakNull)$$

$$\frac{\begin{array}{l}\mathbb{P}(\mathsf{f}) = \mathtt{function}\, \mathsf{f}(\mathtt{x}) : \mathsf{G}' \,\{...\} \\ \mathsf{G} \equiv \mathsf{G}'\end{array}}{\mathbb{P}, \mathcal{T} \vdash \mathsf{f} \prec \mathsf{G}}\ (weakFunc) \qquad \frac{\mathcal{T}(\iota) \leq \mathsf{O}}{\mathbb{P}, \mathcal{T} \vdash \iota \prec \mathsf{O}}\ (weakAddr)$$

$$\frac{\begin{array}{l}\mathsf{v} \notin Addr \\ \mathbb{P}, \mathcal{T} \vdash \mathsf{v} \prec \mathsf{t}\end{array}}{\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t}}\ (strongVal)$$

$$\frac{\begin{array}{l}\mathbb{P}, \mathcal{T} \vdash \iota \prec \mathsf{O} \\ \mathsf{H}(\iota) = \ll \mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_p : \mathsf{v}_p \gg \\ \mathsf{O}(\mathsf{m}) = (\mathsf{t}, \bullet) \implies \\ \qquad\qquad \exists\, i \in 1...p \ :\ \mathsf{m} = \mathsf{m}_i \ \wedge\ \mathbb{P}, \mathcal{T} \vdash \mathsf{v}_i \prec \mathsf{t} \ \text{ (for all members m)} \\ \forall\, i \in 1...p \ :\ \mathsf{O}(\mathsf{m}_i) = (\mathsf{t}, \circ) \implies \mathbb{P}, \mathcal{T} \vdash \mathsf{v}_i \prec \mathsf{t}\end{array}}{\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \iota \lhd \mathsf{O}}\ (strongAddr)$$

$$\frac{\begin{array}{l} dom(\mathcal{T}) = dom(\mathsf{H}) \\ \mathcal{T}(\iota) = \mathsf{O} \implies \mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \iota \lhd \mathsf{O} \ \text{ (for all addresses } \iota) \\ \mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \chi(\mathtt{this}) \lhd \Gamma(\mathtt{this}) \\ \mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \chi(\mathtt{x}) \lhd \Gamma(\mathtt{x})\end{array}}{\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \chi \diamond}\ (wlfHeapStack)$$

Figure 4.16.: Agreement between Environments, Heaps, Stacks and Values

Definition 3 *We say that $\mathcal{T}'$ extends $\mathcal{T}$ denoted by $\mathcal{T}' \sqsubseteq \mathcal{T}$ iff:*

- *$dom(\mathcal{T}) \subseteq dom(\mathcal{T}')$ and*
- *$\mathcal{T}(\iota) = \mathsf{O} \implies \mathcal{T}'(\iota) \sqsubseteq \mathsf{O}$ (for all addresses ι)*

Proposition 4 asserts that store type extension is transitive.

Proposition 4 *For program $\mathbb{P}$, value v, addresse ι, store typings $\mathcal{T}, \mathcal{T}', \mathcal{T}''$, type t:*

1. $\mathcal{T} \sqsubseteq \mathcal{T}',\ \mathcal{T}' \sqsubseteq \mathcal{T}'' \implies \mathcal{T} \sqsubseteq \mathcal{T}''$
2. $\mathcal{T} \sqsubseteq \mathcal{T}',\ \ \mathbb{P}, \mathcal{T}' \vdash \mathsf{v} \prec \mathsf{t} \implies \mathbb{P}, \mathcal{T} \vdash \mathsf{v} \prec \mathsf{t}$
3. $\mathcal{T} \sqsubseteq \mathcal{T}',\ \ \mathcal{T}'(\iota) \neq \mathcal{U}df \implies \mathcal{T}(\iota) \leq \mathcal{T}'(\iota)$

Note that Proposition 4.2 does not hold for strong agreement; that is $\mathcal{T} \sqsubseteq \mathcal{T}',\ \ \mathbb{P}, \mathsf{H}, \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t}$ does not imply $\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \mathsf{v} \lhd \mathsf{t}$.

Lemma 1 states that if an address strongly agrees with a type, heap and store typing, then it also agrees with an extended heap and store typing that leaves that address unchanged. Informally, one can think of an address ι in the heap as potentially having references to other objects in the heap. According to the definition of strong agreement with an address and a type t any member that references an object must weakly agree with the type for that member in t. In the extension of the store typing any of the objects referenced could have more members, *i.e.* belong to a subtype. This is fine from the perspective of ι as any address that weakly agrees with the type given in a store typing will agree with the type given in an extension to that store typing (see Proposition 4.2).

Lemma 1 *For a well-formed program $\mathbb{P}$, heaps H, H', address ι, type t and store typings $\mathcal{T}, \mathcal{T}'$ if:*

$$\mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \iota \lhd \mathsf{t} \tag{1}$$
$$\mathsf{H}(\iota) = \mathsf{H}'(\iota) \tag{2}$$
$$\mathcal{T}' \sqsubseteq \mathcal{T} \tag{3}$$

then:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota \lhd \mathsf{t} \tag{4}$$

Proof: From (1) and the definition of $(strongAddr)$ we get:

$$\mathbb{P}, \mathcal{T} \vdash \iota \prec \mathsf{t} \tag{5}$$
$$\mathsf{H}(\iota) = \ll \mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_p : \mathsf{v}_p \gg \tag{6}$$

$$\mathsf{t}(\mathsf{m}) = (\mathsf{t}'', \bullet) \implies \exists\, i \in 1...p \ :\ \mathsf{m} = \mathsf{m}_i \ \wedge\ \mathbb{P}, \mathcal{T} \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{7}$$

$$\forall\, i \in 1...p \ :\ \mathsf{t}(\mathsf{m}_i) = (\mathsf{t}'', \circ) \implies \mathbb{P}, \mathcal{T} \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{8}$$

From (3), (6), (7), (8) and Proposition 4.2 we get:

$$\mathsf{t}(\mathsf{m}) = (\mathsf{t}'', \bullet) \implies \exists\, i \in 1...p \ :\ \mathsf{m} = \mathsf{m}_i \ \wedge\ \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{9}$$

$$\forall\, i \in 1...p \ :\ \mathsf{t}(\mathsf{m}_i) = (\mathsf{t}'', \circ) \implies \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{10}$$

From (3) with (5) and Proposition 4.2 we get:

$$\mathbb{P}, \mathcal{T}' \vdash \iota \prec \mathsf{t} \tag{11}$$

From (2) and (6) we get:

$$\mathsf{H}'(\iota) = \ll \mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_p : \mathsf{v}_p \gg \tag{12}$$

From (11), (12), (9) and (10) with the definition of $(strongAddr)$ we establish (4).

□

Lemma 2 is used while proving type soundness for those cases where a change of environment is required; this includes the cases (new) and $(methCall)$. The overall idea is similar to that of Lemma 1; that is, if an environment and store typing agrees with a heap and stack, then that same environment and stack will agree with an extended store typing (and corresponding heap).

Lemma 2 *For program $\mathbb{P}$, typing functions $\mathcal{T}$, $\mathcal{T}'$, heaps H, H', stacks χ, χ' and environments Γ, Γ' if:*

$$\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \chi \diamond \tag{1}$$

$$\mathbb{P}, \Gamma', \mathcal{T}' \vdash \mathsf{H}', \chi' \diamond \tag{2}$$

$$\mathcal{T}' \sqsubseteq \mathcal{T} \tag{3}$$

then:

$$\mathbb{P}, \Gamma, \mathcal{T}' \vdash \mathsf{H}', \chi \diamond \tag{4}$$

Proof: From (1) and the definition of $(wlfHeapStack)$ we get:

$$\forall\, \mathsf{var} \ \in\ \{\mathtt{this}, \mathtt{x}\} : \mathbb{P}, \mathsf{H}, \mathcal{T} \vdash \chi(\mathsf{var}) \lhd \Gamma(\mathsf{var}) \tag{5}$$

We only consider the case when $\chi(\mathsf{var})$ is an address. When χ (var) is not an address the proof follows trivially from Proposition 3.4.

From (5) and Proposition 3.3 we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathcal{T}(\chi(\mathsf{var})) \leq \Gamma(\mathsf{var}) \tag{6}$$

From (2) and the definition of $(wlfHeapStack)$ we get:

$$dom(\mathcal{T}') = dom(\mathsf{H}') \tag{7}$$
$$\mathcal{T}'(\iota) = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota \lhd \mathsf{O}''\ \text{(for all addresses } \iota) \tag{8}$$

From (5) with the definition of $(strongAddr)$ (remembering that we are only considering addresses) we have that $\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathbb{P}, \mathcal{T} \vdash \chi(\mathsf{var}) \prec \Gamma(\mathsf{var})$, which with the definition of $(weakAddr)$ gives:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \chi(\mathsf{var}) \in dom(\mathcal{T}) \tag{9}$$

From (9), (3) and the definition of $\sqsubseteq$ we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \chi(\mathsf{var}) \in dom(\mathcal{T}') \tag{10}$$

From (8) and (10) we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi(\mathsf{var}) \lhd \mathcal{T}'(\chi(\mathsf{var})) \tag{11}$$

From (3) and (9) with Proposition 4.3 we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathcal{T}'(\chi(\mathsf{var})) \leq \mathcal{T}(\chi(\mathsf{var})) \tag{12}$$

From (6), (12) and transitivity of subtyping we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathcal{T}'(\chi(\mathsf{var})) \leq \Gamma(\mathsf{var}) \tag{13}$$

From (11), (13) and Proposition 3.2 we get:

$$\forall\,\mathsf{var}\ \in\ \{\mathtt{this},\mathtt{x}\} : \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi(\mathsf{var}) \lhd \Gamma(\mathsf{var}) \tag{14}$$

From (7), (8) and (14) we apply rule $(wlfHeapStack)$ to get:

$$\mathbb{P}, \Gamma, \mathcal{T}' \vdash \mathsf{H}', \chi \diamond \tag{15}$$

We see that (15) establishes (4).

□

4.3.2 Proof of Soundness

Theorem 1 asserts that if the evaluation of a well-typed expression terminates, either it produces a value or a null-pointer exception, `nullPntrExc`, but is never stuck, `stuckErr`. Moreover, if the evaluation produces a value, then it will agree with the type of the expression. The heap and stack produced agree with respect to the final environment and extended store typing.

As far as divergent expressions go Theorem 1 does not say anything. However, the operational semantics forces convergence for standard typing errors or access to members undefined for an object, see Section 3.5.1. Therefore, Theorem 1 suffices to ensure that execution of well-typed expression never accesses non-existing identifiers or members, and is never stuck.

Theorem 1 *For a well-formed program* $\mathbb{P}$*, type* t'*, environments* Γ*,* Γ'*, store typings* $\mathcal{T}$*,* $\mathcal{T}'$*, heaps* H*,* H'*, stacks* χ*,* χ'*, and expression* e*, such that:*

$$\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}' \parallel \Gamma' \tag{1}$$
$$\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \chi \diamond \tag{2}$$
$$\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{w}, \mathsf{H}', \chi' \tag{3}$$

then for some $\mathcal{T}'$*, either* $\mathsf{w} = \mathsf{v}$ *and:*

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t}' \tag{4}$$
$$\mathbb{P}, \Gamma, \mathcal{T}' \vdash \mathsf{H}', \chi' \diamond \tag{5}$$
$$\mathcal{T}' \sqsubseteq \mathcal{T} \tag{6}$$

or $\mathsf{w} =$ `nullPntrExc` *and:*

$$\mathbb{P}, \Gamma, \mathcal{T}' \vdash \mathsf{H}', \chi' \diamond \tag{7}$$
$$\mathcal{T}' \sqsubseteq \mathcal{T} \tag{8}$$

Proof: We show the interesting cases $(memAss)$, (new) and $(memCall)$. The proof is by induction on the derivation $\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{w}, \mathsf{H}', \chi'$.

Case: $(memAss)$ and e is $\mathsf{var.m = e_2}$

Given that e is $\mathsf{var.m} = \mathsf{e_2}$ from (3) and the definition of the operational semantics we get:

$$\mathsf{var}, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}, \chi \tag{9}$$

$$\iota = \chi(\mathsf{var}) \tag{10}$$

$$\mathsf{e_2}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}'', \chi' \tag{11}$$

$$\mathsf{H}' = \mathsf{H}''[\iota \mapsto \mathsf{H}''(\iota)[\mathsf{m} \mapsto \mathsf{v}]] \tag{12}$$

$$\mathsf{w} = \mathsf{v} \tag{13}$$

From (1) we have that $\mathsf{var.m} = \mathsf{e_2}$ is well typed, and so from type rule $(assignAdd)$ we get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e_2} : \mathsf{t}' \parallel \Gamma'' \tag{14}$$

$$\mathsf{var} = \mathsf{e}' \text{ is not a subexpression of } \mathsf{e_2} \tag{15}$$

$$\Gamma''(\mathsf{var}) = \mathsf{O} \tag{16}$$

$$\mathsf{O}(\mathsf{m}) = (\mathsf{t}, \psi) \tag{17}$$

$$\mathsf{t}' \leq \mathsf{t} \tag{18}$$

$$\Gamma' = \Gamma''[\mathsf{var} \mapsto \mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \bullet)]] \tag{19}$$

With (14), (2) and (11) we apply the inductive hypothesis to get for some $\mathcal{T}''$:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \mathsf{v} \lhd \mathsf{t}' \tag{20}$$

$$\mathbb{P}, \Gamma'', \mathcal{T}'' \vdash \mathsf{H}'', \chi' \diamond \tag{21}$$

$$\mathcal{T}'' \sqsubseteq \mathcal{T} \tag{22}$$

From (15) and (10) we get:

$$\chi(\mathsf{var}) = \chi'(\mathsf{var}) = \iota \tag{23}$$

From (21), (23), (16) and the definition of $(wlfHeapStack)$ we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota \lhd \mathsf{O} \tag{24}$$

From (24) with Proposition 3.3 we get:

$$\mathcal{T}''(\iota) \leq \mathsf{O} \tag{25}$$

Let $\mathcal{T}''(\iota)(\mathsf{m}) = (\mathsf{t}'', \psi')$ for some t'' and ψ'. From (25), (17) and the definition of subtyping we get: $\mathsf{t} \equiv \mathsf{t}''$ and $\psi' \leq \psi$. Now let:

$$\mathcal{T}' = \mathcal{T}''[\iota \mapsto \mathcal{T}''[\mathsf{m} \mapsto (\mathsf{t}, \bullet)]] \tag{26}$$

From (26) and the definition of $\sqsubseteq$ we get:

$$\mathcal{T}' \sqsubseteq \mathcal{T}'' \tag{27}$$

From (24) and (27) we know that $\mathcal{T}''$ is defined for ι, and with Proposition 4.3 we get:

$$\mathcal{T}'(\iota) \leq \mathcal{T}''(\iota) \tag{28}$$

From (12) and (26) with the definition of update we have:

$$\forall\, \iota' \neq \iota : \mathsf{H}'(\iota') = \mathsf{H}''(\iota') \tag{29}$$

$$\forall\, \iota' \neq \iota : \mathcal{T}'(\iota') = \mathcal{T}''(\iota') \tag{30}$$

From (21) and the definition of $(wlfHeapStack)$ we have:

$$\forall\, \iota \;:\; \mathcal{T}''(\iota) = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota \lhd \mathsf{O}'' \tag{31}$$

From (31), (29) and (27) we apply Lemma 1 (for all addresses $\iota' \neq \iota$) to get:

$$\forall\, \iota' \neq \iota \;:\; \mathcal{T}'(\iota') = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota' \lhd \mathsf{O}'' \tag{32}$$

We must now show that $\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota \lhd \mathcal{T}'(\iota)$.

From (21) and (23) with the definition of $(wlfHeapStack)$ we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota \lhd \mathcal{T}''(\iota) \tag{33}$$

From (12), (26) and the definition of update we get:

$$\forall\, \mathsf{m} \neq \mathsf{m}' \;:\; \mathsf{H}'(\iota)(\mathsf{m}') = \mathsf{H}''(\iota)(\mathsf{m}') \tag{34}$$

$$\forall\, \mathsf{m} \neq \mathsf{m}' \;:\; \mathcal{T}'(\iota)(\mathsf{m}') = \mathcal{T}''(\iota)(\mathsf{m}') \tag{35}$$

Let:

$$\mathsf{H}''(\iota) = \ll\!\mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_p : \mathsf{v}_p\!\gg \tag{36}$$

From (33) with (36) and the definition of $(strongAddr)$ we get:

$$\mathcal{T}''(\iota)(\mathsf{m}) = (\mathsf{t}'', \bullet) \implies \exists\, i \in 1...p \;:\; \mathsf{m} = \mathsf{m}_i \;\wedge\; \mathbb{P}, \mathcal{T}'' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{37}$$

$$\forall\, i \in 1...p \;:\; \mathcal{T}''(\iota)(\mathsf{m}_i) = (\mathsf{t}'', \circ) \implies \mathbb{P}, \mathcal{T}'' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{38}$$

Consider all members of $\mathsf{H}'(\iota)$ apart from m. From (37), (38) with (34), (35), (36), (27) and Proposition 4.2 we get:

$$\begin{aligned} &\mathcal{T}'(\iota)(\mathsf{m}') = (\mathsf{t}'', \bullet) \wedge \mathsf{m}' \neq \mathsf{m} \\ &\qquad\qquad \implies \exists\, i \in 1...p \;:\; \mathsf{m}' = \mathsf{m}_i \;\wedge\; \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \end{aligned} \tag{39}$$

$$\forall\, i \in 1...p \;:\; \mathcal{T}'(\iota)(\mathsf{m}_i) = (\mathsf{t}'', \circ) \wedge \mathsf{m}_i \neq \mathsf{m} \implies \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{40}$$

We now need to show weak agreement of member m of $\mathsf{H}'(\iota)$. From (12) and the definition of update we have:

$$\mathsf{H}'(\iota)(\mathsf{m}) = \mathsf{v} \tag{41}$$

From (26) and the definition of update we have:

$$\mathcal{T}'(\iota)(\mathsf{m}) = (\mathsf{t}, \bullet) \tag{42}$$

Hence, we must show that $\mathbb{P}, \mathcal{T}' \vdash \mathsf{v} \prec \mathsf{t}$. We proceed by case analysis on v.

If $\mathsf{v} \notin$ *Addr* then from (20) and Proposition 3.4 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t}' \tag{43}$$

From (43) with (18) and proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t} \tag{44}$$

From (44) with the definition of $(strongVal)$ we get:

$$\mathbb{P}, \mathcal{T}' \vdash \mathsf{v} \prec \mathsf{t} \tag{45}$$

Now consider when v is an address. There are two cases to consider: $\mathsf{v} = \iota$ and $\mathsf{v} \neq \iota$.

If $\mathsf{v} = \iota$ then from (20) we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota \lhd \mathsf{t}' \tag{46}$$

From (46) with Proposition 3.3 we get:

$$\mathcal{T}''(\iota) \leq \mathsf{t}' \tag{47}$$

From (28), (47) and (18) with the transitivity of subtyping we get:

$$\mathcal{T}'(\iota) \leq \mathsf{t} \tag{48}$$

From (48) with the definition of $(weakAddr)$ we get:

$$\mathbb{P}, \mathcal{T}' \vdash \mathsf{v} \prec \mathsf{t} \tag{49}$$

If $\mathsf{v} = \iota'$ and $\iota \neq \iota'$, then from (20) we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota' \lhd \mathsf{t}' \tag{50}$$

From (50), (29) ($\mathsf{H}''(\iota') = \mathsf{H}'(\iota')$) and (27) we apply Lemma 1 for ι' to get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota' \lhd \mathsf{t}' \tag{51}$$

From (51) and (18) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota' \lhd \mathsf{t} \tag{52}$$

From (52), $\mathsf{v} = \iota'$ and the definition of $(strongAddr)$ we get:

$$\mathbb{P}, \mathcal{T}' \vdash \mathsf{v} \prec \mathsf{t} \tag{53}$$

From (12) with (36) for $q \geq p$ let:

$$\mathsf{H}'(\iota) = \ll \mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_q : \mathsf{v}_q \gg \tag{54}$$

From (45), (49) and (53) with (39), (40) and (54) we get:

$$\mathcal{T}'(\iota)(\mathsf{m}) = (\mathsf{t}'', \bullet) \implies \exists\, i \in 1...q \ :\ \mathsf{m} = \mathsf{m}_i \ \wedge\ \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{55}$$

$$\forall\, i \in 1...q \ :\ \mathcal{T}'(\iota)(\mathsf{m}_i) = (\mathsf{t}'', \circ) \implies \mathbb{P}, \mathcal{T}' \vdash \mathsf{v}_i \prec \mathsf{t}'' \tag{56}$$

Note that from (26) we have that ι is defined in $\mathcal{T}'$, and therefore with $(weakAddr)$ we get:

$$\mathbb{P}, \mathcal{T}' \vdash \iota \prec \mathcal{T}'(\iota) \tag{57}$$

From (57) (54), (55) and (56) with the definition of $(strongAddr)$ we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota \lhd \mathcal{T}'(\iota) \tag{58}$$

With (58) and (32) we get:

$$\forall \iota' : \ \mathcal{T}'(\iota') = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota' \lhd \mathsf{O}'' \tag{59}$$

We now show that for $\mathsf{var}' \in \{\mathtt{this}, \mathtt{x}\}$ we have $\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi'(\mathsf{var}') \lhd \Gamma'(\mathsf{var}')$.

Consider the case when $\mathsf{var}' = \mathsf{var}$. Let O' be:

$$\mathsf{O}' = \mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \bullet)] \tag{60}$$

From (25) with (26), (60) and definition of subtyping we get:

$$\mathcal{T}'(\iota) \leq \mathsf{O}' \tag{61}$$

From (19) with (60) we get:

$$\Gamma'(\mathsf{var}) = \mathsf{O}' \tag{62}$$

From (62), (23), (58), (61) and Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi'(\mathsf{var}) \lhd \Gamma'(\mathsf{var}) \tag{63}$$

Consider when $\mathsf{var} \neq \mathsf{var}'$ and $\chi'(\mathsf{var}') \neq \iota$.

From (19) and the definition of update we have:

$$\Gamma''(\mathsf{var}') = \Gamma'(\mathsf{var}') \tag{64}$$

From (21) and the definition of $(wlfHeapStack)$ we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \chi'(\mathsf{var}') \lhd \Gamma''(\mathsf{var}') \tag{65}$$

From (65) with (64) we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \chi'(\mathsf{var}') \lhd \Gamma'(\mathsf{var}') \tag{66}$$

With (66), $\chi'(\mathsf{var}') \neq \iota$, (29) and (27) we apply Lemma 1 to get:

$$\chi'(\mathsf{var}') \in Addr \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi'(\mathsf{var}') \lhd \Gamma'(\mathsf{var}') \tag{67}$$

From (66) and Proposition 3.4 we get:

$$\chi'(\mathsf{var}') \notin \mathit{Addr} \implies \mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \chi'(\mathsf{var}') \lhd \Gamma'(\mathsf{var}') \tag{68}$$

Consider when $\mathsf{var} \neq \mathsf{var}'$ and $\chi(\mathsf{var}') = \iota$.

From (21) and the definition of $(wlfHeapStack)$ with (64) we get:

$$\mathbb{P}, \mathsf{H}'', \mathcal{T}'' \vdash \iota \lhd \Gamma'(\mathsf{var}') \tag{69}$$

From (69) and Proposition 3.3 we get:

$$\mathcal{T}''(\iota) \leq \Gamma'(\mathsf{var}') \tag{70}$$

From (70), (28) and the transitivity of subtyping we get:

$$\mathcal{T}'(\iota) \leq \Gamma'(\mathsf{var}') \tag{71}$$

From (58), (71) and Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \iota \lhd \Gamma'(\mathsf{var}') \tag{72}$$

From (21) and the definition of $(strongAddr)$ we have that $dom(\mathcal{T}'') = dom(\mathsf{H}'')$, and so with (12), (26) and the definition of update we get:

$$dom(\mathcal{T}') = dom(\mathsf{H}') \tag{73}$$

From (73), (59), (63), (67), (68), (72) and the definition of $(wlfHeapStack)$ we get:

$$\mathbb{P}, \Gamma', \mathcal{T}' \vdash \mathsf{H}', \chi' \diamond \tag{74}$$

We see that (74) and (13) establish (5).

We must now show that $\mathbb{P}, \mathsf{H}', \mathcal{T}' \vdash \mathsf{v} \lhd \mathsf{t}'$.

If $\mathsf{v} \notin \mathit{Addr}$ then (43) establishes (4). If $\mathsf{v} = \iota'$ and $\iota' \neq \iota$ then (51) establishes (4). If $\mathsf{v} = \iota$ then (47), (28), the transitivity of subtyping and (58) with Proposition 3.2 establish (4).

Finally, (22), (27) and Proposition 4.1 establish (6).

Case: $(memAss)$ $\mathsf{e}_1 \neq \mathsf{var}$

Proof is straightforward.

Case: (new)

Given that e is $\texttt{new}\,\mathsf{f}(\mathsf{e}')$ from (3) and the operational semantics rule (new) we get:

$$\mathsf{e}', \mathsf{H}, \chi \twoheadrightarrow\ \mathsf{v}', \mathsf{H}_1, \chi' \tag{75}$$

$$\mathsf{P}(\mathsf{f}) = \texttt{function}\,\mathsf{f}(\texttt{x}) : \mathsf{G}\{\ \mathsf{e}''\} \tag{76}$$

$$\iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto \ll\gg] \tag{77}$$

$$\mathsf{e}'', \mathsf{H}_2, \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow\ \mathsf{v}, \mathsf{H}', \chi'' \tag{78}$$

$$\mathsf{w} = \iota \tag{79}$$

From (1) we have that $\texttt{new}\,\mathsf{f}(\mathsf{e}')$ is well typed, and from the type rule $(call)$ we get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}' : \mathsf{t} \parallel \Gamma' \tag{80}$$

$$\mathbb{P}(\mathsf{f}) = \texttt{function}\,\mathsf{f}(\texttt{x}) : \mathsf{G}\ \{\mathsf{e}''\} \tag{81}$$

$$\mathsf{t} \leq \mathsf{G}(\texttt{x}) \tag{82}$$

$$\{\mathsf{t}'' \mid (\mathsf{G}(\texttt{this}))(\mathsf{m}) = (\mathsf{t}'', \bullet)\} = \emptyset \tag{83}$$

$$\mathsf{t}' = \mathsf{G}(\texttt{ret}) \tag{84}$$

From (80), (2) and (75) we apply the inductive hypothesis to get for some $\mathcal{T}'$:

$$\mathbb{P}, \mathsf{H}_1, \mathcal{T}' \vdash \mathsf{v}' \lhd \mathsf{t} \tag{85}$$

$$\mathbb{P}, \Gamma', \mathcal{T}' \vdash \mathsf{H}_1, \chi'\ \diamond \tag{86}$$

$$\mathcal{T}'\ \sqsubseteq \mathcal{T} \tag{87}$$

Now let:

$$\mathcal{T}'' = \mathcal{T}'[\iota \mapsto \mathsf{G}(\texttt{this})] \tag{88}$$

From (88) and the definition of $\sqsubseteq$ we have:

$$\mathcal{T}''\ \sqsubseteq \mathcal{T}' \tag{89}$$

Trivially, by $(weakAddr)$ (noting that with (88) ι is defined in $\mathcal{T}''$) we get:

$$\mathbb{P}, \mathcal{T}'' \vdash \iota \prec \mathcal{T}''(\iota) \tag{90}$$

From (83) we have that G ($\mathtt{this}$) has no definite members; with (90), (77), (88) and the definition of $(\mathit{strongAddr})$ we get:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \iota \lhd \mathsf{G}(\mathtt{this}) \tag{91}$$

From (86) and the definition of $(\mathit{strongAddr})$ we get:

$$\forall \iota' : \mathcal{T}'(\iota') = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}_1, \mathcal{T}' \vdash \iota' \lhd \mathsf{O}'' \tag{92}$$

From (92), (88), (77), (91) and the definition of $(\mathit{strongAddr})$ we get:

$$\forall \iota' : \mathcal{T}''(\iota') = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \iota' \lhd \mathsf{O}'' \tag{93}$$

From (85) and (82) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}_1, \mathcal{T}' \vdash \mathsf{v}' \lhd \mathsf{G}(\mathtt{x}) \tag{94}$$

Given that $\mathsf{v}' \neq \iota$ and if v' is an address, then none of the corresponding object's members may point to ι; with (88), (77), (94) and definition of $(\mathit{strongAddr})$ we get:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \mathsf{v}' \lhd \mathsf{G}(\mathtt{x}) \tag{95}$$

Let:

$$\Gamma_1 = \{\mathtt{this} : \mathsf{G}(\mathtt{this}), \mathtt{x} : \mathsf{G}(\mathtt{x})\} \tag{96}$$
$$\chi_1 = \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \tag{97}$$

From (86) and the definition of $(\mathit{wlfHeapStack})$ we get:

$$\mathit{dom}(\mathsf{H}_1) = \mathit{dom}(\mathcal{T}') \tag{98}$$

From (98) with (88) and (77) we get:

$$\mathit{dom}(\mathsf{H}_2) = \mathit{dom}(\mathcal{T}'') \tag{99}$$

From (99), (93), (91), (95) and the definition of $(\mathit{wlfHeapStack})$ we get:

$$\mathbb{P}, \Gamma_1, \mathcal{T}'' \vdash \mathsf{H}_2, \chi_1 \diamond \tag{100}$$

Given the program, $\mathbb{P}$, is well-formed, with (76) and $(prog)$, we get for some t'', Γ'':

$$\mathbb{P}, \Gamma_1 \vdash \mathsf{e}' : \mathsf{t}'' \parallel \Gamma'' \tag{101}$$
$$\mathsf{t}'' \leq \mathsf{G}(\texttt{ret}) \tag{102}$$

With (101), (100) and (78) we apply the inductive hypothesis to get for some $\mathcal{T}'''$:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}''' \vdash \mathsf{v} \lhd \mathsf{t}'' \tag{103}$$
$$\mathbb{P}, \Gamma'', \mathcal{T}''' \vdash \mathsf{H}', \chi'' \diamond \tag{104}$$
$$\mathcal{T}''' \sqsubseteq \mathcal{T}'' \tag{105}$$

From (103) and (102) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}''' \vdash \mathsf{v} \lhd \mathsf{G}(\texttt{ret}) \tag{106}$$

We see that (106) with (84) and (79) establish (4).

From (89), (105) and Proposition 4.1 we get:

$$\mathcal{T}''' \sqsubseteq \mathcal{T}' \tag{107}$$

From (86), (104) and (107) we apply Lemma 2 to get:

$$\mathbb{P}, \Gamma', \mathcal{T}''' \vdash \mathsf{H}', \chi' \diamond \tag{108}$$

We see that (108) establishes (5).

From (87), (107) and Proposition 4.1 we get:

$$\mathcal{T}''' \sqsubseteq \mathcal{T} \tag{109}$$

We see that (109) establishes (6).

Case: $(memCall)$

Given that e is $\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)$, from (3) and the operational semantics rule $(memCall)$ we get:

$$\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \tag{110}$$
$$\mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}_2, \chi' \tag{111}$$
$$\mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \tag{112}$$
$$\mathsf{P}(\mathsf{f}) = \texttt{function}\ \mathsf{f}(\texttt{x}) : \mathsf{G}\{\ \mathsf{e}'\} \tag{113}$$
$$\mathsf{e}', \mathsf{H}_2, \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'' \tag{114}$$
$$\mathsf{w} = \mathsf{v} \tag{115}$$

From (1) we have that $\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)$ is well typed, and from the type rule $(memCall)$ we get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{O} \parallel \Gamma' \tag{116}$$
$$\mathsf{O}(\mathsf{m}) = (\mathsf{G}, \bullet) \tag{117}$$
$$\mathbb{P}, \Gamma' \vdash \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma'' \tag{118}$$
$$\mathsf{t}' \leq \mathsf{G}(\texttt{x}) \tag{119}$$
$$\mathsf{O} \leq \mathsf{G}(\texttt{this}) \tag{120}$$
$$\mathsf{t}' = \mathsf{G}(\texttt{ret}) \tag{121}$$

From (116), (2) and (110) we apply the inductive hypothesis to get for some $\mathcal{T}'$:

$$\mathbb{P}, \mathsf{H}_1, \mathcal{T}' \vdash \iota \lhd \mathsf{O} \tag{122}$$
$$\mathbb{P}, \Gamma', \mathcal{T}' \vdash \mathsf{H}_1, \chi_1 \diamond \tag{123}$$
$$\mathcal{T}' \sqsubseteq \mathcal{T} \tag{124}$$

From (118), (111) and (123) we apply the inductive hypothesis to get for some $\mathcal{T}''$:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \mathsf{v}' \lhd \mathsf{t}' \tag{125}$$
$$\mathbb{P}, \Gamma'', \mathcal{T}'' \vdash \mathsf{H}_2, \chi' \diamond \tag{126}$$
$$\mathcal{T}'' \sqsubseteq \mathcal{T}' \tag{127}$$

From (125) and (119) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \mathsf{v}' \lhd \mathsf{G}(\texttt{x}) \tag{128}$$

From (122) and Proposition 3.3 we get that $\mathcal{T}'(\iota) \leq \mathsf{O}$. From (127) and Proposition 4.3 we get that $\mathcal{T}''(\iota) \leq \mathcal{T}'(\iota)$. Hence, with (120) and the transitivity of subtyping we get:

$$\mathcal{T}''(\iota) \leq \mathsf{G}(\mathtt{this}) \tag{129}$$

From (126) and the definition of $(strongAddr)$ we get:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \iota \lhd \mathcal{T}''(\iota) \tag{130}$$

From (130) and (129) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \iota \lhd \mathsf{G}(\mathtt{this}) \tag{131}$$

From (126) and the definition of $(wlfHeapStack)$ we get:

$$dom(\mathsf{H}_2) = dom(\mathcal{T}'') \tag{132}$$

$$\forall \iota' : \mathcal{T}''(\iota') = \mathsf{O}'' \implies \mathbb{P}, \mathsf{H}_2, \mathcal{T}'' \vdash \iota' \lhd \mathsf{O}'' \tag{133}$$

Let:

$$\Gamma_1 = \{\mathtt{this} : \mathsf{G}(\mathtt{this}), \mathtt{x} : \mathsf{G}(\mathtt{x})\} \tag{134}$$

$$\chi_1 = \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \tag{135}$$

From (132), (133), (131), (128) and the definition of $(wlfHeapStack)$ we get:

$$\mathbb{P}, \Gamma_1, \mathcal{T}'' \vdash \mathsf{H}_2, \chi_1 \diamond \tag{136}$$

Given the program, $\mathbb{P}$, is well-formed, with (113) and $(prog)$ for some t'' and Γ'' we get:

$$\mathbb{P}, \Gamma_1 \vdash \mathsf{e}' : \mathsf{t}'' \parallel \Gamma''' \tag{137}$$

$$\mathsf{t}'' \leq \mathsf{G}(\mathtt{ret}) \tag{138}$$

From (137), (136) and (114) we apply the inductive hypothesis to get for some $\mathcal{T}'''$:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}''' \vdash \mathsf{v} \lhd \mathsf{t}'' \tag{139}$$

$$\mathbb{P}, \Gamma''', \mathcal{T}''' \vdash \mathsf{H}', \chi'' \diamond \tag{140}$$

$$\mathcal{T}''' \sqsubseteq \mathcal{T}'' \tag{141}$$

From (139) and (138) with Proposition 3.2 we get:

$$\mathbb{P}, \mathsf{H}', \mathcal{T}''' \vdash \mathsf{v} \lhd \mathsf{G}(\texttt{ret}) \tag{142}$$

We see that (142) with (115) and (121) establish (4).

From (127) and (141) with Proposition 4.1 we get:

$$\mathcal{T}''' \sqsubseteq \mathcal{T}' \tag{143}$$

From (126), (140) and (143) we apply Lemma 2 to get:

$$\mathbb{P}, \Gamma'', \mathcal{T}''' \vdash \mathsf{H}', \chi' \diamond \tag{144}$$

We see that (144) establishes (5).

From (124), (143) and Proposition 4.1 we get:

$$\mathcal{T}''' \sqsubseteq \mathcal{T} \tag{145}$$

We see that (145) establishes (6).

□

4.4 Related Work

We now discuss some work for type systems that are related and/or have influenced the design of the types presented in this chapter.

Abadi and Cardelli

Type systems for object based languages have been explored and developed in both the imperative and functional settings. In [2, 3] Abadi et al. develop an array of type systems for functional and imperative object calculi. Of particular interest is the **O-1** language [3] (pages 153-165). The typing of objects in **O-1** is similar to the object types used in $\mathrm{JS}_0^{\mathsf{T}}$. For example, the type t_2 from Figure 4.9, defined as:

$$\mu\,\alpha.[\texttt{mSec} : (\mathsf{Int}, \bullet),\ \texttt{add} : ((\alpha \times \alpha \to \alpha), \bullet)]$$

Would be represented in **O-1** as:

$$\mathbf{Object}(\mathbf{X})[\texttt{mSec}^{\circ} : \mathsf{Int}, \texttt{add}^{\circ} : \mathbf{X} \to \mathbf{X}]$$

Note that **O-1** does not have potential members and that functions don't specify the type of the receiver (it is implicit). Of interest are the *variance* annotations [3] given to each member: $\mathtt{m}^{\circ}, \mathtt{m}^{+}, \mathtt{m}^{-}$. Variance annotations allow each member to specify whether their type is invariant, $\mathtt{m}^{\circ}$, contravariant, $\mathtt{m}^{+}$, or covariant, $\mathtt{m}^{-}$. Covariant members can be covariantly subtyped but not updated. Contravariant members can be contravariantly subtyped but not invoked. Invariant members have the obvious meaning. Thus, function types can be covariantly or contravariantly subtyped. The subtype judgement $\mathsf{t} \leq \mathsf{t}'$ for object types differs from $\mathrm{JS}_0^{\mathsf{T}}$ whereby the type t' is unfolded, but t is not. The bound variable in t is assumed to be a subtype of t', $\alpha \leq \mathsf{t}'$. This coupled with the variance annotations means that the subtyping used in **O-1** is stronger than that used in $\mathrm{JS}_0^{\mathsf{T}}$. For example, consider the following $\mathrm{JS}_0^{\mathsf{T}}$ types:

$$\mathsf{t} = \mu\,\alpha.[\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\mathsf{t} \times \mathsf{t} \to \alpha), \bullet)]$$
$$\mathsf{t}' = \mu\,\alpha'.[\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{leapYear} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\mathsf{t} \times \mathsf{t} \to \alpha'), \bullet)]$$

It would not be the case that $\mathsf{t}' \leq \mathsf{t}$ holds because when unfolded the return type of `add` in t is different from the return type in t'. In **O-1** member `add` would be marked as covariant and through the assumption $\alpha \leq \mathsf{t}'$ we could prove $\mathsf{t} \leq \mathsf{t}'$.

Thiemann

In [66] Thiemann gives a type system for a considerable subset of JavaScript. The language is comprised of five core constructs: $\mathsf{e}_1\,[\mathsf{e}_2\,]$ - property access, `new` $\mathsf{e}_0\,(\mathsf{e}_1...)$ - create new object, $\mathsf{e}_0\,(\mathsf{e}_1...)$ - method call, $\mathsf{e}_0 = \mathsf{e}_1$ - assignment and $p(\mathsf{e}_1...)$ - function call. A set of constructs for primitives, like string and integer, is also included along with conditionals and while loops.

The aim of this work was to provide a type system that models JavaScript as closely as possible. In particular, Thiemann points out that JavaScript allows access to non-existent members and coerces values, in most cases, to prevent a runtime error (see Section 3.1.6). A matching relation between types attempts to convert one type to another. For example:

```
var x = "Hello"
x.m = 10
```

Variable `x` initially has a `String` type; with property access `x.m` variable `x` is coerced into an object type.

According to the paper the type system statically detects the following potential runtime errors:

1. Function call and `new` of a value that is not a function.
2. Applying an arithmetic operator to an object that is not a wrapped number.
3. Accessing a member of the `null` or the `undefined` value.
4. Accessing a non-existent variable.

Types, τ, comprise sums of base types (boolean, number, undefined, null and string), object types (plain objects and wrapper objects) and function types. Base types are embellished with *type indices* which consist of values of the appropriate type for that base type or $\top$. For example: `String("Hello")` , `Bool(true)` `Bool(`$\top$`)` , `Number(10)` . A constant value is a refinement of $\top$.

Object types have two components: a feature part and a row part. The feature part corresponds to a base type or function type. One can think of the feature component of an object type as the kind of base or function type being wrapped ($\perp$ is used when the object is not wrapping). The row part, @, corresponds to the properties of the object and is a mapping from strings to types. At the end of each row is a special element @ which gives a default type to members that are not mentioned in the row. This models what happens when accessing a non-existent member of an object. In the next example the first type represents a wrapped string and the second type an object with two members `name` and `male`:

$$\texttt{Obj}(\texttt{String}(\text{"}\texttt{Hello}\text{"}))(\text{"}\texttt{length}\text{"} : \texttt{Number}(5), \delta\tau)$$
$$\texttt{Obj}(\perp)(\text{"}\texttt{name}\text{"} : \texttt{String}(\text{"}\texttt{Chris}\text{"}), \text{"}\texttt{male}\text{"} : \texttt{Boolean}(\texttt{true}))$$

Function types comprise a type for the receiver, `this`, and a row type representing the arguments. The number of formal parameters to a function does not need to match the number of actual parameters; when a function is called with less actual parameters than formal parameters, the unspecified parameters are given the undefined value. Consider the following function type:

$$\texttt{Function}(\texttt{this} : \tau_0; @ \to \tau')$$

The type system has been proven sound, and at the time of writing a prototype implementation is underway.

Anderson, Barbanera, Dezani-Ciancaglini and Drossopoulou

In [9, 26] *alias types* [65, 69] are adapted to give a type system for an object calculus with delegation δ[10]. Alias types produced type systems which are collections of *aliasing constraints*. These constraints describe the shape of the store and every function uses them to specify the store that it expects. The pointers have *singleton types* which are the locations themselves.

The δ calculus allows object members to be added and removed at will. Objects can specify a list of delegate objects; when a method is called on an object, if it is not found then the delegate objects are searched and so on. δ is quite different from JS_0 (it is quite *low level*), but the types do share some similarities. Objects are rows of members and delegates. Members of an object type hold references to other object types. For example, consider the following typing environment:

$$\begin{aligned}\Gamma \;=\; \{&\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota') \mid \mathsf{d}_2 : \mathsf{Obj}(\iota'') \rangle\!\rangle \parallel \langle\!\langle\;\rangle\!\rangle,\\ &\iota' : \langle\!\langle\;\rangle\!\rangle \parallel \langle\!\langle \mathsf{m} : \mathsf{Obj}(\iota'') \rangle\!\rangle,\\ &\iota'' : \langle\!\langle\;\rangle\!\rangle \parallel \langle\!\langle \mathsf{m}' : \langle \Gamma', \mathsf{Obj}(\iota''), \varphi \rangle \rangle\!\rangle\}\end{aligned}$$

The type of address ι describes an object with two delegates of type ι' and ι''. Type ι' has a member m that has type ι''. Type ι'' has a member m' that is a function type whose return type is $\mathsf{Obj}(\iota'')$ and effect is φ. The effect describes how the store is changed; δ has rich effects whereby members can be added and removed from object types. The environment used in type checking is similar to the store typing used in the proof of soundness for $\mathrm{JS}_0^{\mathsf{T}}$. As a technical detail, each function carries an environment for type checking. Unlike $\mathrm{JS}_0^{\mathsf{T}}$, with δ the code for a method is stored in the object itself.

Bono, Damiani and Giannini

In [17] Bono et al. develop a calculus for a JavaScript-like language. The aim was to be able to establish whether a JavaScript program would work in different environments, *i.e.* the various web browsers. Their language contains a primitive for testing the presence of a member: `isdef(e,a)` where `e` is an expression that evaluates to an address, and `a` is an attribute name. `isdef(e,a)` evaluates to true if the object contains the attribute `a` and false otherwise.

In [26] Daniani and Giannini develop a type system for the calculus in [17]. The work was inspired by the application of alias types in [9, 26]. Object types detail the members of an object where the types are addresses or the base type integer. As in $\mathrm{JS}_0^{\mathsf{T}}$, there are annotations on members, *e.g.* potential members, denoted by '?', or the requirement that a member does not exist, denoted by '!'. Potential members have the same role here as they do in $\mathrm{JS}_0^{\mathsf{T}}$.

Both this work and [9, 26] require *another* environment to be carried with the type of a method. This environment is used to type check the method. When type checking a program the environment for a method can mention members not yet present. As commented in [26], this could complicate the design of a type inference algorithm.

Anderson and Drossopoulou

In [11, 12] a different approach to typing JavaScript is described. Code development starts with a JavaScript like language called BabyJ. Through a series of iterations, using type inference, the code is translated to Java. Code starts in an object based style and is translated to a class based style. The type system is a subset of the one used in Java with the exception of the type *Any*. The *Any* type acts as an escape from the type system, *i.e.* the type system ignores variables with *Any* type. New types are introduced through constructor functions that detail the members of objects they create. Functions used as methods specify the constructor function they belong too; it is not possible for a method to be shared between two constructor functions.

The process of going from object based to class based involves replacing occurrences of the *Any* type with the new types introduced. A type inference algorithm based on [57, 58] is used to infer the role of functions, *i.e.* constructor, method or global. The types are nominal with name equivalence; there is no subtyping. Soundness of the type system was proven in the case when there were no occurrences of the *Any* type in the program.

Anderson and Giannini

An earlier version of the type system presented in this thesis is given in [13]. There are many presentational differences with the work shown here. In particular:

- Members of object types are only annotated when they are potential; absence of an annotation indicates that the member is definite.
- The type annotations of functions is in the style of [11, 12] with a separate type annotation for the parameter, return type and receiver:

```
function f(x:t):t′ {
  this:t″
  ...
}
```

- There is no (fix) congruence rule.
- In this thesis member selection is defined in a manner that allows a cleaner presentation of the type rules. For example, in [13] an explicit unfolding of the receiver in rule $(assign - add)$ is required. This is not required in rule $(assignAdd)$, in Figure 4.11, as member selection closes the type.
- The agreement relation between values and types is given in a co-inductive manner, but no proof of soundness is given.

Fisher, Bono and Liquori

Extensible objects have been studied in a functional setting in [33]. An imperative calculus for extensible objects was proposed by Bono and Fisher in [18]; there are two types for objects: the *pro*-types that can be extended and the *obj*-types that cannot. Only *pro*-types can have new members added or existing ones overriden. A *pro*-type can be *sealed* to make it a non-updatable *obj*-type.

In our type system we use recursive types (instead of row types plus universal and existential quantification). Note that Bono and Fisher's aim was to encode classes in their object calculus.

Also related is work done by Liquori [45, 44] on extensible objects. Liquori's system is based on matching [19] and allows object extension that is compatible with subsumption. Objects are created from prototypes, and extended with new functionality by adding new methods. The matching relation formalises the notion of an object inheriting from a prototype and specialising its behaviour.

Bruce and Gent

An imperative, type safe object oriented language, PolyTOIL, was introduced in [20]. Even though the language is class based, its type system does not identify types with classes. This makes the definition of types similar to ours. PolyTOIL, however, does not have extensible objects, so there is no need for identifying potential members.

Rémy and Vouillon

In [62] Rémy and Vouillon extend ML with objects and classes. Objects can be created directly or through classes, and there is support for multiple inheritance. The type system supports object types with recursion: $\texttt{rec}\,\alpha. < \mathtt{m_j} : \tau_{\mathtt{j}} .... >$. The congruence rules for types are based on those in [8]. Hence, there is a congruence rule similar to our (fix) rule. There is a distinction between the fields and methods of an object type; fields are not present in an object type. There is a notion of explicit subtyping that allows one value to be coerced from one type to another. As with our subtyping for object types, it is possible to write functions that work on any object type that has a particular method.

Morrisett et al.

In the context of typed assembly languages, Morrisett et al., in [53], use an initialisation flag on the members of a type to indicate if they have been assigned to. One could think of the potential and definite annotations of our types as representing the state of their initialisation flag.

4.5 Discussion

We have taken our subset of JavaScript, JS_0, and developed a static type system. The main challenges arose from handling the evolution of objects. By using structural types with structural subtyping, we were able capture the notion of functions working with many different kinds of objects. By classifying members of a type into potential and definite, we could locally track how an object evolves. In order to achieve a manageable system, we took the design decision to only track the receiver and parameter within the scope of a function body.

While developing the type system, we always kept in mind the need to develop a type inference algorithm. Our experience in [9, 26] with more advanced type systems showed the difficulties in inferring types in these more complex *flow aware* systems. In the next chapter we shall develop a type inference system for our type system.

5 Type Inference for JS$_0$

IN THIS CHAPTER we describe a type inference system for JS$_0$ based on a system of constraints between type variables. Type variables are used to represent the type of an expression. From a JS$_0$ program we generate a set of type variables with constraints between them. Constraints represent the relationships between types in the program, *e.g.* that the actual parameter to a function call should be a subtype of the formal parameter. If there is solution to a set constraints we say that they are satisfiable. A solution to a set of constraints is a mapping from type variables to types.

We show that the constraint system is sound with respect to the type system. At the expression level, this means that the type given to an expression by the type system is a subtype of the one given by a solution to the constraints. At the program level, this means that a solution to the constraints for a program can be used to generate a well-typed version of the program.

Once the constraints for a program have been generated a process of closure is undertaken. Constraint closure simplifies the extraction of a solution, if one exists, from a set of constraints. The process of closure adds new constraints and is finished when no new constraints are added. We show that a set of constraints and its closure have the same set of solutions.

We define well-formedness of a set of constraints. For a set of constraints to be well-formed, it must be closed and all the constraints must be well-formed. Intuitively, the well-formedness rules express solution satisfaction in the domain of the constraints.

We show that a well-formed set of constraints can be used to generate a solution to a set of constraints. We achieve this by defining a translation relation from type variables to types. We show that the translation relation is deterministic.

We conjecture that a solution generated by the translation relation from a well-formed set of constraints will satisfy those constraints. We have partially proven this, but the proof requires the use of two unproven conjectures. We outline our

$$
\begin{array}{lll}
\mathbb{e} \in \mathit{LabExp} & ::= & \mathbb{var} \mid \mathtt{f} \mid \mathtt{new}\,\mathtt{f}(\mathbb{e}) \mid \mathbb{e}\mathtt{;}\,\mathbb{e} \mid \mathbb{e}.\mathtt{m}(\mathbb{e}) \mid \\
 & & \mathbb{e}.\mathtt{m} \mid \mathtt{f}(\mathbb{e}) \mid \mathbb{lhs} = \mathbb{e} \mid \mathbb{e}_1\,?\,\mathbb{e}_2 : \mathbb{e}_3 \mid \mathtt{null} \mid \mathtt{n} \mid \mathbb{ie} \\
\mathbb{var} \in \mathit{LabEnvVars} & ::= & \mathtt{this\_l} \mid \mathtt{x\_l} \\
\mathbb{lhs} \in \mathit{LeftSide} & ::= & \mathtt{x\_l} \mid \mathbb{e}.\mathtt{m} \\
\mathbb{ie} \in \mathit{InferExp} & ::= & \mathtt{ret\_f} \mid \mathtt{call\_this\_l} \mid \mathtt{call\_x\_l} \mid \mathtt{call\_ret\_l} \\
 & & \\
\mathtt{l} \in \mathit{Lab} & ::= & \mathtt{1} \mid \mathtt{2} \mid \ldots \mid \mathtt{f} \mid \mathtt{f}' \mid \ldots
\end{array}
$$

Figure 5.1.: Syntax of Labeled Expressions

attempts at proving these conjectures in Appendix B. We also outline an alternative approach that does not use these conjectures.

5.1 Type Variables

As in [56, 7, 57, 55], we use type variables to express the - yet unknown - types of expressions. For example, returning to Chapter 3, see Figure 3.2, type variable ⟦new Date(1000)⟧ expresses the type of expression `new Date(1000)`.

Because the types of `this` and x differ for different occurrences in the same method body, we use labels to distinguish them, for example, ⟦this_1⟧, ⟦x_2⟧, ⟦this_3⟧... Labeled type variables ⟦this_f⟧ and ⟦x_f⟧ represent the type of `this` and x at the beginning of the function f, and ⟦ret_f⟧ represents the return type of the function f.

We generate a new label for each method call which is used to generate three type variables. These variables denote the type of the receiver, parameter and return type of the method. For example, for `x.add(y)` we could use label 5 which would generate type variables ⟦call_this_5⟧, ⟦call_x_5⟧ and ⟦call_ret_5⟧. Note that these type variables depend on the label but not on the name of the method. Figure 5.1 gives the syntax of labeled expressions and Figure 5.2 gives the syntax of type variables.

5.2 Constraints and Solutions

Constraints between type variables express the relationship between the type of expressions, *i.e.* which members a type must have, how the members of two types may differ, and whether a type has any definite members. The syntax of constraints is given in Figure 5.2. There are three kinds of constraint: $\tau \ \leq \ \rho$, $\tau \ \lhd \ \tau$ and $\tau^{\Box}$. We use c to range over constraints and C for a set of constraints.

A solution, S, to a set of constraints is a mapping from type variables to types. We now look at part of a solution, S_0, for the constraints that would be generated for the `Date` example in Figure 3.2 (where $\mathsf{t}_2 = \mu\,\alpha.[\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\alpha \times \alpha \rightarrow \alpha), \bullet)]$):

$$\begin{aligned}
\mathsf{S}_0([\![\mathtt{this\_Date}]\!]) &= [\mathtt{mSec} : (\mathsf{Int}, \circ),\ \mathtt{add} : ((\mathsf{t}_2 \times \mathsf{t}_2 \rightarrow \mathsf{t}_2), \circ)] \\
\mathsf{S}_0([\![\mathtt{this\_1}]\!]) &= [\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\mathsf{t}_2 \times \mathsf{t}_2 \rightarrow \mathsf{t}_2), \circ)] \\
\mathsf{S}_0([\![\mathtt{ret\_Date}]\!]) &= \mathsf{t}_2 \\
\mathsf{S}_0([\![\mathtt{x\_Date}]\!]) &= \mathsf{Int} \\
\mathsf{S}_0([\![\mathtt{this\_Date.mSec}]\!]) &= \mathsf{Int} \\
\mathsf{S}_0([\![\mathtt{this\_5}]\!]) &= [\mathtt{mSec} : (\mathsf{Int}, \bullet)]
\end{aligned}$$

The satisfaction of a constraint with respect to a solution is shown in Figure 5.3. Rule $(solSat)$ defines that S satisfies a set of constraints, $\mathsf{S} \vdash \mathsf{C}$, if it satisfies each constraint. We now discuss each kind of constraint and how it is satisfied by a solution.

- $\tau \ \leq \ \rho$ - requires a type variable to be a subtype of ρ: thus, $\tau \leq \mathsf{Int}$ requires τ to be `Int`, *c.f.* rule $(solInt)$; while $\tau \ \leq \ \tau'$ requires τ to be a subtype of τ', *c.f.* $(solSub)$; while $\tau \ \leq \ (\tau_1 \times \tau_2 \rightarrow \tau_3)$ requires τ to be the function type composed from τ_1, τ_2 and τ_3, *c.f.* $(solSubFunc)$; finally, $\tau \ \leq \ [\mathsf{m} : (\tau', \psi)]$ requires τ to have a member m of type τ' with annotation at least ψ, *c.f.* $(solMemChange)$. For example:

$$\begin{aligned}
&\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \ \leq \ [\![\mathtt{this\_Date}]\!] \\
&\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \ \leq \ [\![\mathtt{this\_5}]\!] \\
&\mathsf{S}_0 \vdash [\![\mathtt{this\_Date}]\!] \ \leq \ [\mathtt{mSec} : ([\![\mathtt{this\_Date.mSec}]\!], \circ)] \\
&\mathsf{S}_0 \not\vdash [\![\mathtt{this\_Date}]\!] \ \leq \ [\mathtt{mSec} : ([\![\mathtt{this\_Date.mSec}]\!], \bullet)]
\end{aligned}$$

- $\tau \ \lhd \ \tau'$ - there are two forms of this constraint depending on whether $\lhd$ is $\lhd_{\mathsf{m}}$ or $\lhd_*$. With $\tau \ \lhd_{\mathsf{m}} \ \tau'$ we require τ and τ' to have the same members with the same types, but member m can be potential in τ' but must be definite in

Type Variables

$$\tau \ ::= \ [\![\mathrm{e}]\!]$$

Constraints

$$\begin{array}{rcl}
\rho \in \textit{ConstRhs} & ::= & \tau \mid \sigma \mid [\mathsf{m} : (\tau, \psi)] \\
\sigma \in \textit{FuncInt} & ::= & (\tau \times \tau) \to \tau \mid \mathsf{Int} \\
\\
\mathsf{c} \in \textit{Const} & ::= & \tau \leq \rho \mid \tau \lhd \tau \mid \tau^{\square} \\
\square & ::= & \circ \mid \mathbf{obj} \\
\lhd & ::= & \lhd_{\mathsf{m}} \mid \lhd_{*} \\
\mathsf{C} & \in & \mathcal{P}(\textit{Const})
\end{array}$$

Figure 5.2.: Syntax of Type Variables and Constraints

τ, *c.f.* rule $(solMemChange)$. For example:

$$\mathsf{S}_0 \vdash [\![\texttt{this\_1}]\!] \lhd_{\texttt{mSec}} [\![\texttt{this\_Date}]\!]$$
$$\mathsf{S}_0 \not\vdash [\![\texttt{this\_Date}]\!] \lhd_{\texttt{mSec}} [\![\texttt{this\_1}]\!]$$
$$\mathsf{S}_0 \not\vdash [\![\texttt{this\_1}]\!] \lhd_{\texttt{mSec}} [\![\texttt{this\_5}]\!]$$

Note that we have $\mathsf{S}_0 \vdash [\![\texttt{this\_1}]\!] \leq [\![\texttt{this\_5}]\!]$; this should clarify the difference between the two kinds of constraint.

With $\tau \lhd_* \tau'$ we require that τ is an extension (see Definition 2) of τ', *c.f.* rule $(solAnyMemsChange)$.

- $\tau^{\square}$ - there are two forms of this constraint depending on whether $\square$ is $\circ$ or **obj**. With τ° we require τ to have no definite members, *c.f.* rule $(solNoDefs)$. This is needed for constructors and global functions whose receiver must have no definite members. For example, $\mathsf{S}_0 \vdash [\![\texttt{this\_Date}]\!]^{\circ}$.

  With $\tau^{\mathbf{obj}}$ we require that τ is an object type, *c.f.* rule $(solObjType)$. For example, we require the receiver in a function to be of an object type, and that `null` is given an object type.

5.3 Constraint Generation

Constraint generation for a JS$_0$ program produces a set of constraints between type variables. A *pre-environment*, $\gamma = \{\texttt{this} : \texttt{l},\ \texttt{x} : \texttt{l}',\ \texttt{lab} : \texttt{L}\}$, keeps track of

$$\frac{\begin{array}{l}\mathsf{S}(\tau) = \mathsf{t} \implies \vdash \mathsf{t} \diamond \\ \mathsf{C} = \{\mathsf{c}_1 ... \mathsf{c}_n\} \\ \mathsf{S} \vdash \mathsf{c}_i \ \forall \, i \ \in \ 1...n\end{array}}{\mathsf{S} \vdash \mathsf{C}} \ (solSat)$$

$$\frac{\mathsf{S}(\tau) \leq \mathsf{S}(\tau')}{\mathsf{S} \vdash \tau \ \leq \ \tau'} \ (solSub) \qquad \frac{\mathsf{S}(\tau) \leq (\mathsf{S}(\tau_1) \times \mathsf{S}(\tau_2) \rightarrow \mathsf{S}(\tau_3))}{\mathsf{S} \vdash \tau \ \leq \ (\tau_1 \times \tau_2 \rightarrow \tau_3)} \ (solSubFunc)$$

$$\frac{\mathsf{S}(\tau) = \mathsf{Int}}{\mathsf{S} \vdash \tau \ \leq \ \mathsf{Int}} \ (solInt) \qquad \frac{\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau'), \psi)}{\mathsf{S} \vdash \tau \ \leq \ [\mathsf{m} : (\tau', \psi)]} \ (solMember)$$

$$\frac{\begin{array}{l}\forall \, \mathsf{m}' \neq \mathsf{m} \ : \ \mathsf{S}(\tau)(\mathsf{m}') \equiv \ \mathsf{S}(\tau')(\mathsf{m}') \\ \mathsf{S}(\tau)(\mathsf{m}) \leq \mathsf{S}(\tau')(\mathsf{m})\end{array}}{\mathsf{S} \vdash \tau \ \lhd_{\mathsf{m}} \tau'} \ (solMemChange)$$

$$\frac{\mathsf{S}(\tau) \ \sqsubseteq \mathsf{S}(\tau')}{\mathsf{S} \vdash \tau \ \lhd_{*} \ \tau'} \ (solAnyMemsChange) \qquad \frac{\{\mathsf{m} \mid \mathsf{S}(\tau)(\mathsf{m}) = (\mathsf{t}, \bullet)\} = \emptyset}{\mathsf{S} \vdash \tau^{\circ}} \ (solNoDefs)$$

$$\frac{\mathsf{S}(\tau) = \mathsf{O} \quad \text{(for some object type O)}}{\mathsf{S} \vdash \tau^{\mathbf{obj}}} \ (solObjType)$$

Figure 5.3.: Solution Satisfaction

the current labeling of `this` and x along with the set of labels used so far (stored in the set L). Constraint generation for an expression e in the context of a pre-environment, γ, has the form:

$$\gamma \vdash \mathsf{e} \,:\, \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C}$$

where γ' reflects the changes to the labeling of `this`, x and `lab` while generating constraints, and $\mathbb{e}$ is a labelled version of e for use in type variables. The constraints generated for an expression consist of the union of the constraints for each subexpression augmented by local constraints. The constraint generation rules for expressions are shown in Figures 5.4 and 5.5.

In (var) we generate a labeled expression for `this` and x by looking in the pre-environment for the current label. In the case of `this` and `null` we generate constraint $[\![\mathtt{this\_l}]\!]^{\mathbf{obj}}$ and $[\![\mathtt{null}]\!]^{\mathbf{obj}}$ respectively to ensure that they are object types.

In $(funcId)$ we require f to have a function type derived from the type of the receiver, parameter and return value of the function. The type variables come from the initial labeled `this` and x, and the labeled return variable `ret_f`. For example, function identifier `addFn` produces constraint:

$$[\![\mathtt{addFn}]\!] \;\leq\; ([\![\mathtt{this\_addFn}]\!] \times [\![\mathtt{x\_addFn}]\!] \rightarrow [\![\mathtt{ret\_addFn}]\!])$$

In $(assignAdd)$ we model the update or possible addition of member m of var. Note that we use var to represent `this` or x. $\mathsf{var\_l}$ and $\mathsf{var\_l'}$ represent the type of var *before* and *after* the update (where $\mathtt{l'}$ is fresh). Constraint $[\![\mathsf{var\_l}]\!] \;\leq\; [\mathsf{m} : ([\![\mathsf{var\_l.m}]\!], \circ)]$ requires var to have member m with annotation at least $\circ$ before the update, and constraint $[\![\mathsf{var\_l'}]\!] \;\leq\; [\mathsf{m} : ([\![\mathsf{var\_l'.m}]\!], \bullet)]$ requires var to have member m with annotation definite, $\bullet$, after the update. Using $[\![\mathsf{var\_l}]\!] \;\leq\; [\mathsf{m} : ([\![\mathsf{e}]\!], \_)]$ instead of $[\![\mathsf{var\_l}]\!] \;\leq\; [\mathsf{m} : ([\![\mathsf{var\_l.m}]\!], \_))]$ would have been too restrictive. Namely, a solution would require the type of m to be the same as the type of $[\![\mathsf{e}]\!]$, rather than a supertype. Intuitively, by using $[\![\mathsf{var\_l.m}]\!]$ a solution can give a type for a member that is the least upper bound of all the assignments to that member.

The constraint $[\![\mathsf{var\_l'}]\!] \;\lhd_{\mathsf{m}}\; [\![\mathsf{var\_l}]\!]$ requires that only member m is affected by the assignment. The remaining constraints require that the type of member m, $[\![\mathsf{var\_l'.m}]\!]$, and the overall expression have the type of the right hand side of the assignment. For example, `this.add = addFn` in a pre-environment $\gamma_2 = \{\texttt{this} : 1, \texttt{lab} : \texttt{L}, ...\}$, where $2 \notin \texttt{L}$, generates constraints:

$$\begin{aligned}
[\![\texttt{this\_1}]\!] &\leq [\texttt{add} : ([\![\texttt{this\_1.add}]\!], \circ)] \\
[\![\texttt{this\_2}]\!] &\leq [\texttt{add} : ([\![\texttt{this\_2.add}]\!], \bullet)] \\
[\![\texttt{this\_2}]\!] &\lhd_{\texttt{add}} [\![\texttt{this\_1}]\!] \\
[\![\texttt{addFn}]\!] &\leq [\![\texttt{this\_2.add}]\!] \\
[\![\texttt{addFn}]\!] &\leq [\![\texttt{this\_2.add} = \texttt{addFn}]\!]
\end{aligned}$$

and the post pre-environment $\gamma_2[\texttt{this} \mapsto 2, \texttt{lab} \mapsto \texttt{L} \cup \{2\}]$.

In (new) we model a function being used to create an object. The constraint $[\![\texttt{this\_f}]\!]^\circ$ requires the initial type for `this` in function f to have no definite members. The constraint $[\![\texttt{e}]\!] \leq [\![\texttt{x\_f}]\!]$ requires the actual parameter to have a subtype of the formal parameter where x_f is the initial type of the formal parameter. The constraint $[\![\texttt{new f(e)}]\!] \leq [\![\texttt{ret\_f}]\!]$ requires the overall type of the `new` expression to be a subtype of the return type of the function. For example, expression `new Date(1000)` generates constraints:

$$\begin{aligned}
&[\![\texttt{this\_Date}]\!]^\circ \\
&[\![\texttt{1000}]\!] \leq [\![\texttt{x\_Date}]\!] \\
&[\![\texttt{newDate(1000)}]\!] \leq [\![\texttt{ret\_Date}]\!]
\end{aligned}$$

The rule for global function, $(funcCall)$, is similar in structure to the rule (new).

For member access, $(memAcc)$, and for assignment where the receiver is not `this` or `x`, $(assignUpd)$, the receiver must have the member with a definite annotation. For example, expression `x.mSec` in a pre-environment $\gamma_1 = \{\texttt{x} : 2, ...\}$ generates constraint:

$$[\![\texttt{x\_2}]\!] \leq [\texttt{mSec} : ([\![\texttt{x\_2.mSec}]\!], \bullet)]$$

For method call, $(methCall)$, we consider the label characterizing the occurrence of the call. For a call with label `l` we require the receiver to have a definite member, m, with function type $[\![\texttt{call\_this\_l}]\!] \times [\![\texttt{call\_x\_l}]\!] \to [\![\texttt{call\_ret\_l}]\!]$, as expressed through the constraint $[\![\texttt{e}_1]\!] \leq [\texttt{m} : ([\![\texttt{call\_this\_l}]\!] \times [\![\texttt{call\_x\_l}]\!] \to [\![\texttt{call\_ret\_l}]\!], \bullet)]$. Using Constraint $[\![\texttt{e}_1]\!] \leq ([\![\texttt{e}_1]\!] \times [\![\texttt{e}_2]\!] \to [\![\texttt{e}_1\texttt{.m(e}_2\texttt{)}]\!])$ would have been too restrictive. Namely, it would require all the receivers of the method to have the same type. By using new type variables, we ensure that a solution to the constraints will give a type to the method that is the least upper bound of all the receivers, parameters and return types at the call sites. For example, `x.add(y)` in a pre-environment $\gamma_3 = \{\mathtt{x} : \mathtt{Main}, \mathtt{y} : \mathtt{Main}, \mathtt{lab} : \mathtt{L}, ...\}$, where $5 \notin \mathtt{L}$, generates constraints:

$$\begin{array}{lcl}
[\![\mathtt{x\_Main}]\!] & \leq & [\mathtt{add} : ([\![\mathtt{x\_Main.add}]\!], \bullet)] \\
[\![\mathtt{x\_Main.add}]\!] & \leq & ([\![\mathtt{call\_this\_5}]\!] \times [\![\mathtt{call\_x\_5}]\!] \to [\![\mathtt{call\_ret\_5}]\!]) \\
[\![\mathtt{x\_Main}]\!] & \leq & [\![\mathtt{call\_this\_5}]\!] \\
[\![\mathtt{y\_Main}]\!] & \leq & [\![\mathtt{call\_x\_5}]\!] \\
[\![\mathtt{call\_ret\_5}]\!] & \leq & [\![\mathtt{x\_Main.add(y\_Main)}]\!]
\end{array}$$

and the post pre-environment $\gamma_3[\mathtt{lab} \mapsto \mathtt{L} \cup \{5\}]$.

It is possible to optimize the creation of new variables, by sharing, at the call site. Refer to Section 5.10 where we discuss [71] which shows possible optimizations.

For programs, $(prog)$, see Figure 5.6, we collect the constraints generated for each function with a pre-environment mapping `this` and `x` to their respective initial labelings.

We now look at some of the constraints that would be generated for the web example in Chapter 3 (Figure 3.5). We only give a selection[1] of the constraints generated; the full set is shown in Chapter 6.

1. With pre-environment $\gamma_1 = \{\mathtt{this} : \mathtt{input}, \mathtt{lab} : \mathtt{L}_1, ...\}$, where $1,2 \notin \mathtt{L}_1$, constraint generation for function `input` will produce constraints:

$$\begin{array}{lcl}
\mathsf{C}_1 & = & \{\ [\![\mathtt{this\_input}]\!] \leq [\mathtt{value} : ([\![\mathtt{this\_1.value}]\!], \circ)], \\
& & \quad [\![\mathtt{this\_1}]\!] \leq [\mathtt{value} : ([\![\mathtt{this\_1.value}]\!], \bullet)], \\
& & \quad [\![\mathtt{this\_1}]\!] \leq [\mathtt{disabled} : ([\![\mathtt{this\_1.disabled}]\!], \circ)], \\
& & \quad [\![\mathtt{this\_2}]\!] \leq [\mathtt{disabled} : ([\![\mathtt{this\_2.disabled}]\!], \bullet)], \\
& & \quad [\![\mathtt{this\_1}]\!] \lhd_{\mathtt{value}} [\![\mathtt{this\_input}]\!] \\
& & \quad [\![\mathtt{this\_2}]\!] \lhd_{\mathtt{disabled}} [\![\mathtt{this\_1}]\!] \\
& & \quad ...\}
\end{array}$$

2. With pre-environment $\gamma_2 = \{\mathtt{this} : \mathtt{form}, \mathtt{lab} : \mathtt{L}_2, ...\}$, where $3 \notin \mathtt{L}_2$, con-

[1]The constraints we have shown are pertinent to an example we will give later in this chapter.

$$\frac{}{\begin{array}{l}\gamma \vdash \texttt{null} : \texttt{null} \,||\, \gamma \,||\, \{[\![\texttt{null}]\!]^{\texttt{obj}}\} \\ \gamma \vdash \texttt{n} : \texttt{n} \,||\, \gamma \,||\, \{[\![\texttt{n}]\!] \leq \mathsf{Int}\} \\ \gamma \vdash \texttt{this} : \texttt{this\_}\gamma(\texttt{this}) \,||\, \gamma \,||\, \{[\![\texttt{this\_l}]\!]^{\texttt{obj}}\} \\ \gamma \vdash \texttt{x} : \texttt{x\_}\gamma(\texttt{x}) \,||\, \gamma \,||\, \emptyset\end{array}}\;(var)$$

$$\frac{\mathsf{C} = \{[\![\mathsf{f}]\!] \leq ([\![\texttt{this\_f}]\!] \times [\![\texttt{x\_f}]\!]) \rightarrow [\![\texttt{ret\_f}]\!]\}}{\gamma \vdash \mathsf{f} : \mathsf{f} \,||\, \gamma \,||\, \mathsf{C}}\;(funcId)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e} : \mathbb{e} \,||\, \gamma'' \,||\, \mathsf{C}' \\ \gamma''(\mathsf{var}) = \texttt{l} \\ \texttt{l}' \notin \gamma''(\texttt{lab}) \\ \gamma' = \gamma''[\mathsf{var} \mapsto \texttt{l}', \texttt{lab} \mapsto (\gamma''(\texttt{lab}) \cup \{\texttt{l}'\})\,] \\ \mathsf{var} = \mathsf{e}' \text{ is not a subexpression of } \mathsf{e}_2 \\ \begin{array}{rcl}\mathsf{C} & = & \{[\![\mathsf{var\_l}]\!] \leq [\mathsf{m} : ([\![\mathsf{var\_l.m}]\!], \circ)], \\ & & [\![\mathsf{var\_l}']\!] \leq [\mathsf{m} : ([\![\mathsf{var\_l}'.\mathsf{m}]\!], \bullet)], \\ & & [\![\mathsf{var\_l}']\!] \lhd_{\mathsf{m}} [\![\mathsf{var\_l}]\!], \\ & & [\![\mathbb{e}]\!] \leq [\![\mathsf{var\_l}'.\mathsf{m}]\!], \\ & & [\![\mathbb{e}]\!] \leq [\![\mathsf{var\_l}'.\mathsf{m} = \mathbb{e}]\!]\}\end{array}\end{array}}{\gamma \vdash \mathsf{var.m} = \mathsf{e} : \mathsf{var\_l}'.\mathsf{m} = \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C} \cup \mathsf{C}'}\;(assignAdd)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e} : \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C}' \\ \begin{array}{rcl}\mathsf{C} & = & \{[\![\texttt{this\_f}]\!]^{\circ},\; [\![\mathbb{e}]\!] \leq [\![\texttt{x\_f}]\!] \\ & & [\![\texttt{ret\_f}]\!] \leq [\![\texttt{new}\,\mathsf{f}(\mathbb{e})]\!]\}\end{array}\end{array}}{\gamma \vdash \texttt{new}\,\mathsf{f}(\mathsf{e}) : \texttt{new}\,\mathsf{f}(\mathbb{e}) \,||\, \gamma' \,||\, \mathsf{C} \cup \mathsf{C}'}\;(new)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e} : \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C}' \\ \begin{array}{rcl}\mathsf{C} & = & \{[\![\texttt{this\_f}]\!]^{\circ},\; [\![\mathbb{e}]\!] \leq [\![\texttt{x\_f}]\!] \\ & & [\![\texttt{ret\_f}]\!] \leq [\![\mathsf{f}(\mathbb{e})]\!]\}\end{array}\end{array}}{\gamma \vdash \mathsf{f}(\mathsf{e}) : \mathsf{f}(\mathbb{e}) \,||\, \gamma' \,||\, \mathsf{C} \cup \mathsf{C}'}\;(funcCall)$$

$$\frac{\gamma \vdash \mathsf{e} : \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C}'}{\gamma \vdash \mathsf{e.m} : \mathbb{e}.\mathsf{m} \,||\, \gamma' \,||\, \mathsf{C} \cup \{[\![\mathbb{e}]\!] \leq [\mathsf{m} : ([\![\mathbb{e}.\mathsf{m}]\!], \bullet)]\}}\;(memAcc)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e}_1 : \mathbb{e}_1 \,||\, \gamma' \,||\, \mathsf{C}' \\ \gamma' \vdash \mathsf{e}_2 : \mathbb{e}_2 \,||\, \gamma'' \,||\, \mathsf{C}'' \\ \begin{array}{rcl}\mathsf{C} & = & \{[\![\mathbb{e}_1]\!] \leq [\mathsf{m} : ([\![\mathbb{e}_1.\mathsf{m}]\!], \bullet)], \\ & & [\![\mathbb{e}_2]\!] \leq [\![\mathbb{e}_1.\mathsf{m}]\!], \\ & & [\![\mathbb{e}_2]\!] \leq [\![\mathbb{e}_1.\mathsf{m} = \mathbb{e}_2]\!]\}\end{array}\end{array}}{\gamma \vdash \mathsf{e}_1.\mathsf{m} = \mathsf{e}_2 : \mathbb{e}_1.\mathsf{m} = \mathbb{e}_2 \,||\, \gamma'' \,||\, \mathsf{C} \cup \mathsf{C}' \cup \mathsf{C}''}\;(assignUpd)$$

Figure 5.4.: Constraint Generation for Expressions

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e}_1 \ :\ \mathbb{e}_1 \,||\, \gamma' \,||\, \mathsf{C}' \\ \gamma' \vdash \mathsf{e}_2 \ :\ \mathbb{e}_2 \,||\, \gamma'' \,||\, \mathsf{C}'' \\ \mathtt{l} \notin \gamma''(\mathtt{lab}) \\ \ \mathsf{C} \ = \ \{\llbracket \mathbb{e}_1 \rrbracket \ \leq\ [\mathsf{m} : (\llbracket \mathbb{e}_1.\mathsf{m} \rrbracket, \bullet)], \\ \qquad\quad \llbracket \mathbb{e}_1.\mathsf{m} \rrbracket \ \leq\ ((\llbracket \mathtt{call\_this\_l} \rrbracket \times \llbracket \mathtt{call\_x\_l} \rrbracket) \rightarrow \llbracket \mathtt{call\_ret\_l} \rrbracket), \\ \qquad\quad \llbracket \mathbb{e}_1 \rrbracket \ \leq\ \llbracket \mathtt{call\_this\_l} \rrbracket, \\ \qquad\quad \llbracket \mathbb{e}_2 \rrbracket \ \leq\ \llbracket \mathtt{call\_x\_l} \rrbracket, \\ \qquad\quad \llbracket \mathtt{call\_ret\_l} \rrbracket \ \leq\ \llbracket \mathbb{e}_1.\mathsf{m}(\mathbb{e}_2) \rrbracket\} \end{array}}{\gamma \vdash \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2) \ :\ \mathbb{e}_1.\mathsf{m}(\mathbb{e}_2) \,||\, \gamma''[\mathtt{lab} \mapsto (\gamma''(\mathtt{lab}) \cup \{\mathtt{l}\}] \,||\, \mathsf{C} \cup \mathsf{C}' \cup \mathsf{C}''} \ (methCall)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e}_1 \ :\ \mathbb{e}_1 \,||\, \gamma' \,||\, \mathsf{C}' \\ \gamma' \vdash \mathsf{e}_2 \ :\ \mathbb{e}_2 \,||\, \gamma'' \,||\, \mathsf{C}'' \\ \ \mathsf{C} \ = \ \{\llbracket \mathbb{e}_2 \rrbracket \ \leq\ \llbracket \mathbb{e}_1 ; \mathbb{e}_2 \rrbracket\} \end{array}}{\gamma \vdash \mathsf{e}_1 ; \mathsf{e}_2 \ :\ \mathbb{e}_1 ; \mathbb{e}_2 \,||\, \gamma'' \,||\, \mathsf{C} \cup \mathsf{C}' \cup \mathsf{C}''} \ (seq)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e} \ :\ \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C}' \\ \gamma'(\mathtt{x}) = \mathtt{l} \\ \ \mathsf{C} \ = \ \{\llbracket \mathbb{e} \rrbracket \ \leq\ \llbracket \mathtt{x\_l} \rrbracket,\ \llbracket \mathbb{e} \rrbracket \ \leq\ \llbracket \mathtt{x\_l} = \mathbb{e} \rrbracket\} \end{array}}{\gamma \vdash \mathtt{x} \ = \mathsf{e} \ :\ \mathtt{x\_l} = \mathbb{e} \,||\, \gamma' \,||\, \mathsf{C} \cup \mathsf{C}'} \ (varAss)$$

$$\frac{\begin{array}{l}\gamma \vdash \mathsf{e}_1 \ :\ \mathbb{e}_1 \,||\, \gamma_1 \,||\, \mathsf{C}_1 \\ \gamma_1 \vdash \mathsf{e}_2 \ :\ \mathbb{e}_2 \,||\, \gamma_2 \,||\, \mathsf{C}_2 \\ \gamma_2 \vdash \mathsf{e}_3 \ :\ \mathbb{e}_3 \,||\, \gamma_3 \,||\, \mathsf{C}_3 \\ \gamma_2(\mathtt{this}) = \mathtt{l}_2,\ \gamma_3(\mathtt{this}) = \mathtt{l}_3,\ \gamma_2(\mathtt{x}) = \mathtt{l}_2',\ \gamma_3(\mathtt{x}) = \mathtt{l}_3' \\ \mathtt{l} \notin \gamma_3(\mathtt{lab}) \quad (\text{fresh } \mathtt{l}) \\ \gamma'' = \{\mathtt{this} \mapsto \mathtt{l},\ \mathtt{x} \mapsto \mathtt{l},\ \mathtt{lab} \mapsto (\gamma_3(\mathtt{lab}) \cup \{\mathtt{l}\})\} \\ \ \mathsf{C} \ = \ \{\llbracket \mathbb{e}_1 \rrbracket \ \leq\ \mathsf{Int} \\ \qquad\quad \llbracket \mathtt{this\_l_2} \rrbracket \ \lhd_*\ \llbracket \mathtt{this\_l} \rrbracket,\ \llbracket \mathtt{this\_l_3} \rrbracket \ \lhd_*\ \llbracket \mathtt{this\_l} \rrbracket, \\ \qquad\quad \llbracket \mathtt{x\_l_2'} \rrbracket \ \lhd_*\ \llbracket \mathtt{x\_l} \rrbracket,\ \llbracket \mathtt{x\_l_3'} \rrbracket \ \lhd_*\ \llbracket \mathtt{x\_l} \rrbracket, \\ \qquad\quad \llbracket \mathbb{e}_2 \rrbracket \ \lhd_*\ \llbracket \mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket,\ \llbracket \mathbb{e}_3 \rrbracket \ \lhd_*\ \llbracket \mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket\} \end{array}}{\gamma \vdash \mathsf{e}_1 ?\, \mathsf{e}_2 :\ \mathsf{e}_3 \ :\ \mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3 \,||\, \gamma'' \,||\, \mathsf{C} \cup \mathsf{C}_1 \cup \mathsf{C}_2 \cup \mathsf{C}_3} \ (cond)$$

Figure 5.5.: Constraint Generation for Expressions Cont.

$$\frac{\begin{array}{l} \mathsf{P} = \texttt{function}\, \mathsf{f}_1(\texttt{x})\ \{\ \mathsf{e}_1\ \}\ \cdots\ \texttt{function}\, \mathsf{f}_n(\texttt{x})\ \{\ \mathsf{e}_n\ \} \\ \{\texttt{this}\ \mapsto\ \mathsf{f}_i,\ \texttt{x}\ \mapsto\ \mathsf{f}_i,\ \texttt{lab}\ \mapsto\ \gamma'_{i-1}(\texttt{lab})\} \vdash \mathsf{e}_i\ :\ \oplus_i \mid\mid \gamma'_i \mid\mid \mathsf{C}_i \quad (1 \le i \le n) \\ \gamma_0 = \emptyset \\ \mathsf{f} \in \mathsf{e}_i \implies \mathsf{P}(\mathsf{f}) \neq \mathcal{U}df \ \ (1 \le i \le n) \\ \mathsf{C} = \bigcup\limits_{i\,\in\,1..n} \mathsf{C}_i \cup \{[\![\oplus_\mathtt{i}]\!]\ \le\ [\![\mathtt{ret\_f_i}]\!]\} \end{array}}{\vdash \mathsf{P} :\ \mathsf{C}}\ (prog)$$

Figure 5.6.: Constraint Generation for Programs

straint generation for function `form` will produce constraints:

$$\begin{array}{lcl} \mathsf{C}_2 & = & \{\ \ [\![\mathtt{this\_form}]\!]\ \le\ [\mathtt{onSubmit} : ([\![\mathtt{this\_form.onSubmit}]\!], \circ)], \\ & & \ \ [\![\mathtt{this\_3}]\!]\ \le\ [\mathtt{onSubmit} : ([\![\mathtt{this\_3.onSubmit}]\!], \bullet)], \\ & & \ \ [\![\mathtt{this\_3}]\!]\ \lhd_{\mathtt{onSubmit}}\ [\![\mathtt{this\_form}]\!] \\ & & \ \ [\![\mathtt{onSubmit}]\!]\ \le\ [\![\mathtt{this\_onSubmit}]\!] \to [\![\mathtt{ret\_onSubmit}]\!] \\ & & \ \ ...\} \end{array}$$

3. With pre-environment $\gamma_3 = \{\mathtt{htmlform} : \mathtt{main}, \mathtt{lab} : \mathtt{L}_3, ...\}$, where $4 \notin \mathtt{L}_3$, constraint generation for the main body of code (which we have moved into a function `main`) will produce constraints:

$$\begin{array}{lcl} \mathsf{C}_3 & = & \{\ \ [\![\mathtt{new\ form()}]\!]\ \le\ [\![\mathtt{htmlform\_main}]\!] \\ & & \ \ [\![\mathtt{htmlform\_main}]\!]\ \le\ [\mathtt{submit} : ([\![\mathtt{htmlform\_main.submit}]\!], \circ)], \\ & & \ \ [\![\mathtt{htmlform\_4}]\!]\ \le\ [\mathtt{submit} : ([\![\mathtt{htmlform\_4.submit}]\!], \bullet)], \\ & & \ \ [\![\mathtt{htmlform\_4}]\!]\ \lhd_{\mathtt{submit}}\ [\![\mathtt{htmlform\_main}]\!] \\ & & \ \ [\![\mathtt{ret\_form}]\!]\ \le\ [\![\mathtt{new\ form()}]\!] \\ & & \ \ [\![\mathtt{this\_input}]\!]^\circ \\ & & \ \ [\![\mathtt{this\_form}]\!]^\circ ...\} \end{array}$$

Note that we have abused the notation slightly in that the formal parameter is called `htmlform` rather than `x`.

4. With pre-environment $\gamma_4 = \{\mathtt{theform} : \mathtt{checkform}, \mathtt{lab} : ......\}$ constraint generation for function `checkform` will produce constraints:

$$\begin{array}{lcl} \mathsf{C}_4 & = & \{\ \ [\![\mathtt{theform\_checkForm}]\!]\ \le\ [\mathtt{submit} : ([\![\mathtt{theform\_checkForm.submit}]\!], \bullet)], \\ & & \ \ ...\} \end{array}$$

Again, we have abused the notation slightly in that the formal parameter is called `theform` rather than x.

5. With pre-environment $\gamma_5 = \{\texttt{htmlform} : 7, \texttt{lab} : L_5, ...\}$ constraint generation for the method call `htmlform.onSubmit()`, in the `main` function, will produce constraints:

$$\begin{aligned} C_5 &= \{[\![\texttt{htmlform\_7}]\!] \leq [\texttt{onSubmit} : ([\![\texttt{htmlform\_7.onSubmit}]\!], \bullet)], \\ &\quad [\![\texttt{htmlform\_7.onSubmit}]\!] \leq [\![\texttt{call\_this\_1}]\!] \to [\![\texttt{call\_ret\_1}]\!] \\ &\quad [\![\texttt{htmlform\_7}]\!] \leq [\![\texttt{call\_this\_1}]\!] \\ &\quad ...\} \end{aligned}$$

Again, we have abused the notation by allowing a function with no parameters.

6. With pre-environment $\gamma_6 = \{\texttt{htmlform} : \texttt{onSumit}, ...\}$ constraint generation for function `onSubmit` will produce constraints:

$$\begin{aligned} C_6 &= \{[\![\texttt{this\_onSubmit}]\!] \leq [\![\texttt{x\_checkform}]\!] \\ &\quad [\![\texttt{this\_checkform}]\!]^{\circ} \\ &\quad ...\} \end{aligned}$$

5.4 Soundness of the Constraints

We now show that the constraints are sound with respect to the type system. We first give some auxiliary definitions that will be useful in the proof. Given a solution, S, and pre-environment, γ, we can generate an environment, Γ, as follows:

$$\Gamma_{\text{gen}}(\gamma, S) = \{\texttt{this} \mapsto S(\texttt{this\_}\gamma(\texttt{this})), \texttt{x} \mapsto S(\texttt{x\_}\gamma(\texttt{x}))\}$$

Judgement $S, \mathbb{P} \vdash \texttt{e}$, shown in Definition 4, expresses *function identifier satisfaction* of an expression with respect to a program and a solution. If a function identifier appears in an expression it should be present in the program with an annotation consistent with the solution. We say that f is present in an expression, $\texttt{f} \in \texttt{e}$, if f appears in e. For example: $\texttt{f}, \texttt{f}' \in \texttt{new}\ \texttt{f}(\texttt{f}')$.

With function $\mathcal{T}$, shown in Figure 5.7, we define the application of a solution to a program. Given a program, P, $\mathcal{T}$ produces the corresponding $\mathrm{JS}_0^{\mathsf{T}}$ program, $\mathbb{P}$, by using the solution to find the type of the formal parameter, receiver and return type of all the functions.

$$\mathcal{T}(\mathsf{f},\mathsf{S}) = \begin{cases} \texttt{function}\ \mathsf{f}(\texttt{x}) : \mathsf{G}\ \{\mathsf{e}\} & \text{if } \mathsf{P}(\mathsf{f}) = \texttt{function}\ \mathsf{f}(\texttt{x})\ \{\mathsf{e}\} \\ \quad \text{where } \mathsf{G} = (\mathsf{S}(\llbracket \texttt{this\_f} \rrbracket) \times \mathsf{S}(\llbracket \texttt{x\_f} \rrbracket) \to \mathsf{S}(\llbracket \texttt{ret\_f} \rrbracket)) & \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

$$\mathcal{T}(\mathsf{P},\mathsf{S}) = \mathcal{T}(\mathsf{f}_1,\mathsf{S})...\mathcal{T}(\mathsf{f}_n,\mathsf{S}) \quad \text{where } \mathsf{P} = \texttt{function}\ \mathsf{f}_1(\texttt{x})\ \{\mathsf{e}_1\}...\texttt{function}\ \mathsf{f}_n(\texttt{x})\ \{\mathsf{e}_\mathsf{n}\}$$

Figure 5.7.: Translating From JS_0 to JS_0^T

Definition 4 $\mathsf{S},\mathbb{P} \vdash \mathsf{e}$ *iff* $\mathsf{f} \in \mathsf{e} \implies \mathbb{P}(\mathsf{f}) = \texttt{function}\ \mathsf{f}(\texttt{x}) : (\mathsf{S}(\llbracket \texttt{this\_f} \rrbracket) \times \mathsf{S}(\llbracket \texttt{x\_f} \rrbracket) \to \mathsf{S}(\llbracket \texttt{ret\_f} \rrbracket))$.

5.4.1 Soundness of Constraints for Expressions

Theorem 2 asserts the soundness of the constraints at the expression level. The type given to an expression by the type system is a subtype of the one given by a solution to the constraints. The expression is type checked in a program generated using the solution: $\mathcal{T}(\mathsf{P},\mathsf{S})$. The environments used for type checking are those produced by Γ_{gen} with pre-environments γ and γ'.

Theorem 2 *For program* P*, expression* e*, pre-environment* γ*, solution* S*, if:*

$\gamma \vdash \mathsf{e} : ⓔ \mid\mid \gamma' \mid\mid \mathsf{C}$ *(1)*
$\mathsf{S} \vdash \mathsf{C}$ *(2)*
$\mathsf{S},\mathbb{P} \vdash \mathsf{e}$ *(3)*
$\mathbb{P} = \mathcal{T}(\mathsf{P},\mathsf{S})$ *(4)*
$\Gamma = \Gamma_{\mathsf{gen}}(\gamma,\mathsf{S})$ *(5)*
$\Gamma' = \Gamma_{\mathsf{gen}}(\gamma',\mathsf{S})$ *(6)*

then there exits a t *such that:*

$\mathbb{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \mid\mid \Gamma'$ *(7)*
$\mathsf{t} \leq \mathsf{S}(\llbracket ⓔ \rrbracket)$ *(8)*

Proof: We proceed by induction on the derivation $\gamma \vdash \mathsf{e} : ⓔ \mid\mid \gamma' \mid\mid \mathsf{C}$. We consider the interesting cases: (var), $(memAcc)$, $(methCall)$, $(assignAdd)$, (new) and $(cond)$.

Case: (var)

We consider the case when `e =null`. Given that the last rule applied was (var) we get from (1):

$$\gamma \vdash \texttt{null} \; : \; \texttt{null} \,||\, \gamma \,||\, \{ \, [\![\texttt{null}]\!]^{\mathbf{obj}} \} \tag{9}$$

From (2) and $[\![\texttt{null}]\!]$ with solution satisfaction rule $(solObjType)$ we get for some O:

$$\mathsf{S}([\![\texttt{null}]\!]) = \mathsf{O} \tag{10}$$

From type rule $(const)$ with (4) and O we get:

$$\mathbb{P}, \Gamma \vdash \texttt{null} \; : \mathsf{O} \,\|\, \Gamma \tag{11}$$

We see that (10) and (11) establish (8) and (7) respectively.

Case: $(memAcc)$

Given that the last rule applied was $(memAcc)$ we get that there exist suitable e', $\mathbb{e}'$, γ' and C' such that:

$$\mathbb{e} \text{ is } \mathbb{e}'.\mathsf{m}, \mathsf{e} \text{ is } \mathsf{e.m} \tag{12}$$

$$\gamma \vdash \mathsf{e}' \; : \; \mathbb{e}' \,||\, \gamma' \,||\, \mathsf{C}' \tag{13}$$

$$\mathsf{C} = \mathsf{C}' \cup \{[\![\mathbb{e}']\!] \; \leq \; [\mathsf{m} : ([\![\mathbb{e}'.\mathsf{m}]\!], \bullet)]\} \tag{14}$$

From (14) and (2) we get:

$$\mathsf{S} \vdash \mathsf{C}' \tag{15}$$

$$\mathsf{S} \vdash [\![\mathbb{e}']\!] \; \leq \; [\mathsf{m} : ([\![\mathbb{e}'.\mathsf{m}]\!], \bullet)] \tag{16}$$

From (16) and solution satisfaction rule $(solMember)$ we get:

$$\mathsf{S}([\![\mathbb{e}']\!])(\mathsf{m}) \leq (\mathsf{S}([\![\mathbb{e}'.\mathsf{m}]\!]), \bullet) \tag{17}$$

From (3) with Definition 4 we get:

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}' \tag{18}$$

From (13), (15), (18), (4), (5), and (6) we apply the inductive hypothesis to get that for some t:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}' : \mathsf{t} \mid\mid \Gamma' \tag{19}$$

$$\mathsf{t} \leq \mathsf{S}([\![\mathbb{e}']\!]) \tag{20}$$

From (20) and (17) with definition of subtyping we get for some t′:

$$\mathsf{t}(\mathsf{m}) = (\mathsf{t}', \bullet) \tag{21}$$

$$\mathsf{t}' \equiv \mathsf{S}([\![\mathbb{e}'.\mathsf{m}]\!]) \tag{22}$$

With (19) and (21) we apply type rule (*memAcc*), from Figure 4.11, to get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}'.\mathsf{m} : \mathsf{t}' \mid\mid \Gamma' \tag{23}$$

We see that (23) establishes (7).

From (22) with subtyping rule ($\leq$ *cong*) we get:

$$\mathsf{t}' \leq \mathsf{S}([\![\mathbb{e}'.\mathsf{m}]\!]) \tag{24}$$

We see that (24) and (12) establish (8).

Case: $(methCall)$

Given that the last rule applied was $(methCall)$ we get that there exist suitable $\mathsf{e}_1, \mathsf{e}_2, \mathbb{e}_1, \mathbb{e}_2, \gamma''', \gamma'', \mathsf{C}''$ and C' so that:

$$\mathbb{e} \text{ is } \mathbb{e}_1.\mathsf{m}(\mathbb{e}_2), \mathsf{e} \text{ is } \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2) \tag{25}$$

$$\gamma \vdash \mathsf{e}_1 \ : \ \mathbb{e}_1 \mid\mid \gamma''' \mid\mid \mathsf{C}' \tag{26}$$

$$\gamma''' \vdash \mathsf{e}_2 \ : \ \mathbb{e}_2 \mid\mid \gamma'' \mid\mid \mathsf{C}'' \tag{27}$$

$$\mathtt{l} \ \notin \ \gamma''(\mathtt{lab}) \tag{28}$$

$$\mathsf{C}''' = \{[\![\mathbb{e}_1]\!] \ \leq \ [\mathsf{m} : ([\![\mathbb{e}_1.\mathsf{m}]\!], \bullet)] \tag{29}$$

$$[\![\mathbb{e}_1.\mathsf{m}]\!] \ \leq \ (([\![\mathtt{call\_this\_l}]\!] \times [\![\mathtt{call\_x\_l}]\!]) \to [\![\mathtt{call\_ret\_l}]\!]), \tag{30}$$

$$[\![\mathbb{e}_1]\!] \ \leq \ [\![\mathtt{call\_this\_l}]\!] \tag{31}$$

$$[\![\mathbb{e}_2]\!] \ \leq \ [\![\mathtt{call\_x\_l}]\!] \tag{32}$$

$$[\![\mathtt{call\_ret\_l}]\!] \ \leq \ [\![\mathbb{e}_1.\mathsf{m}(\mathbb{e}_2)]\!]\} \tag{33}$$

$$\gamma' = \gamma''[\mathtt{lab} \ \mapsto (\gamma''(\mathtt{lab}) \ \cup \ \{\mathtt{l}\}] \tag{34}$$

$$\mathsf{C} = \mathsf{C}' \cup \mathsf{C}'' \cup \mathsf{C}''' \tag{35}$$

From (2) and (35) we have that S is a solution to $C' \cup C'' \cup C'''$ and so trivially:

$$S \vdash C' \tag{36}$$
$$S \vdash C'' \tag{37}$$
$$S \vdash C''' \tag{38}$$

From (38) and (29) with satisfaction rule $(solMember)$, we have that S ($[\![\mathbb{e}_1]\!]$) must be an object type:

$$S([\![\mathbb{e}_1]\!])(m) \leq (S([\![\mathbb{e}_1.m]\!]), \bullet) \tag{39}$$

Let:

$$\Gamma'' = \Gamma_{gen}(\gamma''', S) \tag{40}$$

From (3) with Definition 4 we get:

$$S, \mathbb{P} \vdash e_1 \tag{41}$$

From (26), (36), (41), (4), (5) and (40) we apply the inductive hypothesis to get that for some O:

$$\mathbb{P}, \Gamma \vdash e_1 : O \parallel \Gamma'' \tag{42}$$
$$O \leq S([\![e_1]\!]) \tag{43}$$

Furthermore, from the definition of subtyping with (43) and (39) we get for some t'':

$$O(m) = (t'', \bullet) \tag{44}$$
$$t'' \equiv S([\![\mathbb{e}_1.m]\!]) \tag{45}$$

From (38) with (30) and the definition of satisfaction, we have that $S([\![\mathbb{e}_1.m]\!])$ must be a function type; therefore, by congruence we get:

$$t''(\texttt{this}) \equiv S([\![\texttt{call\_this\_l}]\!]) \tag{46}$$
$$t''(\texttt{x}) \equiv S([\![\texttt{call\_x\_l}]\!]) \tag{47}$$
$$t''(\texttt{ret}) \equiv S([\![\texttt{call\_ret\_l}]\!]) \tag{48}$$

Let:

$$\Gamma' = \Gamma_{\mathsf{gen}}(\gamma'', \mathsf{S}) \tag{49}$$

From (3) with Definition 4 we get:

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}_2 \tag{50}$$

From (27), (37), (50), (4), (40) and (49) we apply the inductive hypothesis to get for some t''':

$$\mathbb{P}, \Gamma'' \vdash \mathsf{e}_2 : \mathsf{t}''' \parallel \Gamma' \tag{51}$$

$$\mathsf{t}''' \leq \mathsf{S}(\llbracket \mathsf{e}_2 \rrbracket) \tag{52}$$

From (38) with (31) and definition of satisfaction we have $\mathsf{S}(\llbracket \mathbb{e}_1 \rrbracket) \leq \mathsf{S}(\llbracket \mathtt{call\_this\_l} \rrbracket)$. From (46), the commuativity of $\equiv$, and ($\leq$ *cong*) we get $\mathsf{S}(\llbracket \mathbb{e}_1 \rrbracket) \leq \mathsf{t}''(\mathtt{this})$. Therefore, with (43) we get $\mathsf{O} \leq \mathsf{t}''(\mathtt{this})$. By a similar argument with (52), (38), (32) and (47) we get $\mathsf{t}''' \leq \mathsf{t}''(\mathtt{x})$. Hence, with (42), (51) and (44) we apply rule (*methCall*) to get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2) : \mathsf{t}''(\mathtt{ret}) \parallel \Gamma' \tag{53}$$

We see that (53) establishes (7).

By similar argument as that used above with (48), (33) and (38) we get $\mathsf{t}''(\mathtt{ret}) \leq \mathsf{S}(\llbracket \mathbb{e}_1.\mathsf{m}(\mathbb{e}_2) \rrbracket)$ which with (25) establish (8). Finally, because the only difference between γ'' and γ' is the labels we get that $\Gamma' = \Gamma_{\mathsf{gen}}(\gamma'', \mathsf{S}) = \Gamma_{\mathsf{gen}}(\gamma', \mathsf{S})$.

Case: (*assignAdd*)

Given that the last rule applied was (*assignAdd*) we get that there exist suitable $\mathsf{e}', \mathbb{e}', \gamma''$ and C' so that:

$$\mathbb{e} \text{ is } \mathsf{var\_l'}.\mathsf{m} = \mathbb{e}', \mathsf{e} \text{ is } \mathsf{var}.\mathsf{m} = \mathsf{e}' \tag{54}$$

$$\gamma \vdash \mathsf{e}' : \mathbb{e}' \parallel \gamma'' \parallel \mathsf{C}' \tag{55}$$

$$\gamma''(\mathsf{var}) = \mathtt{l} \tag{56}$$

$$\mathtt{l}' \notin \gamma''(\mathtt{lab}) \tag{57}$$

$$\gamma' = \gamma''[\mathsf{var} \mapsto \mathtt{l}', \mathtt{lab} \mapsto (\gamma''(\mathtt{lab}) \cup \{\mathtt{l}'\})] \tag{58}$$

$$\mathsf{var} = \mathsf{e}' \text{ is not a subexpression of } \mathsf{e}_2 \tag{59}$$

$$\mathsf{C}'' = \{[\![\mathtt{var\_1}]\!] \;\leq\; [\mathsf{m} : ([\![\mathtt{var\_1.m}]\!], \circ)], \tag{60}$$
$$[\![\mathtt{var\_1}']\!] \;\leq\; [\mathsf{m} : ([\![\mathtt{var\_1}'\mathtt{.m}]\!], \bullet)], \tag{61}$$
$$[\![\mathtt{var\_1}']\!] \;\lhd_{\mathsf{m}}\; [\![\mathtt{var\_1}]\!], \tag{62}$$
$$[\![\mathbb{e}']\!] \;\leq\; [\![\mathtt{var\_1}'\mathtt{.m}]\!] \tag{63}$$
$$[\![\mathbb{e}']\!] \;\leq\; [\![\mathtt{var\_1}'\mathtt{.m} = \mathbb{e}]\!]\} \tag{64}$$
$$\mathsf{C} = \mathsf{C}' \;\cup\; \mathsf{C}'' \tag{65}$$

From (2) and (65) we have that S is a solution to $\mathsf{C}' \cup \mathsf{C}''$ and so trivially:

$$\mathsf{S} \vdash \mathsf{C}' \tag{66}$$
$$\mathsf{S} \vdash \mathsf{C}'' \tag{67}$$

Let:

$$\Gamma'' = \Gamma_{\mathsf{gen}}(\gamma'', \mathsf{S}) \tag{68}$$

From (3) with Definition 4 we get:

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}' \tag{69}$$

From (55),(66), (69), (4), (5) and (68) we apply the inductive hypothesis to get that there exist a t such that:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}' : \mathsf{t} \parallel \Gamma'' \tag{70}$$
$$\mathsf{t} \leq \mathsf{S}([\![\mathbb{e}']\!]) \tag{71}$$

From (67) with (60) and satisfaction rule $(solMember)$, from Figure 5.3, we get that $\mathsf{S}([\![\mathtt{var\_1}]\!])$ must be an object type such that:

$$\mathsf{S}([\![\mathtt{var\_1}]\!])(\mathsf{m}) \leq (\mathsf{S}([\![\mathtt{var\_1.m}]\!]), \circ) \tag{72}$$

Hence, there exists some t'' and ψ such that:

$$\mathsf{S}([\![\mathtt{var\_1}]\!])(\mathsf{m}) = (\mathsf{t}'', \psi) \tag{73}$$
$$\mathsf{t}'' \equiv \mathsf{S}([\![\mathtt{var\_1.m}]\!]) \tag{74}$$

By the same argument as above, from (66) and (61) we have $\mathsf{S}([\![\mathtt{var\_1}']\!])(\mathsf{m}) \leq (\mathsf{S}([\![\mathtt{var\_1}'\mathtt{.m}]\!]), \bullet)$; therefore, we have that there exists some t''' such that:

$$\mathsf{S}(\llbracket \mathsf{var\_l}' \rrbracket)(\mathsf{m}) = (\mathsf{t}''', \bullet) \tag{75}$$
$$\mathsf{t}''' \equiv \mathsf{S}(\llbracket \mathsf{var\_l}'.\mathsf{m} \rrbracket) \tag{76}$$

From (62), (67) and satisfaction rule $(solMemChange)$ we get:

$$\forall \mathsf{m}' \neq \mathsf{m} : \mathsf{S}(\llbracket \mathsf{var\_l}' \rrbracket)(\mathsf{m}') \equiv \mathsf{S}(\llbracket \mathsf{var\_l} \rrbracket)(\mathsf{m}') \tag{77}$$
$$\mathsf{S}(\llbracket \mathsf{var\_l}' \rrbracket)(\mathsf{m}) \leq \mathsf{S}(\llbracket \mathsf{var\_l} \rrbracket)(\mathsf{m}) \tag{78}$$

From (78) with (73) and (75) we get:

$$(\mathsf{t}''', \bullet) \leq (\mathsf{t}'', \psi) \tag{79}$$

From subtyping rule $(\leq mem)$ with (79) we get:

$$\mathsf{t}'' \equiv \mathsf{t}''' \tag{80}$$

From (80), (74) and (76) we get:

$$\mathsf{S}(\llbracket \mathsf{var\_l}'.\mathsf{m} \rrbracket) \equiv \mathsf{t}'' \tag{81}$$

From (81) and subtyping rule $(\leq cong)$ we get:

$$\mathsf{S}(\llbracket \mathsf{var\_l}'.\mathsf{m} \rrbracket) \leq \mathsf{t}'' \tag{82}$$

From (63), (67) and satisfaction rule $(solSub)$ we get:

$$\mathsf{S}(\llbracket \mathbb{e}' \rrbracket) \leq \mathsf{S}(\llbracket \mathsf{var\_l}'.\mathsf{m} \rrbracket) \tag{83}$$

From (71), (83) (82) and the transitivity of subtyping we get:

$$\mathsf{t} \leq \mathsf{t}'' \tag{84}$$

From (56) and (68) with the Definition of Γ_{gen} we get:

$$\Gamma''(\mathsf{var}) = \mathsf{S}(\llbracket \mathsf{var\_l} \rrbracket) \tag{85}$$

Let:

$$\Gamma''' = \Gamma''[\mathsf{var} \mapsto \mathsf{S}(\llbracket \mathsf{var\_l} \rrbracket)[\mathsf{m} \mapsto (\mathsf{t}'', \bullet)]] \tag{86}$$

From (70), (59), (73), (85), (84) and (86) we apply type rule $(assignAdd)$, from Figure 4.11, to get:

$$\mathbb{P}, \Gamma \vdash \mathsf{var.m} = \mathsf{e}' : \mathsf{t} \parallel \Gamma''' \tag{87}$$

We must now show that $\Gamma''' = \Gamma_{\mathsf{gen}}(\gamma', \mathsf{S})$.

First, consider the case when $\mathsf{var}' \neq \mathsf{var}$.

From (86) and definition of environment update we get:

$$\Gamma'''(\mathsf{var}') = \Gamma''(\mathsf{var}') \tag{88}$$

From (58) and the definition of update we get for some $\mathtt{l}''$:

$$\gamma''(\mathsf{var}') = \gamma'(\mathsf{var}') = \mathtt{l}'' \tag{89}$$

From (88), (89) and (58) with the definition of update we get:

$$\Gamma'''(\mathsf{var}') = \mathsf{S}([\![\mathsf{var}'\_\mathtt{l}'']\!]) \tag{90}$$

Consider the case when $\mathsf{var}' = \mathsf{var}$.

From (77), (78), (73) and (80) we get:

$$\mathsf{S}([\![\mathsf{var}\_\mathtt{l}']\!]) = \mathsf{S}([\![\mathsf{var}\_\mathtt{l}]\!])[\mathsf{m} \mapsto (\mathsf{t}'', \bullet)] \tag{91}$$

From (91) with (86) we get:

$$\Gamma''' = \Gamma''[\mathsf{var} \mapsto \mathsf{S}([\![\mathsf{var}\_\mathtt{l}']\!])] \tag{92}$$

From (58) we get:

$$\gamma'(\mathtt{this}) = \mathtt{l}' \tag{93}$$

From (92), (93), (90), (89), (58) and the definition of Γ_{gen} we get:

$$\Gamma''' = \Gamma_{\mathsf{gen}}(\gamma', \mathsf{S}) \tag{94}$$

We see that (87) and (94) establish (7).

From (64), (67) and satisfaction rule $(solSub)$ we get:

$$\mathsf{S}([\![\mathbb{e}']\!]) \leq \mathsf{S}([\![\mathsf{var\_}\mathtt{1}'.\mathsf{m} = \mathbb{e}']\!]) \tag{95}$$

From (71), (95) and the transitivity of subtyping we get:

$$\mathsf{t} \leq \mathsf{S}([\![\mathsf{var\_}\mathtt{1}'.\mathsf{m} = \mathbb{e}']\!]) \tag{96}$$

We see that (96) and (54) establish (8).

Case: (new)

Given that the last rule applied was (new) we get that there exist suitable $\mathsf{e}', \mathbb{e}', \gamma'$ and C' so that:

$$\mathbb{e} \text{ is } \mathtt{new}\,\mathsf{f}(\mathbb{e}'), \mathsf{e} \text{ is } \mathtt{new}\,\mathsf{f}(\mathsf{e}') \tag{97}$$
$$\gamma \vdash \mathsf{e}' \ : \ \mathbb{e}' \mid\mid \gamma' \mid\mid \mathsf{C}' \tag{98}$$
$$\mathsf{C}'' = \{[\![\mathtt{this\_f}]\!]^{\circ} \tag{99}$$
$$\quad [\![\mathbb{e}']\!] \ \leq \ [\![\mathtt{x\_f}]\!] \tag{100}$$
$$\quad [\![\mathtt{ret\_f}]\!] \ \leq \ [\![\mathtt{new}\,\mathsf{f}(\mathbb{e}')]\!]\} \tag{101}$$
$$\mathsf{C} = \mathsf{C}' \ \cup \ \mathsf{C}'' \tag{102}$$

From (2) with (102) we have that S is a solution to $\mathsf{C}' \ \cup \ \mathsf{C}''$ and so trivially:

$$\mathsf{S} \vdash \mathsf{C}' \tag{103}$$
$$\mathsf{S} \vdash \mathsf{C}'' \tag{104}$$

From (3) with Definition 4 we get:

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}' \tag{105}$$

From (98), (103), (105), (4), (5), and (6) we apply the inductive hypothesis to get that there exists a t such that:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}' : \mathsf{t} \mid\mid \Gamma'' \tag{106}$$
$$\mathsf{t} \leq \mathsf{S}([\![\mathbb{e}']\!]) \tag{107}$$

From (3) with Definition 4 and (4) we get:

$$\mathbb{P}(\mathsf{f}) \ = \ \mathtt{function}\,\mathsf{f}(\mathtt{x}) : \mathsf{G}\ \{\mathsf{e}''\} \tag{108}$$

with G such that:

$$\mathsf{G}(\texttt{this}) = \mathsf{S}([\![\texttt{this\_f}]\!]) \tag{109}$$

$$\mathsf{G}(\texttt{x}) = \mathsf{S}([\![\texttt{x\_f}]\!]) \tag{110}$$

$$\mathsf{G}(\texttt{ret}) = \mathsf{S}([\![\texttt{ret\_f}]\!]) \tag{111}$$

From (2) and (100) with solution satisfaction rule $(solSub)$, from Figure 5.3, we get:

$$\mathsf{S}([\![\mathbb{e}']\!]) \leq \mathsf{S}([\![\texttt{x\_f}]\!]) \tag{112}$$

From (112), (107), (110) and the transitivity of subtyping we get:

$$\mathsf{t} \leq \mathsf{G}(\texttt{x}) \tag{113}$$

From (2) and (99) with solution satisfaction rule $(solNoDefs)$ we get:

$$\{\mathsf{m} \mid \mathsf{S}([\![\texttt{this\_f}]\!])(\mathsf{m}) = (\mathsf{t}, \bullet)\} = \emptyset \tag{114}$$

From (106), (108), (113), (114) and (111) we apply type rule $(call)$, from Figure 4.11, to get:

$$\mathbb{P}, \Gamma \vdash \texttt{new}\ \mathsf{f}(\mathsf{e}) : \mathsf{G}(\texttt{ret}) \parallel \Gamma' \tag{115}$$

We see that (115) establishes (7).

Finally, from (101), (2) and solution satisfaction rule $(solSub)$ we get:

$$\mathsf{S}([\![\texttt{ret\_f}]\!]) \leq \mathsf{S}([\![\texttt{new}\ \mathsf{f}(\mathbb{e}')]\!]) \tag{116}$$

From (116) and (111) we get:

$$\mathsf{G}(\texttt{ret}) \leq \mathsf{S}([\![\texttt{new}\ \mathsf{f}(\mathbb{e})]\!]) \tag{117}$$

We see that (117) establishes (8).

Case: $(cond)$

Given that the last rule applied was $(cond)$ we get that there exists suitable $\mathsf{e}_1,\ \mathsf{e}_2,\ \mathsf{e}_3, \mathbb{e}_1,\ \mathbb{e}_2,\ \mathbb{e}_3,\ \gamma_1,\ \gamma_2,\ \gamma_3,\ \mathsf{C}_1,\ \mathsf{C}_2$ and C_3 so that:

$$\mathbb{e} \text{ is } \mathbb{e}_1 ?\ \mathbb{e}_2 :\ \mathbb{e}_3, \mathsf{e} \text{ is } \mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3 \tag{118}$$

$$\gamma \vdash \mathsf{e}_1 \ : \ \mathbb{e}_1 \,||\, \gamma_1 \,||\, \mathsf{C}_1 \tag{119}$$

$$\gamma_1 \vdash \mathsf{e}_2 \ : \ \mathbb{e}_2 \,||\, \gamma_2 \,||\, \mathsf{C}_2 \tag{120}$$

$$\gamma_2 \vdash \mathsf{e}_3 \ : \ \mathbb{e}_3 \,||\, \gamma_3 \,||\, \mathsf{C}_3 \tag{121}$$

$$\gamma_2(\mathtt{this}) = \mathtt{l}_2,\ \gamma_3(\mathtt{this}) = \mathtt{l}_3,\ \gamma_2(\mathtt{x}) = \mathtt{l}'_2,\ \gamma_3(\mathtt{x}) = \mathtt{l}'_3 \tag{122}$$

$$\mathtt{l} \ \notin \ \gamma_3(\mathtt{lab}) \quad (\text{fresh } \mathtt{l}) \tag{123}$$

$$\gamma' = \{\mathtt{this} \ \mapsto \ \mathtt{l},\, \mathtt{x} \ \mapsto \ \mathtt{l},\, \mathtt{lab} \ \mapsto \ (\gamma_3(\mathtt{lab}) \cup \{\mathtt{l}\})\} \tag{124}$$

$$\mathsf{C}' = \{[\![\mathbb{e}_1]\!] \ \leq \ \mathsf{Int} \tag{125}$$

$$[\![\mathtt{this\_l_2}]\!] \ \lhd_* \ [\![\mathtt{this\_l}]\!],\ [\![\mathtt{this\_l_3}]\!] \ \lhd_* \ [\![\mathtt{this\_l}]\!], \tag{126}$$

$$[\![\mathtt{x\_l'_2}]\!] \ \lhd_* \ [\![\mathtt{x\_l}]\!],\ [\![\mathtt{x\_l'_3}]\!] \ \lhd_* \ [\![\mathtt{x\_l}]\!], \tag{127}$$

$$[\![\mathbb{e}_2]\!] \ \lhd_* \ [\![\mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3]\!],\ [\![\mathbb{e}_3]\!] \ \lhd_* \ [\![\mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3]\!] \tag{128}$$

$$\mathsf{C} = \mathsf{C}' \ \cup \ \mathsf{C}_1 \ \cup \ \mathsf{C}_2 \ \cup \ \mathsf{C}_3 \tag{129}$$

From (2) and (129) we have that S is a solution to $\mathsf{C}' \ \cup \ \mathsf{C}_1 \ \cup \ \mathsf{C}_2 \ \cup \ \mathsf{C}_3$ and so trivially:

$$\mathsf{S} \vdash \mathsf{C}' \tag{130}$$

$$\mathsf{S} \vdash \mathsf{C}_1 \tag{131}$$

$$\mathsf{S} \vdash \mathsf{C}_2 \tag{132}$$

$$\mathsf{S} \vdash \mathsf{C}_3 \tag{133}$$

From (3) with Definition 4 we get:

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}_1 \tag{134}$$

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}_2 \tag{135}$$

$$\mathsf{S}, \mathbb{P} \vdash \mathsf{e}_3 \tag{136}$$

Let:

$$\Gamma_1 = \Gamma_{\mathsf{gen}}(\gamma_1, \mathsf{S}) \tag{137}$$

From (119), (131), (134), (4), (5) and (137) we apply the inductive hypothesis to get that there exists a t_1 such that:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{t}_1 \,\|\, \Gamma_1 \tag{138}$$

$$\mathsf{t}_1 \leq \mathsf{S}([\![\mathbb{e}_1]\!]) \tag{139}$$

From (125), (130) and definition of satisfaction we have that $\mathsf{S}([\![\mathbb{e}_1]\!])$ must be of type Int; therefore, with (139) and definition of subtyping we have that t_1 must also be of type Int, and so with (138) we get:

$$\mathbb{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{Int} \parallel \Gamma_1 \tag{140}$$

Let:

$$\Gamma_2 = \Gamma_{\mathsf{gen}}(\gamma_2, \mathsf{S}) \tag{141}$$

From (120), (132), (135), (4), (137) and (141) we apply the inductive hypothesis to get that there exists a t_2 such that:

$$\mathbb{P}, \Gamma_1 \vdash \mathsf{e}_2 : \mathsf{t}_2 \parallel \Gamma_2 \tag{142}$$

$$\mathsf{t}_2 \leq \mathsf{S}(\llbracket \mathbb{e}_2 \rrbracket) \tag{143}$$

Let:

$$\Gamma_3 = \Gamma_{\mathsf{gen}}(\gamma_3, \mathsf{S}) \tag{144}$$

From (121), (133), (136), (4), (141) and (144) we apply the inductive hypothesis to get that there exists a t_3 such that:

$$\mathbb{P}, \Gamma_2 \vdash \mathsf{e}_3 : \mathsf{t}_3 \parallel \Gamma_3 \tag{145}$$

$$\mathsf{t}_3 \leq \mathsf{S}(\llbracket \mathbb{e}_3 \rrbracket) \tag{146}$$

We must now show that $\Gamma' = \Gamma_2 \sqcup \Gamma_3$.

From (126), (130) and satisfaction rule $(solAnyMemsChange)$ we get:

$$\mathsf{S}(\llbracket \mathtt{this\_l}_2 \rrbracket) \sqsubseteq \mathsf{S}(\llbracket \mathtt{this\_l} \rrbracket) \tag{147}$$

$$\mathsf{S}(\llbracket \mathtt{this\_l}_3 \rrbracket) \sqsubseteq \mathsf{S}(\llbracket \mathtt{this\_l} \rrbracket) \tag{148}$$

From (147) and the definition of $\sqsubseteq$ we get:

$$\mathsf{S}(\llbracket \mathtt{this\_l} \rrbracket)(\mathsf{m}) = (\mathsf{t}_1, \bullet) \implies \mathsf{S}(\llbracket \mathtt{this\_l}_2 \rrbracket)(\mathsf{m}) = (\mathsf{t}_2, \bullet) \tag{149}$$

$$\mathsf{t}_1 \equiv \mathsf{t}_2 \tag{150}$$

Similarly, with (148) and the definition of $\sqsubseteq$ we get:

$$\mathsf{S}(\llbracket \mathtt{this\_l} \rrbracket)(\mathsf{m}) = (\mathsf{t}_1, \bullet) \implies \mathsf{S}(\llbracket \mathtt{this\_l}_3 \rrbracket)(\mathsf{m}) = (\mathsf{t}_3, \bullet) \tag{151}$$

$$\mathsf{t}_1 \equiv \mathsf{t}_3 \tag{152}$$

From (149), (150), (151) and (152) with the definition of $\sqcup$, see Figure 4.15, we get:

$$\mathsf{S}([\![\texttt{this\_l}]\!] = \mathsf{S}([\![\texttt{this\_l}_2]\!]) \sqcup \mathsf{S}([\![\texttt{this\_l}_3]\!]) \tag{153}$$

From (6), (141), (144) and the definition of Γ_{gen} we get:

$$\Gamma'(\texttt{this}) = \mathsf{S}([\![\texttt{this\_l}]\!]) \tag{154}$$

$$\Gamma_2(\texttt{this}) = \mathsf{S}([\![\texttt{this\_l}_2]\!]) \tag{155}$$

$$\Gamma_3(\texttt{this}) = \mathsf{S}([\![\texttt{this\_l}_3]\!]) \tag{156}$$

With (153), (154), (155) and (156) we get:

$$\Gamma'(\texttt{this}) = \Gamma_2(\texttt{this}) \sqcup \Gamma_3(\texttt{this}) \tag{157}$$

By the same argument we get:

$$\Gamma'(\texttt{x}) = \Gamma_2(\texttt{x}) \sqcup \Gamma_3(\texttt{x}) \tag{158}$$

From (157), (158) and the definition of $\sqcup$ for environments, see Figure 4.15, we get:

$$\Gamma' = \Gamma_2 \sqcup \Gamma_3 \tag{159}$$

From (140), (142) and (145) we apply the type rule $(cond)$ to get:

$$\mathbb{P}, \Gamma \vdash (\mathsf{e}_1 ?\, \mathsf{e}_2 :\ \mathsf{e}_3) : \mathsf{t}_2 \sqcup \mathsf{t}_3 \parallel \Gamma_2 \sqcup \Gamma_3 \tag{160}$$

From (159) with (160) we get:

$$\mathbb{P}, \Gamma \vdash (\mathsf{e}_1 ?\, \mathsf{e}_2 :\ \mathsf{e}_3) : \mathsf{t}_2 \sqcup \mathsf{t}_3 \parallel \Gamma' \tag{161}$$

We see that (161) establishes (7).

From (128), (130), the satisfaction rule $(solAnyMemsChange)$, and Proposition 1 we get:

$$\mathsf{S}([\![\mathbb{e}_2]\!]) \leq \mathsf{S}([\![\mathbb{e}_1 ?\, \mathbb{e}_2 :\ \mathbb{e}_3]\!]) \tag{162}$$

$$\mathsf{S}(\llbracket \mathbb{e}_3 \rrbracket) \leq \mathsf{S}(\llbracket \mathbb{e}_1 ?\ \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket) \tag{163}$$

From (162), (163), (143) and (146) with the transitivity of subtyping we get:

$$\mathsf{t}_2 \leq \mathsf{S}(\llbracket \mathbb{e}_1 ?\ \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket) \tag{164}$$

$$\mathsf{t}_3 \leq \mathsf{S}(\llbracket \mathbb{e}_1 ?\ \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket) \tag{165}$$

From (164), (165) and the definition of subtyping we get:

$$\mathsf{t}_2 \sqcup \mathsf{t}_3 \leq \mathsf{S}(\llbracket \mathbb{e}_1 ?\ \mathbb{e}_2 :\ \mathbb{e}_3 \rrbracket) \tag{166}$$

We see that (166) establishes (8).

□

5.4.2 Soundness of Constraints for Programs

Theorem 3 asserts the soundness of the constraints at the program level. If there is a solution to the constraints for a program they can be used to generate a well-typed version of the program.

Theorem 3 *For program* P *and solution* S *if:*

$$\vdash \mathsf{P} :\ \mathsf{C} \tag{1}$$

$$\mathsf{S} \vdash \mathsf{C} \tag{2}$$

$$\mathbb{P} = \mathcal{T}(\mathsf{P}, \mathsf{S}) \tag{3}$$

then:

$$\vdash \mathbb{P} \diamond \tag{4}$$

Proof: From (1) and constraint generation rule *(prog)*, see Figure 5.6, we get:

$$\mathsf{P} = \texttt{function}\ \mathsf{f}_1(\texttt{x})\ \{\ \mathsf{e}_1\ \}\ \cdots\ \texttt{function}\ \mathsf{f}_n(\texttt{x})\ \{\ \mathsf{e}_n\ \} \tag{5}$$

$$\gamma_i = \{\texttt{this}\ \mapsto\ \mathsf{f}_i,\ \texttt{x}\ \mapsto\ \mathsf{f}_i,\ \texttt{lab}\ \mapsto\ \gamma'_{i-1}(\texttt{lab})\}\ \wedge\ \gamma_0 = \emptyset \tag{6}$$

$$\gamma_i \vdash \mathsf{e}_\mathsf{i}\ :\ \mathbb{e}_i \,||\, \gamma'_i \,||\, \mathsf{C}_i \quad 1 \leq i \leq n \tag{7}$$

$$\mathsf{f} \in \mathsf{e}_\mathsf{i}\ \Longrightarrow\ \mathsf{P}(\mathsf{f}) \neq \mathcal{U}df\ \ 1 \leq i \leq n \tag{8}$$

$$\mathsf{C} = \bigcup_{i\,\in\,1..n} \mathsf{C}_i \cup\ \{\llbracket \mathbb{e}_\mathtt{i} \rrbracket\ \leq\ \llbracket \texttt{ret\_f}_\mathtt{i} \rrbracket\} \tag{9}$$

For each function f_k from $\mathbb{P}$, with (3), the definition of $\mathcal{T}$, and Definition 4 we get for $k \in 1...n$:

$$\mathbb{P}(\mathsf{f}_k) = \texttt{function}\,\mathsf{f}(\texttt{x}) : (\mathsf{S}([\![\texttt{this\_f}_\mathsf{k}]\!]) \times \mathsf{S}([\![\texttt{x\_f}_\mathsf{k}]\!]) \rightarrow \mathsf{S}([\![\texttt{ret\_f}_\mathsf{k}]\!]))\ \{\mathsf{e}_\mathsf{k}\} \tag{10}$$

From (6) and (7) we get for $k \in 1...n$:

$$\gamma_k = \{\texttt{this} \mapsto \mathsf{f}_k,\ \texttt{x} \mapsto \mathsf{f}_k,\ \texttt{lab} \mapsto \gamma_{k-1}(\texttt{lab})\} \tag{11}$$

$$\gamma_k \vdash \mathsf{e}_\mathsf{k}\ :\ \mathbb{e}_k \mid\mid \gamma'_k \mid\mid \mathsf{C}_k \tag{12}$$

From the definition of Γ_{gen} with (11) we get for $k \in 1...n$:

$$\Gamma_k = \Gamma_{\mathsf{gen}}(\gamma_k, \mathsf{S}) = \{\texttt{this} : \mathsf{S}([\![\texttt{this\_f}_\mathsf{k}]\!]),\ \texttt{x} : \mathsf{S}([\![\texttt{x\_f}_\mathsf{k}]\!])\} \tag{13}$$

Let for $k \in 1...n$:

$$\Gamma'_k = \Gamma_{\mathsf{gen}}(\gamma'_k, \mathsf{S}) \tag{14}$$

From (8), (2), (3) and the definition of $\mathcal{T}$ we get for $k \in 1...n$:

$$\mathbb{P}, \mathsf{S} \vdash \mathsf{e}_\mathsf{k} \tag{15}$$

From (12), (2), (9) (noting that S is a solution to C_i for $i \in 1..n$), (15), (3), (13) and (14) we apply Theorem 2 to get for $k \in 1...n$ there exists $\mathsf{t}_1...\mathsf{t}_k$ such that:

$$\mathbb{P}, \Gamma_k \vdash \mathsf{e}_\mathsf{k} : \mathsf{t}_k \mid\mid \Gamma'_k \tag{16}$$

$$\mathsf{t}_k \leq \mathsf{S}([\![\mathsf{e}_\mathsf{k}]\!]) \tag{17}$$

From (9), (2) and solution satisfaction rule $(solSub)$, see Figure 5.3, we get for $k \in 1...n$:

$$\mathsf{S}([\![\mathbb{e}_\mathsf{k}]\!]) \leq \mathsf{S}([\![\texttt{ret\_f}_\mathsf{k}]\!]) \tag{18}$$

From (18), (17) and the transitivity of subtyping we get for $k \in 1...n$:

$$\mathsf{t} \leq \mathsf{S}([\![\texttt{ret\_f}_\mathsf{k}]\!]) \tag{19}$$

From (2) and the satisfaction rule $(solSat)$ we have that all the types in S are well-formed; hence, for $k \in 1...n$ with (10), (16), (13), (19) and the definition of well-formed programs, see Figure 4.12, we establish (4).

$\square$

5.5 Constraint Closure

Constraint closure simplifies the extraction of a solution, if one exists, from a set of constraints. The closing relation, $\mathsf{C} \longrightarrow \mathsf{C}'$, is defined in Figure 5.8. We assume that the closure of a set of constraints includes the reflexive closure. We now discuss each of the closure rules and then show that the closure relation is sound.

In $(closeTrans)$ we add a constraint implied by the transitivity of subtyping.

In $(closeTransMem)$ the type variable τ is required to have the same members, with the same types and potentially more definite annotations, as type variable τ'. Therefore, τ is also required to have the member m with type τ'' and annotation ψ, as expressed by $\tau \le [\mathsf{m} : (\tau'', \psi)]$. For example, consider the following constraints:

$$\begin{aligned}\mathsf{C} \;=\; &\{[\![\texttt{this\_1}]\!] \le [\texttt{mSec} : ([\![\texttt{this\_1.mSec}]\!], \bullet)],\\ &[\![\texttt{this\_2}]\!] \lhd_{\mathsf{add}} [\![\texttt{this\_1}]\!]\}\end{aligned}$$

Closure rule $(closeTransMem)$ applied to C adds constraint: $[\![\texttt{this\_2}]\!] \le [\texttt{mSec} : ([\![\texttt{this\_1.mSec}]\!], \bullet)]$; thus, ensuring that $[\![\texttt{this\_2}]\!]$ has member $\texttt{mSec}$.

In $(closeBalance)$ the type variable τ is required to be a subtype of type variable τ', and τ is required to be a subtype of a σ, *i.e.* either of type Int, or of a function type. Because the subtype relationship for type Int and function types is the identity, it follows that τ and σ will have to be "the same"; therefore, it follows that τ' will have to be a subtype of σ. For example, consider the following constraints:

$$\begin{aligned}\mathsf{C} \;=\; &\{[\![\texttt{1000}]\!] \le [\![\texttt{x\_Date}]\!],\\ &[\![\texttt{1000}]\!] \le [\![\mathsf{Int}]\!]\}\end{aligned}$$

Closure rule $(closeBalance)$ applied to C adds constraint $[\![\texttt{x\_Date}]\!] \le [\![\mathsf{Int}]\!]$ expressing that $[\![\texttt{x\_Date}]\!]$ must have an integer type.

In $(closeBalanceMem)$ the type variable τ is required to have member m' with type τ'' and annotation ψ. Because of constraint $\tau \lhd_{\mathsf{m}'} \tau'$, τ' is required to have the same members as τ with the same annotations with the exception of member m of τ which can be more defined. It follows that τ' will also have member m with type τ''. Care must be taken with the annotation: if $\mathsf{m} = \mathsf{m}'$ then the annotation will be $\circ$, otherwise they will be same.

Intuitively, transitivity will add members in one direction and $(closeBalanceMem)$ will add members in the *reverse* direction (against transitivity). To see why special treatment of the annotation for m is required, consider constraints C:

$$\begin{aligned}\mathsf{C} \quad = \quad & \{[\![\texttt{this\_2}]\!] \lhd_{\mathsf{add}} [\![\texttt{this\_1}]\!], \\ & [\![\texttt{this\_1}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_1}.\mathsf{add}]\!], \circ)], \\ & [\![\texttt{this\_2}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_2}.\mathsf{add}]\!], \bullet)]\}\end{aligned}$$

If we were to add constraint $[\![\texttt{this\_1}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_1}.\mathsf{add}]\!], \bullet)]$ (*instead of* $[\![\texttt{this\_1}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_1}.\mathsf{add}]\!], \circ)]$) a solution to C would *not* be a solution to this constraint, because `add` would have to be definite in the type of $[\![\texttt{this\_1}]\!]$.

Before proceeding with the rest of the closure rules, we define (in Definition 5) when two type variables are "equivalent".

Definition 5 *We say that two type variables are "equivalent" in a set of constraints* C *if* $\tau \ \leq \ \tau', \tau' \ \leq \ \tau \in \mathsf{C}$

In $(closeCong)$ the same type variable, τ, is required to contain a member m with type τ' and type τ''. It follows that τ' should be equivalent with τ''. For example, consider constraints:

$$\begin{aligned}\mathsf{C} \quad = \quad & \{[\![\texttt{this\_2}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_1}.\mathsf{add}]\!], \circ)], \\ & [\![\texttt{this\_2}]\!] \ \leq \ [\mathsf{add} : ([\![\texttt{this\_2}.\mathsf{add}]\!], \bullet)]\}\end{aligned}$$

Closure rule $(closeCong)$ applied to C would add constraints:

$$\begin{aligned}[\![\texttt{this\_1}.\mathsf{add}]\!] \ \leq \ [\![\texttt{this\_2}.\mathsf{add}]\!] \\ [\![\texttt{this\_2}.\mathsf{add}]\!] \ \leq \ [\![\texttt{this\_1}.\mathsf{add}]\!]\end{aligned}$$

In $(closeCongFunc)$ type variable τ is required to be a subtype of two function types. The subtype relation for function types requires that they are congruent; therefore, it follows that the receiver, argument and return types should be equivalent.

We say that a set of constraints is closed, see Definition 6, if the application of the closure relation adds no new constraints.

Definition 6 C *is closed,* $\vdash \mathsf{C} \diamond_{cl}$*, if for any* C'*:* $\mathsf{C} \longrightarrow \mathsf{C}'$ *implies that* $\mathsf{C} = \mathsf{C}'$*.*

We require that if a set of constraints, C, has a solution S then any new constraints added via closure are satisfied by S, *i.e.* a set of constraints and its closure have the same set of solutions. With Lemma 3 we prove this property.

Lemma 3 *For constraints* C *and solution* S *if:*

$$\mathsf{S} \vdash \mathsf{C} \tag{1}$$
$$\mathsf{C} \longrightarrow \mathsf{C}' \tag{2}$$

then:

$$\mathsf{S} \vdash \mathsf{C}' \tag{3}$$

Proof: We proceed by showing that for each of the closure rules the added constraints preserve the solution.

Case: (closeTrans)

From (2) with closure rule $(closeTrans)$ we get:

$$\tau \ \leq \ \tau', \tau' \ \leq \ \rho \in \mathsf{C} \tag{4}$$

From (1), (4) and solution satisfaction rule $(solSub)$ we get:

$$\mathsf{S}(\tau) \leq \mathsf{S}(\tau') \tag{5}$$

We now do case analysis on ρ.

Subcase: $\rho \ = \tau''$

Given that $\rho = \tau$ with (4) we get:

$$\tau' \ \leq \ \tau'' \in \mathsf{C} \tag{6}$$

From (1), (6) and solution satisfaction rule $(solSub)$ we get:

$$\mathsf{S}(\tau') \leq \mathsf{S}(\tau'') \tag{7}$$

From (5), (7) and the transitivity of subtyping we get:

$$\mathsf{S}(\tau) \leq \mathsf{S}(\tau'') \tag{8}$$

With (8) we apply solution satisfaction rule $(solSub)$ to get:

$$\mathsf{S} \vdash \tau' \ \leq \ \tau'' \tag{9}$$

Subcase: $\rho = [\mathsf{m} : (\tau'', \psi)]$

Given that $\rho = [\mathsf{m} : (\tau'', \psi)]$ with (4) we get:

$$\tau' \ \leq \ [\mathsf{m} : (\tau'', \psi)] \in \mathsf{C} \tag{10}$$

From (1), (10) and solution satisfaction rule $(solMember)$ we get:

$$\mathsf{S}(\tau')(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{11}$$

For some $\mathsf{t}', \mathsf{t}'', \psi'$ and ψ'' let:

$$\mathsf{S}(\tau)(\mathsf{m}) = (\mathsf{t}'', \psi'') \tag{12}$$
$$\mathsf{S}(\tau')(\mathsf{m}) = (\mathsf{t}', \psi') \tag{13}$$

From (5), (11), (12), (13) and the definition of subtyping we get:

$$\mathsf{t}'' \equiv \mathsf{t}' \equiv \mathsf{S}(\tau'') \tag{14}$$
$$\psi'' \leq \psi' \leq \psi \tag{15}$$

From (5), (11), (14), (15) and the definition of subtyping, rule $(\leq mem)$, we get:

$$\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{16}$$

From (16) we apply solution satisfaction rule $(solMem)$ to get:

$$\mathsf{S} \vdash \tau \ \leq \ [\mathsf{m} : (\tau'', \psi)] \tag{17}$$

Subcase: $\rho \ = \sigma$ and $\sigma = \mathsf{Int}$

From (1) we have that $\mathsf{S}(\tau') = \mathsf{Int}$; from the definition of subtyping with $\mathsf{S}(\tau) \leq \mathsf{S}(\tau')$ we get $\mathsf{S}(\tau) = \mathsf{Int}$. Hence, we get $\mathsf{S} \vdash \tau \ \leq \ \mathsf{Int}$.

Subcase: $\rho = \sigma$ and $\sigma = (\tau_1 \times \tau_2 \rightarrow \tau_3)$

Given that $\rho = \sigma$ and $\sigma = (\tau_1 \times \tau_2 \rightarrow \tau_3)$ with (4) we get:

$$\tau' \ \leq \ (\tau_1 \times \tau_2 \rightarrow \tau_3) \in \mathsf{C} \tag{18}$$

From (1), (18) and the solution satisfaction rule $(solSubFunc)$ we get:

$$\mathsf{S}(\tau') \leq (\mathsf{S}(\tau_1) \times \mathsf{S}(\tau_2) \rightarrow \mathsf{S}(\tau_3)) \tag{19}$$

For some $\mathsf{t}_1', \mathsf{t}_2'$ and t_3' let:

$$\mathsf{S}(\tau') = (\mathsf{t}_1' \times \mathsf{t}_2' \rightarrow \mathsf{t}_3') \tag{20}$$

From (20), (19) and the definition of subtyping we get:

$$\mathsf{t}_1' \equiv \mathsf{S}(\tau_1),\ \mathsf{t}_2' \equiv \mathsf{S}(\tau_2),\ \mathsf{t}_3' \equiv \mathsf{S}(\tau_3) \tag{21}$$

From (5), (20) and the definition of subtyping we have that $\mathsf{S}\ (\tau)$ must also be a function type; therefore, for some $\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3$ we get:

$$\mathsf{S}(\tau) = (\mathsf{t}_1 \times \mathsf{t}_2 \rightarrow \mathsf{t}_3) \tag{22}$$

From (22), (21), (5) and the definition of subtyping we get:

$$\mathsf{t}_1 \equiv \mathsf{S}(\tau_1),\ \mathsf{t}_2 \equiv \mathsf{S}(\tau_2),\ \mathsf{t}_3 \equiv \mathsf{S}(\tau_3) \tag{23}$$

From (22), (23) and the definition of subtyping, rule $(\leq func)$, we get:

$$\mathsf{S}(\tau) \leq (\mathsf{S}(\tau_1) \times \mathsf{S}(\tau_2) \rightarrow \mathsf{S}(\tau_3)) \tag{24}$$

From (24) we apply solution satisfaction rule $(solSubFunc)$ to get:

$$\mathsf{S} \vdash \tau \ \leq \ (\tau_1 \times \tau_2 \rightarrow \tau_3) \tag{25}$$

Case: $(closeTransMem)$

From (2) with closure rule $(closeTransMem)$ we get:

$$\tau \ \lhd \ \tau',\ \tau' \ \leq \ [\mathsf{m} : (\tau'', \psi)] \in \mathsf{C} \tag{26}$$

From (1), (26) and the solution satisfaction rule $(solMember)$ we get:

$$\mathsf{S}(\tau')(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{27}$$

We have two cases to consider for $\lhd$: $\lhd_*$ and $\lhd_{\mathsf{m}'}$ (for some m').

Subcase: $\lhd_*$

Given that $\lhd$ is $\lhd_*$ from (26), (1) and solution satisfaction rule $(solAnyMemsChange)$ we get:

$$\mathsf{S}(\tau) \ \sqsubseteq \mathsf{S}(\tau') \tag{28}$$

From (27) we have that $\mathsf{S}(\tau')$ has a member m, and from (28) with Proposition 1 we get:

$$\mathsf{S}(\tau)(\mathsf{m}) \leq \mathsf{S}(\tau')(\mathsf{m}) \tag{29}$$

From (29), (27) and the transitivity of subtyping we get:

$$\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{30}$$

Subcase: $\lhd_{\mathsf{m}'}$

Given that $\lhd$ is $\lhd_{\mathsf{m}'}$ from (26), (1) and solution satisfaction rule $(solMemChange)$ we get:

$$\forall\, \mathsf{m}'' \neq \mathsf{m}' \ : \ \mathsf{S}(\tau)(\mathsf{m}'') \equiv \ \mathsf{S}(\tau')(\mathsf{m}'') \tag{31}$$

$$\mathsf{S}(\tau)(\mathsf{m}') \leq \mathsf{S}(\tau')(\mathsf{m}') \tag{32}$$

There are two cases to consider: $\mathsf{m} = \mathsf{m}'$ and $\mathsf{m} \neq \ \mathsf{m}'$.

When $\mathsf{m} = \mathsf{m}'$ with (32), (27), the definition of subtyping, and the transitivity of subtyping we get:

$$\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{33}$$

When $\mathsf{m} \neq \ \mathsf{m}'$ the case follows from (31) and subtyping rule $(\leq cong)$.

From (30), (33) and solution satisfaction rule $(solMember)$ we get:

$$\mathsf{S} \vdash \tau \ \leq \ [\mathsf{m} : (\tau'', \psi)] \tag{34}$$

Case: $(closeBalance)$

When $\rho = \mathsf{Int}$ and $\rho = (\tau_1 \times \tau_2 \to \tau_3)$ the proof is similar to the case for $(closeTrans)$.

Case: $(closeBalanceMem)$

From (2) with closure rule $(closeBalance)$ we get:

$$\tau \ \lhd_{\mathsf{m}'} \tau', \ \tau \ \leq \ [\mathsf{m} : (\tau'', \psi)] \in \mathsf{C} \tag{35}$$

From (1), (35) and solution satisfaction rule $(solMember)$ we get:

$$\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau''), \psi) \tag{36}$$

From (1), (35) and solution satisfaction rule $(solMemChange)$ we get:

$$\forall \ \mathsf{m}' \neq \mathsf{m} \ : \ \mathsf{S}(\tau)(\mathsf{m}') \equiv \ \mathsf{S}(\tau')(\mathsf{m}') \tag{37}$$

$$\mathsf{S}(\tau)(\mathsf{m}) \leq \mathsf{S}(\tau')(\mathsf{m}) \tag{38}$$

There are two cases to consider: $\mathsf{m} = \mathsf{m}'$ and $\mathsf{m} \neq \mathsf{m}'$.

Subcase: $\mathsf{m} = \mathsf{m}'$

For some $\mathsf{t}, \mathsf{t}', \psi'$ and ψ'' let:

$$\mathsf{S}(\tau)(\mathsf{m}) = (\mathsf{t}, \psi') \tag{39}$$

$$\mathsf{S}(\tau')(\mathsf{m}) = (\mathsf{t}', \psi'') \tag{40}$$

From (36), (38), (39), (40) and the definition of subtyping we get:

$$\mathsf{t} \equiv \mathsf{t}' \equiv \mathsf{S}(\tau'') \tag{41}$$

From the definition of subtyping we get:

$$\psi' \leq \circ \tag{42}$$

From (36), (38), (41), (42) and the definition of subtyping, rule $(\leq mem)$, we get:

$$\mathsf{S}(\tau')(\mathsf{m}) \leq (\mathsf{S}(\tau''), \circ) \tag{43}$$

From (43) we apply solution satisfaction rule $(solMember)$ to get:

$$\mathsf{S} \vdash \tau' \ \leq \ [\mathsf{m} : (\tau'', \circ)] \tag{44}$$

Subcase: $\mathsf{m} \neq \mathsf{m}'$

Same argument as the case when $\mathsf{m} = \mathsf{m}'$ except that trivially we have $\psi = \psi' = \psi''$ in $(closeBalanceMem)$.

Case: $(closeBalanceMems)$

Same argument as case $(closeBalanceMem)$.

Case: $(closeCong)$

Straightforward.

Case: $(closeCongFunc)$

Straightforward.

□

5.6 Well-formed Constraints

Well-formedness of constraints, $\vdash \mathsf{C}\diamond$, see Figure 5.9, ensures that C can be used to create a solution that satisfies C. For a set of constraints to be well-formed, it must be closed and all the constraints must be well-formed.

Intuitively, rules $(wlfNoDefs)$ and $(wlfEnsureObj)$ correspond to the solution satisfaction rules $(solNoDefs)$ and $(solMemChange)$ respectively, see Figure 5.3, where $\mathsf{S}\ (\tau)(\mathsf{m})$ is represented by looking for constraints detailing members, $\tau \leq [\mathsf{m} : (\_ , \_)]$.

Surprisingly, well-formedness rule $(wlfEnsureObj)$ only requires that τ' has a member m and does not need the other requirements present in $(solMemChange)$. Also, there is no well-formedness rule for constraints of the form $\tau \leq \tau'$. We shall see in Section 5.8 (when we prove that well-formed constraints do indeed give satisfying solutions) that by closing constraints we satisfy the other requirements imposed by solution satisfaction.

Rules $(wlfMix1)$, $(wlfMix2)$ and $(wlfMix3)$ ensure that the constraints do not *mix* object types, function types and integers. Conflicting information with respect to a type variable indicates a type error in the program. For example, consider the following program fragment:

$$\frac{\begin{array}{l}\mathsf{c}_1, ..., \mathsf{c}_n \;\longrightarrow\; \mathsf{c}'_1, ...\mathsf{c}'_m \\ \mathsf{c}_1, ..., \mathsf{c}_n \;\in\; \mathsf{C}\end{array}}{\mathsf{C} \;\longrightarrow\; \mathsf{C} \cup \{\mathsf{c}'_1, ...\mathsf{c}'_m\}}\,(closeMany1) \qquad \frac{}{\mathsf{C} \;\longrightarrow\; \mathsf{C}}\,(closeMany2)$$

$$\frac{}{\tau \;\leq\; \tau',\; \tau' \;\leq\; \rho \;\longrightarrow\; \tau \;\leq\; \rho}\,(closeTrans)$$

$$\frac{}{\tau \;\lhd\; \tau',\; \tau' \;\leq\; [\mathsf{m} : (\tau'', \psi)] \;\longrightarrow\; \tau \;\leq\; [\mathsf{m} : (\tau'', \psi)]}\,(closeTransMem)$$

$$\frac{}{\tau \;\leq\; \tau',\; \tau \;\leq\; \sigma \;\longrightarrow\; \tau' \;\leq\; \sigma}\,(closeBalance)$$

$$\frac{\psi' = \circ \; (if\ \mathsf{m} = \mathsf{m}') \;\; \psi' = \psi \; (otherwise)}{\tau \;\lhd_{\mathsf{m}'}\; \tau',\; \tau \;\leq\; [\mathsf{m} : (\tau'', \psi)] \;\longrightarrow\; \tau' \;\leq\; [\mathsf{m} : (\tau'', \psi')]}\,(closeBalanceMem)$$

$$\frac{}{\tau \;\lhd_*\; \tau',\; \tau \;\leq\; [\mathsf{m} : (\tau'', \psi)] \;\longrightarrow\; \tau' \;\leq\; [\mathsf{m} : (\tau'', \circ)]}\,(closeBalanceMems)$$

$$\frac{}{\tau \;\leq\; [\mathsf{m} : (\tau', \_)],\; \tau \;\leq\; [\mathsf{m} : (\tau'', \_)] \;\longrightarrow\; \tau' \;\leq\; \tau'',\; \tau'' \;\leq\; \tau'}\,(closeCong)$$

$$\frac{}{\left.\begin{array}{l}\tau \;\leq\; (\tau_1 \times \tau_2 \to \tau_3), \\ \tau \;\leq\; (\tau'_1 \times \tau'_2 \to \tau'_3)\end{array}\right\} \;\longrightarrow\; \left\{\begin{array}{l}\tau'_1 \;\leq\; \tau_1, \tau_1 \;\leq\; \tau'_1, \\ \tau'_2 \;\leq\; \tau_2, \tau_2 \;\leq\; \tau'_2, \\ \tau_3 \;\leq\; \tau'_3, \tau'_3 \;\leq\; \tau_3\end{array}\right.}\,(closeCongFunc)$$

Figure 5.8.: Constraint Closure

```
x = 2;
x.m = 10;
```

The constraints generated include:

$$\begin{aligned} [\![\texttt{x\_1}]\!] &\leq \mathsf{Int} \\ [\![\texttt{x\_1}]\!] &\leq [\texttt{m} : ([\![\texttt{x\_1.m}]\!], \circ)] \\ &\ldots \end{aligned}$$

The above constraints violate $(wlfMix1)$ and $(wlfMix3)$. In other words, type variable $[\![\texttt{x\_1}]\!]$ is expected to be an integer *and* an object type with member m. Naturally, one can see from the program text that variable x is being used in two conflicting ways. We shall see in Chapter 6 that the implementation will highlight such programming errors.

Note that rule $(wlfMix1)$ also covers constraints of the form $\tau^{\mathbf{obj}}$. Constraints for τ must be of the form $\tau \leq [\mathsf{m} : (\_, \_)]$ to ensure an object type. If there are no constraints for τ, as we shall see in Section 5.7, we will default to creating an empty object type; thus, satisfying the requirement that τ is an object type.

Recall that in Chapter 1 we introduced an error into function `checkform`, in Figure 1.1, by misspelling the access to member `submit` of the parameter `theform`: `theform.submi.disabled = 1`. We now show how this mistake manifests itself in the constraint system and is *picked up* by our well-formed constraints rules. We continue from the example constraints given in Section 5.3. With pre-environment $\gamma_4 = \{\texttt{theform} : \texttt{checkform}, \texttt{lab} : ......\}$ constraint generation for the erroneous function, `checkform`, will produce constraints:

$$\begin{aligned} \mathsf{C}_4 \quad = \quad \{ \; & [\![\texttt{theform\_checkForm}]\!] \leq [\texttt{submi} : ([\![\texttt{theform\_checkForm.submi}]\!], \bullet)], \\ & ...\} \end{aligned}$$

Through the closure process, constraints will be added that *link* together the receiver in `form` and `onSubmit` and the parameter of `checkform`. This will cause member constraints (that will infer members) to be added to the receiver of `form`.

$$\frac{\begin{array}{l}\vdash \mathsf{C} \diamond_{cl} \\ \mathsf{C} = \{\mathsf{c}_1 ... \mathsf{c}_n\} \\ \mathsf{C} \vdash \mathsf{c}_i \quad \forall\, i \in 1...n\end{array}}{\vdash \mathsf{C}\diamond}\ (wlfAll)$$

$$\frac{\tau \le [\mathsf{m} : (\_\,, \bullet)] \notin \mathsf{C}}{\mathsf{C} \vdash \tau^{\circ}}\ (wlfNoDefs) \qquad \frac{\tau' \le [\mathsf{m} : (\_\,,\_\,)] \in \mathsf{C}}{\mathsf{C} \vdash \tau \lhd_{\mathsf{m}} \tau'}\ (wlfEnsureObj)$$

$$\frac{\tau \le (\_ \times \_ \to \_\,) \notin \mathsf{C} \wedge \tau \le \mathsf{Int} \notin \mathsf{C}}{\begin{array}{l}\mathsf{C} \vdash \tau \le [\mathsf{m} : (\tau', \psi)] \\ \mathsf{C} \vdash \tau^{\mathbf{obj}}\end{array}}\ (wlfMix1)$$

$$\frac{\tau \le [\mathsf{m} : (\_\,,\_\,)] \notin \mathsf{C} \wedge \tau \le \mathsf{Int} \notin \mathsf{C}}{\mathsf{C} \vdash \tau \le (\tau_1 \times \tau_2 \to \tau_3)}\ (wlfMix2)$$

$$\frac{\tau \le (\_ \times \_ \to \_\,) \in \mathsf{C} \wedge \tau \le [\mathsf{m} : (\_\,,\_\,)] \in \mathsf{C}}{\mathsf{C} \vdash \tau \le \mathsf{Int}}\ (wlfMix3)$$

Figure 5.9.: Well-formed Constraints

In particular, the constraint detailing the erroneous member `submi`:

$$\begin{aligned}\mathsf{C} \;=\; &\{[\![\texttt{this\_form}]\!] \le [\texttt{onSubmit} : ([\![\texttt{this\_1.onSubmit}]\!], \circ)],\\ &[\![\texttt{this\_form}]\!] \le [\texttt{submit} : ([\![\texttt{this\_2.submit}]\!], \circ)],\\ &[\![\texttt{this\_form}]\!] \le [\texttt{submi} : ([\![\texttt{this\_2.submi}]\!], \bullet)]\\ &...\}\end{aligned}$$

We see that constraints $[\![\texttt{this\_form}]\!] \le [\texttt{submi} : ([\![\texttt{this\_2.submi}]\!], \bullet)]$ and $[\![\texttt{this\_form}]\!]^{\circ}$, violate well-formedness rule $(wlfNoDefs)$.

5.7 From Constraints to Types

We now show how well-formed constraints, $\vdash \mathsf{C} \diamond$, can be used to generate a solution. We first define a *type variable function*, V, which maps type variables, $\tau_1, \tau_2 ...$, to variables in the type system, $\alpha_1, \alpha_2, ...$ We denote the empty type variable mapping by $\mathsf{V}_{\emptyset}$.

In this section, we shall be referring to two sets of type variables: those from the type system, $\alpha, \alpha' ...$, and those used in type inference, $\tau, \tau' ...$ Therefore, to avoid confusion, we shall refer to the type system type variables as *variables* and to the type inference variables as *type variables*.

A type variable mapping is well-formed, $\mathsf{C} \vdash \mathsf{V}\diamond$, see Definition 7, if equivalent type variables, $\tau \le \tau', \tau' \le \tau \in \mathsf{C}$, are mapped to the same variable. Note that a well-formed type variable mapping is undefined for type variables representing integer and function types.

Definition 7 *A type variable function* V, *is well-formed for* C, *i.e.* $\mathsf{C} \vdash \mathsf{V}\diamond$, *iff:*

- $\tau, \tau' \in dom(\mathsf{V}) : \tau \le \tau', \tau' \le \tau \in \mathsf{C} \implies \mathsf{V}(\tau) = \mathsf{V}(\tau')$
- $\tau \le \tau', \tau' \le \tau \in \mathsf{C} \wedge \mathsf{V}(\tau) = \mathcal{U}df \implies \mathsf{V}(\tau') = \mathcal{U}df$
- $\tau \le \mathsf{Int} \in \mathsf{C}$ *or* $\tau \le (\_ \times \_ \to \_) \in \mathsf{C} \implies \mathsf{V}(\tau) = \mathcal{U}df$

The translation relation, $\mathsf{C}, \mathsf{V}, \tau \to \mathtt{tp}$, defined in Figure 5.10, translates a type variable, τ, into a type. If a type variable has no constraints indicating whether it should be an object, function or integer type we default to making it an object type with no members. The extension of V by a type variable and variable, denoted by $\mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha)$, is defined as follows:

$$\frac{\mathsf{V}(\tau) = \alpha}{\mathsf{C}, \mathsf{V}, \tau \to \alpha}\,(var) \qquad \frac{\tau \ \le \ \mathsf{Int} \in \mathsf{C}}{\mathsf{C}, \mathsf{V}, \tau \to \mathsf{Int}}\,(int) \qquad \frac{\begin{array}{l}\tau \ \le \ [\mathsf{m} : \_\,] \notin \mathsf{C} \\ \tau \ \le \ (\_ \times \_ \to \_) \notin \mathsf{C} \\ \tau \ \le \ \mathsf{Int} \notin \mathsf{C}\end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.[\,]}\,(unknown)$$

$$\frac{\begin{array}{l} n \geq 1 \\ \{\mathsf{m}_1 ... \mathsf{m}_n\} = \{\mathsf{m} \mid \tau \ \le \ [\mathsf{m} : (\_, \_)] \ \in \ \mathsf{C}\ \} \\ \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha) \ \ (\alpha \notin Range(\mathsf{V})) \\ \tau \ \le \ [\mathsf{m}_i : (\tau_i, \_)] \ \in \ \mathsf{C}\ (for\ i \ \in 1...n) \\ \mathsf{C}, \mathsf{V}_0, \tau_i \to \mathsf{tp}_i \\ \psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.[\mathsf{m_1} : (\mathsf{tp}_1, \psi_1) ... \mathsf{m_n} : (\mathsf{tp}_\mathtt{n}, \psi_\mathtt{n})]}\,(objtype)$$

$$\frac{\begin{array}{l} \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha) \ \ (\alpha \notin Range(\mathsf{V})) \\ \tau \ \le \ (\tau_1 \times \tau_2 \to \tau_3) \in \mathsf{C} \\ \mathsf{C}, \mathsf{V}_0, \tau_i \to \mathsf{tp}_i \quad (for\ i \ \in 1...3) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.(\mathsf{tp}_1 \times \mathsf{tp}_2 \to \mathsf{tp}_3)}\,(functype)$$

Figure 5.10.: Translation from Type Variables to Types

$$\mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha) = \begin{cases} \perp & \text{if } \mathsf{V}(\tau) \neq \mathcal{U}\mathit{df} \text{ or} \\ & \quad \exists\, \tau' :\ \mathsf{V}(\tau') = \alpha \\ \mathsf{V} \cup \{(\tau', \alpha) \mid \tau \ \leq\ \tau', \tau' \ \leq\ \tau \in \mathsf{C}\} & \text{otherwise} \end{cases}$$

$\mathsf{V} \bullet_{\mathsf{C}}$ (τ,α) extends V so that τ and all equivalent type variables map to α. If V is already defined for a type variable, or there is another type variable that already maps to α, then the result is $\perp$, *i.e.* the extension is invalid.

Because each step of the translation either extends V, or finishes when $\mathsf{V}(\tau) = \alpha$ (or $\mathsf{t} = \mathsf{Int}$ or $\mathsf{t} = \mu\, \alpha.[\,]$), termination is guaranteed.

We define function $\mathcal{A}(\mathsf{C}, \tau, \mathsf{m})$, see definition below, which determines the annotation that any solution satisfying C should give to m in τ. This is achieved, by looking for constraints detailing whether m should be definite or potential for τ, *i.e.* constraints of the form: $\tau \ \leq\ [\mathsf{m} : (\_\,,\_)]$.

$$\mathcal{A}(\mathsf{C}, \tau, \mathsf{m}) \quad = \quad \begin{cases} \bullet & \text{if } \tau \ \leq\ [\mathsf{m} : (\_\,,\bullet)] \ \in\ \mathsf{C} \\ \circ & \text{if } \tau \ \leq\ [\mathsf{m} : (\_\,,\circ)] \ \in\ \mathsf{C} \text{ and } \tau \ \leq\ [\mathsf{m} : (\_\,,\bullet)] \ \notin\ \mathsf{C} \\ \mathcal{U}\mathit{df} & \text{otherwise} \end{cases}$$

Proposition 5 *For constraints* C, *type variables* τ, τ', *annotations* ψ, ψ', *and member* m*:*

1. $\vdash \mathsf{C}\diamond,\ \ \tau \ \leq\ \tau', \tau' \ \leq\ \tau \in \mathsf{C} \quad \Longrightarrow \quad \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}) = \mathcal{A}(\mathsf{C}, \tau', \mathsf{m})$
2. $\vdash \mathsf{C}\diamond,\ \ \tau \ \leq\ \tau' \in \mathsf{C} \quad \Longrightarrow \quad \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}) \leq \mathcal{A}(\mathsf{C}, \tau', \mathsf{m})$

Figure 5.11 gives some examples of types produced by the translation relation.

Lemma 4 states that the translation of a type variable to a type t does not introduce any free variables; the only free variables possible in t are those that are in the range of V. The definition of free variables for types is given in Figure 4.2. Returning to the examples of Figure 5.11 we have that:

$$\begin{array}{lclclcl} \mathit{Range}(\mathsf{V}_1) & = & \emptyset & & \mathcal{FV}(\mathsf{tp}_1) & = & \emptyset \\ \mathit{Range}(\mathsf{V}_2) & = & \{\alpha\} & & \mathcal{FV}(\mathsf{tp}_2) & = & \{\alpha'\} \\ \mathit{Range}(\mathsf{V}_3) & = & \{\alpha, \alpha'\} & & \mathcal{FV}(\mathsf{tp}_2) & = & \{\alpha\} \end{array}$$

Lemma 4 *For constraints* C, *type variable function* V, *and type variable* τ*if:*

$$\begin{array}{lcl}
\mathsf{C} & = & \{\tau \leq [\mathsf{m} : (\tau', \bullet)],\ \tau' \leq [\mathsf{m} : (\tau'', \bullet)], \tau'' \leq \mathsf{Int}\} \\
\\
\mathsf{V}_1 & = & \mathsf{V}_\emptyset \\
\mathsf{V}_2 & = & \{\tau' \mapsto \alpha'\} \\
\mathsf{V}_3 & = & \{\tau \mapsto \alpha,\ \tau' \mapsto \alpha'\} \\
\\
 & & \mathsf{C}, \mathsf{V}_1, \tau \rightarrow \mathsf{tp}_1 \\
 & & \mathsf{C}, \mathsf{V}_2, \tau \rightarrow \mathsf{tp}_2 \\
 & & \mathsf{C}, \mathsf{V}_3, \tau \rightarrow \mathsf{tp}_3 \\
\\
\mathsf{tp}_1 & = & \mu\, \alpha.[\mathsf{m} : (\mu\, \alpha'.[\mathsf{m} : (\mathsf{Int}, \bullet)], \bullet)] \\
\mathsf{tp}_2 & = & \mu\, \alpha.[\mathsf{m} : (\alpha', \bullet)] \\
\mathsf{tp}_3 & = & \alpha
\end{array}$$

Figure 5.11.: Examples of Types Produced by the Translation Relation

$$\mathsf{C}, \mathsf{V}, \tau \rightarrow \mathsf{tp} \tag{1}$$

then

$$\mathcal{FV}(\mathsf{tp}) \subseteq \mathit{Range}(\mathsf{V}) \tag{2}$$

Proof: We proceed by induction on the derivation $\mathsf{C}, \mathsf{V}, \tau \rightarrow \mathsf{tp}$.

Case: (var)

Given that the last rule applied was (var) with (1) we get:

$$\mathsf{V}(\tau) = \alpha \tag{3}$$

$$\mathsf{tp} = \alpha \tag{4}$$

From (4) with the definition of $\mathcal{FV}$ we get:

$$\mathcal{FV}(\mathsf{tp}) = \{\alpha\} \tag{5}$$

From (5) and (3) we get:

$$\mathcal{FV}(\mathsf{tp}) \subseteq \mathit{Range}(\mathsf{V}) \tag{6}$$

We see that (6) establishes (2).

Case: $(\mathit{unknown})$

Given that the last rule applied was $(\mathit{unknown})$ with (1) we get:

$$\mathsf{tp} = \mu\, \alpha.[\,] \tag{7}$$

From (7) with the definition of $\mathcal{FV}$ we get:

$$\mathcal{FV}(\mathsf{tp}) = \emptyset \tag{8}$$

From (8) we get:

$$\mathcal{FV}(\mathsf{tp}) \subseteq \mathit{Range}(\mathsf{V}) \tag{9}$$

We see that (9) establishes (2).

Case: $(\mathit{objtype})$

Given that the last rule applied was $(\mathit{objtype})$ with (1) we get that there exist suitable α, n, $\tau_1 ..., \tau_n, \mathsf{tp}_1 ... \mathsf{tp}_n, \psi_1 ... \psi_n$, and $\mathsf{m}_1 ... \mathsf{m}_n$ so that:

$$n \geq 1 \tag{10}$$
$$\{\mathsf{m}_1 ... \mathsf{m}_n\} = \{\mathsf{m} \mid \tau \ \leq \ [\mathsf{m} : (\_, \_)] \ \in \ \mathsf{C}\ \} \tag{11}$$
$$\mathsf{V}(\tau) = \mathcal{U}\mathit{df} \tag{12}$$
$$\mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha) \ \ (\alpha \notin \mathit{Range}(\mathsf{V})) \tag{13}$$
$$\tau \ \leq \ [\mathsf{m}_i : (\tau_i, \_)] \ \in \ \mathsf{C}\ (\mathit{for}\ i \ \in 1...n) \tag{14}$$
$$\mathsf{C}, \mathsf{V}_0, \tau_i \rightarrow \mathsf{tp}_i \tag{15}$$
$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{16}$$
$$\mathsf{tp} = \mu\, \alpha.[\mathtt{m_1} : (\mathtt{tp_1}, \psi_1) ... \mathtt{m_n} : (\mathtt{tp_n}, \psi_\mathtt{n})] \tag{17}$$

With (15) we apply the inductive hypothesis n times to get (for $i \ \in \ 1...n$):

$$\mathcal{FV}(\mathsf{tp}_i) \subseteq \mathit{Range}(\mathsf{V}_0) \tag{18}$$

From (18) we get:

$$\bigcup_{i\in 1\ldots n} \mathcal{FV}(\mathsf{tp}_i) \subseteq \mathit{Range}(\mathsf{V}_0) \tag{19}$$

From (19) we get:

$$(\bigcup_{i\in 1\ldots n} \mathcal{FV}(\mathsf{tp}_i)) \setminus \{\alpha\} \subseteq \mathit{Range}(\mathsf{V}_0) \setminus \{\alpha\} \tag{20}$$

From (13) with the definition of extension we get:

$$\mathit{Range}(\mathsf{V}) = \mathit{Range}(\mathsf{V}_0) \setminus \{\alpha\} \tag{21}$$

From (20) and (21) we get:

$$\bigcup_{i\in 1\ldots n} \mathcal{FV}(\mathsf{tp}_i) \setminus \{\alpha\} \subseteq \mathit{Range}(\mathsf{V}) \tag{22}$$

From (17), (22) and the definition of $\mathcal{FV}$ we get:

$$\mathcal{FV}(\mathsf{tp}) \subseteq \mathit{Range}(\mathsf{V}) \tag{23}$$

We see that (23) establishes (2).

Case: $(\mathit{functype})$

Similar to previous case $(\mathit{objtype})$.

□

Theorem 4 asserts that the translation from type variables to types, starting with an empty type variable function, $\mathsf{V}_\emptyset$, produces well-formed types.

Theorem 4 *For constraints* C *and type variable* τ*, if:*

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} \tag*{(1)}$$

then:

$$\vdash \mathsf{t} \diamond \tag*{(2)}$$

Proof: For integers, Int, the proof is immediate. For object types Lemma 4 asserts that there are no free variables in a type produced from the empty type variable function. The uniqueness of members is guaranteed through $\{\mathsf{m}_1...\mathsf{m}_n\} = \{\mathsf{m} \mid \tau \leq [\mathsf{m} : (\_, \_)] \in \mathsf{C}\}$ in rule $(objtype)$. Similar reasoning applies to function types. $\square$

5.8 Satisfaction of Well-formed Constraints

In this section we conjecture that well-formed constraints can be used to create a solution that will satisfy those constraints. We first define and prove some auxiliary properties.

5.8.1 Variable Renamings

An important notion throughout this section will be the equivalence of types up to a renaming function. We define a variable renaming, β:*ObjVar* → *ObjVar*, as a bijection between type variables.

Definition 8 *We say that β extends β', denoted by β ext β', if $\beta'(\tau) \neq Udf$ then $\beta(\tau) = \beta'(\tau)$.*

The translation relation, in the cases $(objtype)$ and $(functype)$, extends the type variable function, $\mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha)$. The choice of variable, α, is made non-deterministically; we only require that $\alpha \notin \mathsf{V}$. Therefore, we must prove that two runs of the translation on the same (or equivalent) type variable produce the same type up to the renaming of variables.

With $\mathsf{tp}[\beta]$, see Definition 9, we denote the substitution of a type variable mapping to a type. Note that the substitution only replaces free variables. Types tp and tp' are congruent up to renaming, $\mathsf{tp} \equiv_\beta \mathsf{tp}'$, see Definition 10, if β applied to tp is congruent to tp'.

Definition 9 *For type tp, variable renaming β, we define the application of a variable renaming to a type as: $\mathsf{tp}[\beta] = \mathsf{tp}[\alpha_1/\alpha'_1]...[\alpha_n/\alpha'_n]$ where $\{(\alpha_1, \alpha'_1)...(\alpha_n, \alpha'_n)\} = \{(\alpha, \alpha') \mid \beta(\alpha) = \alpha'\}$.*

Definition 10 *For types $\mathsf{tp}, \mathsf{tp}'$, variable renaming β:*

1. $\mathsf{tp} \equiv_\beta \mathsf{tp}' \iff \mathsf{tp}[\beta] \equiv \mathsf{tp}'$
2. $(\mathsf{tp}, \psi) \equiv_\beta (\mathsf{tp}', \psi) \iff (\mathsf{tp}, \psi)[\beta] \equiv (\mathsf{tp}', \psi)$

Consider the following examples, where $\beta(\alpha_1) = \alpha_2$ and $\beta(\alpha_3) = \alpha_4$:

$$\alpha_1 \equiv_\beta \alpha_2$$
$$\mu\, \alpha_1.[\mathsf{m}_1 : (\alpha_1, \bullet), \mathsf{m}_2 : (\alpha_3, \bullet)] \equiv_\beta \mu\, \alpha_1.[\mathsf{m}_1 : (\alpha_1, \bullet), \mathsf{m}_2 : (\alpha_4, \bullet)]$$

Definition 11 *Two type variable functions* V *and* V' *agree with respect to a variable renaming,* $\mathsf{C} \vdash \mathsf{V} \simeq_\beta \mathsf{V}'$ *iff* $\mathsf{C} \vdash \mathsf{V}\diamond$ *and* $\mathsf{C} \vdash \mathsf{V}'\diamond$ *and* $dom(\beta) \subseteq Range(\mathsf{V})$ *and for all* τ *:* $\beta(\mathsf{V}(\tau)) = \mathsf{V}'(\tau)$*.*

Proposition 6 *For constraints* C*, type variables mappings* V*,* V*, and type variable* τ*:*

1. $\vdash \mathsf{C}\diamond,\ \mathsf{C} \vdash \mathsf{V}\diamond,\ \tau\ \leq\ [\mathsf{m} : (\_,\_)]\ \in\ \mathsf{C},\ \mathsf{V}' = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha)\ \implies\ \mathsf{C} \vdash \mathsf{V}'\diamond$

Proof: Follows from the definition of extension of a type variable function and the well-formed constraints. □

Proposition 7 asserts that two types composed of members that are equivalent up to renaming are themselves equivalent up to renaming.

Proposition 7 *For variables renamings* β*,* β'*, types* $\mathsf{tp}_1 ... \mathsf{tp}_n, \mathsf{tp}'_1 ... \mathsf{tp}'_n$*, annotations* $\psi_1 ... \psi_n, \psi'_1, \psi'_n$*, and variables* α*,* α'*, if:*

$$\beta' = \beta, \alpha \mapsto \alpha' \tag{1}$$
$$(\mathsf{tp}_1, \psi_1) \equiv_{\beta'} (\mathsf{tp}'_1, \psi'_1)\ ...\ (\mathsf{tp}_n, \psi_n) \equiv_{\beta'} (\mathsf{tp}'_n, \psi'_n) \tag{2}$$
$$\mathsf{t} = \mu\ \alpha.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1) ... \mathsf{m}_n : (\mathsf{tp}_n, \psi_n)] \tag{3}$$
$$\mathsf{t}' = \mu\ \alpha'.[\mathsf{m}_1 : (\mathsf{tp}'_1, \psi'_1) ... \mathsf{m}_n : (\mathsf{tp}'_n, \psi'_n)] \tag{4}$$

then:

$$\mathsf{t} \equiv_\beta \mathsf{t}' \tag{5}$$

Proof: From (3) with β let:

$$\mathsf{t}'' = \mathsf{t}[\beta] \tag{6}$$

From (6), (3) and the definition of substitution we get:

$$\mathsf{t}'' = \mu\ \alpha.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1)[\beta] ... \mathsf{m}_n : (\mathsf{tp}_n, \psi_n)[\beta]] \tag{7}$$

From (1) and (7) we apply congruence rule $(\alpha Conv)$ with $[\alpha/\alpha']$ to get t''' such that:

$$\mathsf{t}'' \equiv \mathsf{t}''' \tag{8}$$
$$\mathsf{t}''' = \mu\ \alpha'.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1)[\beta][\alpha/\alpha'] ... \mathsf{m}_n : (\mathsf{tp}_n, \psi_n)[\beta][\alpha/\alpha']] \tag{9}$$

From (1), (2) and Definition 10.2 we get (for $i \in 1...n$):

$$(\mathsf{tp}_i, \psi_i)[\beta][\alpha/\alpha'] \equiv (\mathsf{tp}'_i, \psi'_i)[\beta][\alpha/\alpha'] \tag{10}$$

From (9), (4), (10) and congruence rule $(\equiv reorder)$, from Figure 4.5, we get:

$$\mathsf{t}''' \equiv \mathsf{t}' \qquad (11)$$

With (11), (6), (8), Definition 10.1, and the definition of congruence (noting that α is bound in t and α' is bound in t$'$) we get:

$$\mathsf{t} \equiv_{\beta} \mathsf{t}' \qquad (12)$$

We see that (12) establishes (4).

□

Proposition 8 states Proposition 7 for function types.

Proposition 8 *For variables renamings β, β', object types O, O', types $\mathsf{tp}_2, \mathsf{tp}_3, \mathsf{tp}_2', \mathsf{tp}_3'$, and variables α, α', if:*

$$\beta' = \beta, \alpha \mapsto \alpha' \qquad (1)$$
$$\mathsf{O}_1 \equiv_{\beta'} \mathsf{O}_1',\ \mathsf{tp}_2 \equiv_{\beta'} \mathsf{tp}_2',\ \mathsf{tp}_3 \equiv_{\beta'} \mathsf{tp}_3' \qquad (2)$$
$$\mathsf{t} = \mu\, \alpha.(\mathsf{O}_1 \times \mathsf{tp}_2 \to \mathsf{tp}_3) \qquad (3)$$
$$\mathsf{t}' = \mu\, \alpha'.(\mathsf{O}_1' \times \mathsf{tp}_2' \to \mathsf{tp}_3') \qquad (4)$$

then:

$$\mathsf{t} \equiv_{\beta} \mathsf{t}' \qquad (5)$$

Proof: Proof similar to Proposition 7. □

5.8.2 Determinism of the Translation

We now show that the translation relation, $\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tp}$, is deterministic; two equivalent type variables will translate to types that are congruent up to renaming. We prove this property with Lemma 5.

Lemma 5 *For constraints C, type variables τ, τ', and type variable functions V, V', if:*

$$\vdash \mathsf{C}\diamond \qquad (1)$$
$$\tau \leq \tau', \tau' \leq \tau \in \mathsf{C} \qquad (2)$$
$$\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tp} \qquad (3)$$
$$\mathsf{C}, \mathsf{V}', \tau' \to \mathsf{tp}' \qquad (4)$$
$$\mathsf{C} \vdash \mathsf{V} \simeq_{\beta} \mathsf{V}' \qquad (5)$$

then:

$$\mathsf{tp} \equiv_\beta \mathsf{tp}' \tag{6}$$

Proof: We proceed by induction on the derivation $\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tp}$.

Case: (var)

Given that the last rule applied was (var) with (3) we get:

$$\mathsf{V}(\tau) = \alpha \tag{7}$$
$$\mathsf{tp} = \alpha \tag{8}$$

From (5) with Definition 11 we get:

$$\mathsf{C} \vdash \mathsf{V}\diamond \tag{9}$$
$$\mathsf{C} \vdash \mathsf{V}'\diamond \tag{10}$$
$$\forall\, \tau : \beta(\mathsf{V}(\tau)) = \mathsf{V}'(\tau) \tag{11}$$

From (2), (9) and Definition 7 we get:

$$\mathsf{V}(\tau) = \mathsf{V}(\tau') \tag{12}$$

With (11) and (12) we get:

$$\beta(\mathsf{V}(\tau)) = \beta(\mathsf{V}(\tau')) = \mathsf{V}'(\tau') \tag{13}$$

From (13) we have that $\mathsf{V}'(\tau') \neq \mathcal{U}df$; therefore, with (10) the last rule applied to (4) was (var) and so we get for some α':

$$\mathsf{V}'(\tau') = \alpha' \tag{14}$$
$$\mathsf{tp}' = \alpha' \tag{15}$$

From (13), (7), (8),(14), (15) and Definition 10.1 we establish (6).

Case: (int)

Given that the last rule applied was (int) with (3) we get:

$$\tau \;\leq\; \mathsf{Int} \in \mathsf{C} \tag{16}$$
$$\mathsf{tp} = \mathsf{Int} \tag{17}$$

From (1) (the constraints are closed), (2), (16) and closure rule $(closeBalance)$ we get:

$$\tau' \ \leq \ \mathsf{Int} \in \mathsf{C} \tag{18}$$

Because the constraints are well-formed, (1), there are no other constraints in C for τ' *c.f.* $(wlfMix3)$. From (5) we get that $\mathsf{C} \vdash \mathsf{V}' \diamond$ and with (18) implies $\mathsf{V}'(\tau') = \mathcal{U}df$. Therefore, the last rule applied to (4) was (int):

$$\mathsf{tp}' = \mathsf{Int} \tag{19}$$

Therefore, we have that $\mathsf{tp} \equiv \mathsf{tp}'$ which establishes (6).

Case: $(objtype)$

Given that the last rule applied was $(objtype)$ with (3) we get that there exist suitable α, n, $\tau_1 ..., \tau_n, \mathsf{tp}_1 ... \mathsf{tp}_n, \psi_1 ... \psi_n$, and $\mathsf{m}_1 ... \mathsf{m}_n$ so that:

$$n \geq 1 \tag{20}$$

$$\{\mathsf{m}_1 ... \mathsf{m}_n\} = \{\mathsf{m} \mid \tau \ \leq \ [\mathsf{m} : (\_, \_)] \ \in \ \mathsf{C}\ \} \tag{21}$$

$$\mathsf{V}(\tau) = \mathcal{U}df \tag{22}$$

$$\mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha) \ \ (\alpha \notin Range(\mathsf{V})) \tag{23}$$

$$\tau \ \leq \ [\mathsf{m}_i : (\tau_i, \_)] \ \in \ \mathsf{C}\ (for\ i \ \in 1...n) \tag{24}$$

$$\mathsf{C}, \mathsf{V}_0, \tau_i \to \mathsf{tp}_i \tag{25}$$

$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{26}$$

$$\mathsf{tp} = \mu\ \alpha.[\mathtt{m_1} : (\mathtt{tp_1}, \psi_1) ... \mathtt{m_n} : (\mathtt{tp_n}, \psi_\mathtt{n})] \tag{27}$$

$$\mathsf{V}'' = \mathsf{V}_n \tag{28}$$

From (5) with Definition 11 we get:

$$\mathsf{C} \vdash \mathsf{V}\diamond \tag{29}$$

$$\mathsf{C} \vdash \mathsf{V}'\diamond \tag{30}$$

$$dom(\beta) \subseteq Range(\mathsf{V}') \tag{31}$$

$$\forall\, \tau \ : \beta(\mathsf{V}(\tau)) = \mathsf{V}'(\tau) \tag{32}$$

From (1) (the constraints are closed), (2), (20), (24) and closure rule $(closeTrans)$ there exists at least one constraint such that: $\tau' \ \leq \ [\mathsf{m}_i : (\tau_i', \_)] \ \in \ \mathsf{C}$ (for some τ_i'). With (1) and well-formed rule $(wlfMix1)$ we have that $\tau \ \leq \ (\_ \times \_ \rightarrow \_) \notin \mathsf{C}$, $\tau \ \leq \ \mathsf{Int} \notin \mathsf{C}$. With (2), (5) and (22) we have that $\mathsf{V}'(\tau') = \mathcal{U}\!df$. Therefore, $(objtype)$ was the last rule applied to (4) and there exist suitable α', n', $\tau'_1 ..., \tau'_{n'}, \mathsf{tp}'_1 ... \mathsf{tp}'_{n'}, \psi'_1 ... \psi'_{n'}$, and $\mathsf{m}'_1 ... \mathsf{m}'_{n'}$ so that:

$$n' \geq 1 \tag{33}$$

$$\{\mathsf{m}'_1 ... \mathsf{m}'_n\} = \{\mathsf{m} \mid \tau' \ \leq \ [\mathsf{m} : (\_, \_)] \ \in \ \mathsf{C}\ \} \tag{34}$$

$$\mathsf{V}'(\tau') = \mathcal{U}\!df \tag{35}$$

$$\mathsf{V}'_0 = \mathsf{V}' \bullet_{\mathsf{C}} (\tau', \alpha') \ \ (\alpha' \notin Range(\mathsf{V}')) \tag{36}$$

$$\tau' \ \leq \ [\mathsf{m}_i : (\tau'_i, \_)] \ \in \ \mathsf{C}\ (for\ i \ \in 1...n') \tag{37}$$

$$\mathsf{C}, \mathsf{V}'_0, \tau'_i \rightarrow \mathsf{tp}'_i \tag{38}$$

$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{39}$$

$$\mathsf{tp}' = \mu\, \alpha'.[\mathtt{m_1} : (\mathsf{tp}'_1, \psi'_1) ... \mathtt{m_n} : (\mathsf{tp}'_{\mathtt{n}}, \psi'_{\mathtt{n}})] \tag{40}$$

$$\mathsf{V}''' = \mathsf{V}'_{n'} \tag{41}$$

From (1), (2) and closure rule $(closeTrans)$ we get:

$$\tau \ \leq \ [\mathsf{m} : (\tau'', \psi)] \ \in \ \mathsf{C} \implies \tau' \ \leq \ [\mathsf{m} : (\tau'', \psi)] \ \in \ \mathsf{C} \tag{42}$$

$$\tau' \ \leq \ [\mathsf{m} : (\tau'', \psi)] \ \in \ \mathsf{C} \implies \tau \ \leq \ [\mathsf{m} : (\tau'', \psi)] \ \in \ \mathsf{C} \tag{43}$$

From (42), (43), (21) and (34) we have that:

$$n = n' \tag{44}$$

$$\{\mathsf{m}_1 ... \mathsf{m}_n\} = \{\mathsf{m}'_1 ... \mathsf{m}'_n\} \tag{45}$$

For simplicity, let us assume that $\mathsf{m}_i = \mathsf{m}'_i$ (for all $i \ \in 1..n$). From (24) and (37) choose constraints $i \in 1...n$:

$$\tau \ \leq \ [\mathsf{m}_i : (\tau_i, \psi_i)] \ \in \ \mathsf{C} \tag{46}$$

$$\tau' \ \leq \ [\mathsf{m}_i : (\tau'_i, \psi'_i)] \ \in \ \mathsf{C} \tag{47}$$

From (46) with (42) we get (for $i \in 1...n$):

$$\tau' \ \leq \ [\mathsf{m}_i : (\tau_i, \psi_i)] \ \in \ \mathsf{C} \tag{48}$$

From (1) (the constraint are closed), (47), (48) and closure rule $(closeCong)$ we get (for $i \in \ 1...n$):

$$\tau_i \ \leq \ \tau'_i, \tau'_i \ \leq \ \tau_i \in \mathsf{C} \tag{49}$$

From (1), (29), (46), (23) and Proposition 6 we have:

$$\mathsf{C} \vdash \mathsf{V}_0\diamond \tag{50}$$

Similarly, from (1), (30), (47), (36) and Proposition 6 we have:

$$\mathsf{C} \vdash \mathsf{V}'_0\diamond \tag{51}$$

From the definition of extension, (23) and (36) we know that $\alpha \notin dom(\mathsf{V})$ and $\alpha' \notin dom(\mathsf{V}')$. Furthermore, from (31) we know that α' is not in the range of β. Therefore, let β_0 be the extension of β, $\beta_0\ ext\ \beta$ where:

$$\beta_0(\alpha) = \alpha' \tag{52}$$

Therefore, with (50), (51), (32) and (52) we get:

$$\mathsf{C} \vdash \mathsf{V}_0 \simeq_{\beta_0} \mathsf{V}'_0 \tag{53}$$

From (1), (49), (25), (38) and (53) we apply the inductive hypothesis n times to get (for $i\ \in 1...n$):

$$\mathsf{tp}_i \equiv_{\beta_0} \mathsf{tp}'_i \tag{54}$$

From (1), (49), (26), (39) and Proposition 5.1 we have that (for $i = 1...n$):

$$\mathcal{A}(\mathsf{C}, \tau', \mathsf{m}_i) = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{55}$$

From (52), (54), (27), (40), (55) and Proposition 7 we get:

$$\mathsf{tp} \equiv \mathsf{tp}' \tag{56}$$

We see that (56) establishes (6).

Case: $(functype)$

Similar to case $(objtype)$

Case: $(unknown)$

Straightforward given that the empty object is congruent to the empty object.

□

Lemma 6 asserts the determinism of the translation relation when starting with the empty type variable function.

Lemma 6 *For constraints* C *and type variables* τ, τ'*, if:*

$$\vdash \mathsf{C}\diamond \qquad (1)$$
$$\tau \leq \tau', \tau' \leq \tau \in \mathsf{C} \qquad (2)$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} \qquad (3)$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau' \rightarrow \mathsf{t}' \qquad (4)$$

then:

$$\mathsf{t} \equiv \mathsf{t}' \qquad (5)$$

Proof: Follows from Lemma 5 noting that $\mathsf{C} \vdash \mathsf{V}_\emptyset \simeq_\emptyset \mathsf{V}_\emptyset$ and $\mathsf{t} \equiv_\emptyset \mathsf{t}' \implies \mathsf{t} \equiv \mathsf{t}'$. Remember that we denote the empty variable renaming by $\emptyset$. □

5.8.3 Generating Solutions

A *generated solution*, S_C, see Definition 12, is a solution that maps each type variable in a well-formed set of constraints to the type produced by the translation relation.

Definition 12 *A generated solution,* S_C*, is a solution such that* $\mathsf{S}_\mathsf{C}(\tau) = \mathsf{tp}$ *iff* $\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{tp}$.

We now state some auxiliary properties that will be used in Conjecture 3. Proposition 9 asserts that a generated solution does not *mix* types that are expected, through the constraints $\tau \leq \tau' \in \mathsf{C}$, to be in the subtype relation.

Proposition 9 *For constraints* C *and type variables* τ*,* τ'*, if:*

$$\vdash \mathsf{C}\diamond \qquad (1)$$
$$\tau \leq \tau' \in \mathsf{C} \qquad (2)$$

then:

$$\mathsf{S}_\mathsf{C}(\tau) = \mu\,\alpha.[\,...\,] \textit{ and } \mathsf{S}_\mathsf{C}(\tau') = \mu\,\alpha'.[\,...\,] \textit{ or} \qquad (3)$$
$$\mathsf{S}_\mathsf{C}(\tau) = \mu\,\alpha.(\_ \times \_ \rightarrow \_) \textit{ and } \mathsf{S}_\mathsf{C}(\tau') = \mu\,\alpha'.(\_ \times \_ \rightarrow \_) \textit{ or} \qquad (4)$$
$$\mathsf{S}_\mathsf{C}(\tau) = \mathsf{Int} \textit{ and } \mathsf{S}_\mathsf{C}(\tau') = \mathsf{Int} \qquad (5)$$

Proof: Proof follows from the well-formedness rules $(wlfMix1)$, $(wlfMix2)$ and $(wlfMix3)$. □

$$\begin{array}{lcl}
\mathsf{C} & = & \{\tau \;\leq\; [\mathsf{m} : (\tau', \bullet)], \tau' \;\leq\; [\mathsf{m} : (\tau, \bullet)]\} \\
\\
 & & \mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} \\
 & & \mathsf{C}, \mathsf{V}_\emptyset, \tau' \rightarrow \mathsf{t}' \\
\\
\mathsf{t} & = & \mu\,\alpha.[\mathsf{m} : ([\mathsf{m} : (\alpha, \bullet)], \bullet)] \\
\mathsf{t}' & = & \mu\,\alpha'.[\mathsf{m} : ([\mathsf{m} : (\alpha', \bullet)], \bullet)] \\
\mathsf{t}(\mathsf{m}) & = & [\mathsf{m} : (\mathsf{t}, \bullet)]
\end{array}$$

Figure 5.12.: Example for Conjecture 1

With Conjecture 1 we assert that the translation of constraint $\tau \;\leq\; [\mathsf{m} : (\tau', \psi)] \;\in\; \mathsf{C}$ gives a type to member m (in the translation of τ) that is a subtype of the member type formed by the translation of τ' starting from the empty type variable mapping and annotation ψ. Recall that for two member types to be subtypes, $(\mathsf{t}, \psi) \leq (\mathsf{t}', \psi')$, t must be congruent to t' and $\psi \leq \psi'$. Consider the example in Figure 5.12; we can show that $\mathsf{t}(\mathsf{m})$ is congruent to t' using congruence rule $(\equiv\; fix)$, see Section 4.1.4, where t'' is $[\mathsf{m} : ([\mathsf{m} : (\alpha, \bullet)], \bullet)]$.

Conjecture 1 *For constraints* C*, type variables,* τ, τ'*, member,* m*, and annotation* ψ*, if:*

$$\begin{array}{lr}
\vdash \mathsf{C}\diamond & (1) \\
\tau \;\leq\; [\mathsf{m} : (\tau', \psi)] \;\in\; \mathsf{C} & (2) \\
\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} & (3) \\
\mathsf{C}, \mathsf{V}_\emptyset, \tau' \rightarrow \mathsf{t}' & (4)
\end{array}$$

then:

$$\mathsf{t}(\mathsf{m}) \leq (\mathsf{t}', \psi) \qquad (5)$$

Conjecture 2 is similar to Conjecture 1 but for constraints that produce function types, $\tau \;\leq\; (\tau_1 \times \tau_2 \rightarrow \tau_3) \in \mathsf{C}$. That is, the translation of τ produces a type that is a subtype of the type formed by the translation of τ_1, τ_2 and τ_3 starting from the empty type variable mapping.

Conjecture 2 *For constraints* C*, type variables,* $\tau, \tau_1, \tau_2, \tau_3$*, if:*

$$\vdash \mathsf{C}\diamond \tag{1}$$
$$\tau \;\leq\; (\tau_1 \times \tau_2 \rightarrow \tau_3) \in \mathsf{C} \tag{2}$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} \tag{3}$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau_i \rightarrow \mathsf{t}_i \;\; (\textit{for } i \;\in 1...3) \tag{4}$$

then:

$$\mathsf{t} \leq (\mathsf{t}_1 \times \mathsf{t}_2 \rightarrow \mathsf{t}_3) \tag{5}$$

Please see Appendix B for an outline of an attempt to prove Conjectures 1 and 2.

5.8.4 Generated Solution Satisfaction Conjecture

Conjecture 3 asserts that a generated solution from a well-formed set of constraints will satisfy those constraints. We outline a proof of Conjecture 3 that uses Conjectures 1 and 2. In Appendix B we outline an alternative proof attempt based on extending the translation relation.

Conjecture 3 *For constraints* C*, if:*

$$\vdash \mathsf{C}\diamond \tag{1}$$

then:

$$\mathsf{S}_\mathsf{C} \vdash \mathsf{C} \tag{2}$$

Proof: We show that $\forall \mathsf{c} \in \mathsf{C} : \mathsf{S}_\mathsf{C} \vdash \mathsf{c}$. We proceed by case analysis on the syntactic structure of c.

Case: $\mathsf{c} = \tau \;\leq\; \tau'$

We proceed by case analysis on the kind of type that $\mathsf{S}_\mathsf{C}(\tau)$ and $\mathsf{S}_\mathsf{C}(\tau')$ can be. Because of Proposition 9 with (1) and $\tau \;\leq\; \tau' \in \mathsf{C}$ there are only three cases to consider:

Case: $\begin{array}{l} \mathsf{S}_\mathsf{C}(\tau) = \mu\,\alpha.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1)...\mathsf{m}_\mathtt{n} : (\mathsf{tp}_\mathtt{p}, \psi_\mathtt{p})] \\ \mathsf{S}_\mathsf{C}(\tau') = \mu\,\alpha'.[\mathsf{m}'_1 : (\mathsf{tp}'_1, \psi'_1)...\mathsf{m}'_\mathtt{n} : (\mathsf{tp}'_\mathtt{n}, \psi'_\mathtt{n})] \end{array}$

From $\mathsf{S}_\mathsf{C}(\tau)$ and $\mathsf{S}_\mathsf{C}(\tau')$ with Definition 12 we get:

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{S}_\mathsf{C}(\tau) \tag{3}$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau' \rightarrow \mathsf{S}_\mathsf{C}(\tau') \tag{4}$$

From $\mathsf{S_C}(\tau)$ and $\mathsf{S_C}(\tau')$ we get that the last rule applied to (3) and (4) was $(objtype)$. For each member, m'_i, of $\mathsf{S_C}(\tau)$ we get (for $i \in 1...n$):

$$\tau' \ \leq \ [\mathsf{m}_i : (\tau_i, \psi''_i)] \ \in \ \mathsf{C} \tag{5}$$

From $\tau \ \leq \ \tau'$, (1), (5) and closure rule $(closeTrans)$ we get (for $i \in 1...n$):

$$\tau \ \leq \ [\mathsf{m}_i : (\tau_i, \psi''_i)] \ \in \ \mathsf{C} \tag{6}$$

From (5) and (6) we get that for some $q \geq 0$ $p = n + q$, and that $\mathsf{m}_i = \mathsf{m}'_i$ for $i \in 1...n$. Therefore, with (3), (4) and the definition of $(objtype)$ we get (for $i \in 1...n$):

$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{7}$$
$$\psi'_i = \mathcal{A}(\mathsf{C}, \tau', \mathsf{m}_i) \tag{8}$$

From (1), $\tau \ \leq \ \tau'$, (7), (8) and Proposition 5.2 we get (for $i \in 1...n$):

$$\psi_i \leq \psi'_i \tag{9}$$

From Definition 12 with $\mathsf{S_C}$ and (5) we get (for $i \in 1...n$):

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau_i \rightarrow \mathsf{S_C}(\tau_i) \tag{10}$$

From (1), (5), (6), (3), (4) and (10) with Conjecture 1 we get (for $i \in 1...n$):

$$\mathsf{S_C}(\tau)(\mathsf{m}_i) \leq (\mathsf{S_C}(\tau_i), \psi''_i) \tag{11}$$
$$\mathsf{S_C}(\tau')(\mathsf{m}_i) \leq (\mathsf{S_C}(\tau_i), \psi''_i) \tag{12}$$

From (11), (12), (9) and the subtype rule $(\leq obj)$ we get:

$$\mathsf{S_C}(\tau) \leq \mathsf{S_C}(\tau') \tag{13}$$

From (13) and satisfaction rule $(solSub)$, from Figure 5.3, we get:

$$\mathsf{S_C} \vdash \tau \ \leq \ \tau' \tag{14}$$

Case: $\mathsf{S_C}(\tau) = \mu\, \alpha.(\mathsf{O}_1 \times \mathsf{tp}_2 \rightarrow \mathsf{tp}_3)$
$\mathsf{S_C}(\tau') = \mu\, \alpha'.(\mathsf{O}'_1 \times \mathsf{tp}'_2 \rightarrow \mathsf{tp}'_3)$

From $\mathsf{S_C}(\tau)$, $\mathsf{S_C}(\tau')$ and Definition 12 we get:

$$\mathsf{C}, \mathsf{V}_{\emptyset}, \tau \rightarrow \mathsf{tp} \tag{15}$$
$$\mathsf{C}, \mathsf{V}_{\emptyset}, \tau' \rightarrow \mathsf{tp}' \tag{16}$$

From $\mathsf{S}_{\mathsf{C}}(\tau)$ and $\mathsf{S}_{\mathsf{C}}(\tau')$ we get that the last rule applied to (15) and (16) was $(functype)$. Therefore, with translation rule $(functype)$ we get for some $\tau_1, \tau_1', \tau_2, \tau_2', \tau_3$ and τ_3':

$$\tau \ \leq \ (\tau_1 \times \tau_2 \rightarrow \tau_3) \in \mathsf{C} \tag{17}$$
$$\tau' \ \leq \ (\tau_1' \times \tau_2' \rightarrow \tau_3') \in \mathsf{C} \tag{18}$$

From (1), $\tau \ \leq \ \tau'$, (18) and the closure rule $(closeTrans)$ we get:

$$\tau \ \leq \ (\tau_1' \times \tau_2' \rightarrow \tau_3') \in \mathsf{C} \tag{19}$$

From (1), $\tau \ \leq \ \tau'$, (17), (19) and the closure rule $(closeCongFunc)$ we get:

$$\tau_1' \ \leq \ \tau_1, \ \tau_1 \ \leq \ \tau_1' \in \mathsf{C} \tag{20}$$
$$\tau_2' \ \leq \ \tau_2, \ \tau_2 \ \leq \ \tau_2' \in \mathsf{C} \tag{21}$$
$$\tau_3' \ \leq \ \tau_3, \ \tau_3 \ \leq \ \tau_3' \in \mathsf{C} \tag{22}$$

From $\mathsf{S}_{\mathsf{C}}(\tau)$, $\mathsf{S}_{\mathsf{C}}(\tau')$, (15), (16) and the definition of $(functype)$ we get:

$$\mathsf{V} = \mathsf{V}_{\emptyset} \bullet_{\mathsf{C}} (\tau, \alpha) \tag{23}$$
$$\mathsf{V}' = \mathsf{V}_{\emptyset} \bullet_{\mathsf{C}} (\tau', \alpha') \tag{24}$$

Let:

$$\beta_0 = \alpha \mapsto \alpha' \tag{25}$$

With (23), (24), (25) and Definition 7 we get:

$$\mathsf{C} \vdash \mathsf{V} \simeq_{\beta_0} \mathsf{V}' \tag{26}$$

From (15), (16), (17), (18), (23), (24) and the definition of $(functype)$ we get (for $i \in 1...3$):

$$\mathsf{C}, \mathsf{V}, \tau_i \rightarrow \mathsf{tp}_i \tag{27}$$
$$\mathsf{C}, \mathsf{V}', \tau_i' \rightarrow \mathsf{tp}_i' \tag{28}$$

From (27), (28), $\mathsf{S_C}(\tau)$ and $\mathsf{S_C}(\tau')$ we get:

$$\mathsf{tp}_1 = \mathsf{O}_1 \tag{29}$$
$$\mathsf{tp}'_1 = \mathsf{O}'_1 \tag{30}$$

From (1), (20), (21), (22), (27), (28), (26), (29) and (30) with Lemma 5 we get:

$$\mathsf{O}_1 \equiv_{\beta_0} \mathsf{O}'_1 \tag{31}$$
$$\mathsf{tp}_2 \equiv_{\beta_0} \mathsf{tp}'_2 \tag{32}$$
$$\mathsf{tp}_3 \equiv_{\beta_0} \mathsf{tp}'_3 \tag{33}$$

From (25), (31), (32), (33), $\mathsf{S_C}(\tau)$, $\mathsf{S_C}(\tau')$ and Proposition 8 we get:

$$\mathsf{S_C}(\tau) \equiv \mathsf{S_C}(\tau') \tag{34}$$

From (34) and subtyping rule $(\leq\ cong)$ we get:

$$\mathsf{S_C}(\tau) \leq \mathsf{S_C}(\tau') \tag{35}$$

From (35) and satisfaction rule $(solSub)$, from Figure 5.3, we get:

$$\mathsf{S_C} \vdash \tau\ \leq\ \tau' \tag{36}$$

Case: $\mathsf{S_C}(\tau) = \mathsf{Int}, \mathsf{S_C}(\tau') = \mathsf{Int}$

Straightforward.

Case: $\mathsf{c} = \tau\ \lhd_{\mathsf{m}}\ \tau'$

From (1), the definition of $(wlfEnsureObj)$, and $\tau\ \lhd_{\mathsf{m}}\ \tau'$ we get for some τ'' and ψ''':

$$\tau'\ \leq\ [\mathsf{m} : (\tau'', \psi''')] \in \mathsf{C} \tag{37}$$

From (37), (1) and the closure rule $(closeTransMem)$ we get:

$$\tau\ \leq\ [\mathsf{m} : (\tau'', \psi''')] \in \mathsf{C} \tag{38}$$

From (1) and the closure rules $(closeTransMem)$ and $(closeBalanceMem)$ we get:

$$\forall\, \mathsf{m}' \neq \mathsf{m} :\ \tau' \ \leq\ [\mathsf{m}' : (\tau'', \psi'')] \ \in\ \mathsf{C} \iff \tau\ \leq\ [\mathsf{m}' : (\tau'', \psi'')]\ \in\ \mathsf{C} \tag{39}$$

From Definition 12 each type variable in S_C is translated starting with the empty type variable function, $\mathsf{V}_\emptyset$. Thus, with (1), (37) and (38) the last rule applied to produce $\mathsf{S}_\mathsf{C}(\tau)$ and $\mathsf{S}_\mathsf{C}(\tau')$ was $(objtype)$:

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \to \mathsf{S}_\mathsf{C}(\tau) \tag{40}$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau' \to \mathsf{S}_\mathsf{C}(\tau') \tag{41}$$

Furthermore, from (37), (38) and (39) we have that $\mathsf{S}_\mathsf{C}(\tau)$ and $\mathsf{S}_\mathsf{C}(\tau')$ have the same members; therefore, for some $\alpha, \alpha', \mathsf{tp}_1 ... \mathsf{tp}_n, \mathsf{tp}'_1 ... \mathsf{tp}'_n, \psi_1 ... \psi_n$ and $\psi'_1 ... \psi'_n$ we get:

$$\mathsf{S}_\mathsf{C}(\tau) = \mu\ \alpha.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1) ... \mathsf{m} : (\mathsf{tp}, \psi) ... \mathsf{m}_\mathtt{n} : (\mathsf{tp}_\mathtt{n}, \psi_\mathtt{n})] \tag{42}$$
$$\mathsf{S}_\mathsf{C}(\tau') = \mu\ \alpha'.[\mathsf{m}_1 : (\mathsf{tp}'_1, \psi'_1) ... \mathsf{m} : (\mathsf{tp}', \psi') ... \mathsf{m}_\mathtt{p} : (\mathsf{tp}'_\mathtt{n}, \psi'_\mathtt{n})] \tag{43}$$

From (42), (43) and the definition of $(objtype)$ we get (for $i \in 1...n$):

$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{44}$$
$$\psi'_i = \mathcal{A}(\mathsf{C}, \tau', \mathsf{m}_i) \tag{45}$$

From (39), (44) and (45) we get:

$$\psi_i = \psi'_i \ \text{ (for all } i \text{ such that } \mathsf{m}_i \neq \mathsf{m}) \tag{46}$$

From (1), (39), (37) and (38) for any member m_i such that $\tau' \ \leq\ [\mathsf{m}_i : (\tau_i, \psi''_i)] \ \in\ \mathsf{C}$ then we have that $\tau \ \leq\ [\mathsf{m}_i : (\tau_i, \psi''_i)] \ \in\ \mathsf{C}$. Therefore, from Definition 12 with S_C we get (for $i \in 1...n$):

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau_i \to \mathsf{S}_\mathsf{C}(\tau_i) \tag{47}$$

Thus, from (1), (40), (41) and (47) with Conjecture 1 we get (for $i \in 1...n$):

$$\mathsf{S}_\mathsf{C}(\tau)(\mathsf{m}_i) \leq (\mathsf{S}_\mathsf{C}(\tau_i), \psi''_i) \tag{48}$$
$$\mathsf{S}_\mathsf{C}(\tau')(\mathsf{m}_i) \leq (\mathsf{S}_\mathsf{C}(\tau_i), \psi''_i) \tag{49}$$

From (48), (49), (46) and the congruence rule $(\equiv mem)$, from Figure 4.5, we get:

$$\mathsf{S_C}(\tau)(\mathsf{m}_i) \equiv \mathsf{S_C}(\tau')(\mathsf{m}_i) \ \text{ (for all } i \text{ such that } \mathsf{m}_i \neq \mathsf{m}) \tag{50}$$

From (42) and (43) let:

$$\psi = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}) \tag{51}$$

$$\psi' = \mathcal{A}(\mathsf{C}, \tau', \mathsf{m}) \tag{52}$$

If $\psi' = \circ$ then from the definition of subtyping we have that $\psi \leq \psi'$. If $\psi' = \bullet$ then from (1), the definition of $(closeTransMem)$, and the definition of $\mathcal{A}$ we get that $\psi =\bullet$, thus:

$$\psi \leq \psi' \tag{53}$$

From (42), (43), (48), (40), (53) and the subtyping rule $(\leq mem)$, from Figure 4.6, we get:

$$\mathsf{S_C}(\tau)(\mathsf{m}) \leq \mathsf{S_C}(\tau')(\mathsf{m}) \tag{54}$$

From (50), (54) and the satisfaction rule $(solMemChange)$, from Figure 5.3, we get:

$$\mathsf{S_C} \vdash \tau \ \lhd_{\mathsf{m}} \tau' \tag{55}$$

Case: $\mathsf{c} = \tau \ \lhd_* \ \tau'$

Similar to previous case.

Case: $\mathsf{c} = \tau \ \leq \ [\mathsf{m} : (\tau', \psi)]$

Follows from Conjecture 1.

Case: $\mathsf{c} = \tau \ \leq \ (\tau_1 \times \tau_2 \to \tau_3)$

Follows from Conjecture 2.

Case: $\mathsf{c} = \tau^\circ$

From Definition 12 with $\mathsf{S_C}$ we get:

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \to \mathsf{S_C}(\tau) \tag{56}$$

In the cases where the last rule applied to (56) was (int), $(unknown)$ or $(functype)$ it follows trivially that $\mathsf{S_C}(\tau)$ satisfies $(solNoDefs)$. Note that because the translation of τ starts with $\mathsf{V}_\emptyset$, the last rule applied to (56) cannot be (var).

We now consider the case where the last rule applied to (56) was $(objtype)$. We assume that τ translates to an object type; therefore, for some $\alpha, \mathsf{m}_1, ..., \mathsf{m}_n, \mathsf{tp}_1, ..., \mathsf{tp}_n$ and $\psi_1, ..., \psi_n$ we get:

$$\mathsf{S_C}(\tau) = \mu\,\alpha.[\mathsf{m_1} : (\mathsf{tp_1}, \psi_1)...\mathsf{m_n} : (\mathsf{tp_n}, \psi_\mathtt{n})] \tag{57}$$

From (56), (57) and the definition of $(objtype)$ we get for $i \in 1...n$:

$$\psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{58}$$

From (58), (1), the definition of rule $(wlfNoDefs)$, and the definition of $\mathcal{A}$ we get (for $i \in 1...n$):

$$\circ = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \tag{59}$$

From (59) and (57) we get:

$$\mathsf{S_C}(\tau) = \mu\,\alpha.[\mathsf{m_1} : (\mathsf{tp_1}, \circ)...\mathsf{m_n} : (\mathsf{tp_n}, \circ)] \tag{60}$$

From (60) we get:

$$\{\mathsf{m} \mid \mathsf{S_C}(\tau)(\mathsf{m}) = (\mathsf{t}, \bullet)\} = \emptyset \tag{61}$$

With (61) and the satisfaction rule $(solNoDefs)$, from Figure 5.3, we get:

$$\mathsf{S_C} \vdash \tau^\circ \tag{62}$$

Case: $\tau^{\mathbf{obj}}$

From τ° and Definition 12 with $\mathsf{S_C}$we get:

$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \to \mathsf{S_C}(\tau) \tag{63}$$

From (1) with rule $(wlfMix1)$ we have two cases to consider. First, there are no constraints for τ (apart from $\tau^{\mathbf{obj}}$) in which case the last rule applied to

(63) was $(unknown)$. If there are constraints for τ by $(wlfMix1)$ they must be of the form $\tau \leq [\mathsf{m} : (\_ , \_)]$ in which case the last rule applied to (63) was $(objtype)$. In both cases $\mathsf{S_C}(\tau)$ is an object type which satisfies the satisfaction rule $(solObjtype)$. Therefore, we establish (2).

□

5.9 Putting It All Together

We can now state the property: if type inference succeeds for a given program then that program will not get stuck. Without loss of generality we assume that there is a `main` function that execution starts with.

Conjecture 4 *For program* P*, empty heap* $\mathsf{H}_\emptyset$*, well-formed stack* χ*, if:*

$$\texttt{function main(x)}\ \{\,\mathsf{e}\,\} \in \mathsf{P} \qquad (1)$$
$$\vdash \mathsf{P} : \mathsf{C} \qquad (2)$$
$$\vdash \mathsf{C}\diamond \qquad (3)$$
$$\mathsf{e}, \mathsf{H}_\emptyset, \chi \twoheadrightarrow \mathsf{w}, \mathsf{H}', \chi' \qquad (4)$$

then:

$$\mathsf{w} \neq \texttt{stuckErr} \qquad (5)$$

Proof: From (3) and Conjecture 3 we have that the generated solution $\mathsf{S_C}$ from C is satisfied, $\mathsf{S_C} \vdash \mathsf{C}$. With $\mathsf{S_C}$ we can create a typed version of P: $\mathbb{P} = \mathcal{T}(\mathsf{P}, \mathsf{S_C})$. With (2) we can apply Theorem 3 to get that $\mathbb{P}$ is well-formed, $\vdash \mathbb{P}\diamond$. From $\mathbb{P}$ we can extract an environment, Γ, for checking function `main`. Furthermore, from $\mathbb{P}$ and the definition of well-formed programs, from Figure 4.12, we have that e is well-typed, $\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \mid\mid \Gamma'$. From the empty heap, $\mathsf{H}_\emptyset$, we can use an empty store typing, $\mathcal{T}_\emptyset$, with the well-formed stack, χ, to get that $\mathbb{P}, \Gamma, \mathcal{T}_\emptyset \vdash \mathsf{H}_\emptyset, \chi\ \diamond$. Finally, we apply Theorem 1 which ensures that the execution of e will not get stuck, *i.e.* $\mathsf{w} \neq$ `stuckErr`. This establishes (5). □

5.10 Related Work

We now discuss some work on type inference systems that are related and/or have influenced the design of the system presented in this chapter.

Palsberg

In [56] Palsberg considers type inference for the Abadi Cardelli object calculus[3]. The calculus allows member update but not member addition. Type inference is done for four of the first order type systems for the Abadi Cardelli object calculus. The most complex of these type systems allows recursive types and subtyping. The type rules are expressed in a way that requires the application of subsumption for each subterm. This has important consequences for the type inference algorithm.

The Palsberg system of constraints is a subset of those used in this thesis and contains two kinds:

$$\begin{aligned} \tau &\leq \tau' \\ \tau &\leq [\mathsf{m} : \tau] \end{aligned}$$

The use of subsumption is encoded in the system by having two type variables for each program subexpression: one before subtyping and one after. For variables there is x before subtyping and $[\![\mathsf{x}]\!]$ after. Similarly, for member access there is $[\![\mathsf{e.m}]\!]$ before subtyping and $< \mathsf{e.m} >$ after.

Our type system has no subsumption rule. Instead, it explicitly requires subtyping where necessary, *e.g.* with method call the actual parameter being a subtype of the formal parameter. Hence, the subtype relation is always used *explicitly* between the types of expressions in the program. Therefore, we don't need to use two type variables to model the application of subsumption.

The constraints are represented through a graph. Palsberg uses the graph to prove that the system operates in log-space. The nodes of the graph are the type variables. There are two kinds of edge: edges representing the constraints between type variables, $\tau \leq \tau'$, and edges representing the members of an object type, $\tau \leq [\mathsf{m} : \tau]$.

Once generated, the graph is closed. The closure process is simpler than the one presented in this thesis. In our system, closure adds new constraints; in Palsberg's system it adds new edges to the graph. Edges are added for the reflexive and transitive closure. Edges are also added to ensure that common members between two type variables are the same; this corresponds to our $(closeCong)$ rule.

A criterion for well-formedness of graphs is given which corresponds to our well-formedness of constraints. This ensures that the subtypes required by the constraints are honoured. In our system we must check that the types are not mixed (rules $(noMix1)$, $(noMix2)$ and $(noMix3)$), the annotations are correct (rule $(wlfNoDefs)$), and where appropriate a type variable is given an object type (rule $(wlfEnsureObj)$).

The graph is then converted to a finite automaton which is used to annotate the program. This corresponds to our translation relation for generating a solution from a set of constraints.

In [59] a type system is defined for the Abadi Cardelli object calculus with concatenation that uses recursive types. The definition of the object types is like ours (without function types) and the subtyping adopted is width subtyping. A type inference algorithm is given that broadly works like the system above. As in [56], two type variables are used to represent the type before and after subtyping. The algorithm is NP complete.

Palsberg and Schwartzbach

In [57, 58] type inference is developed for a simple class based language. Because the language is class based, the types in [57, 58] are the set of classes defined in the program.

The overall idea is that each expression, `e`, is assigned a type variable denoted by $[\![$`e`$]\!]$. Initial values can be given to type variables representing literal values such as integers, characters and strings. Each different kind of expression generates a set of constraints between type variables. We can think of the set of constraints as a directed graph where nodes are type variables and edges are inclusions between the type variables. As each expression in the program is analysed new constraints are added which in turn adds new edges to the graph. When an edge is added between nodes, the types *flow* along the edge. Once all expressions have been analysed, the type of an expression can be found by looking at the appropriate node in the graph.

The constraints for the simple cases, *e.g.* assignment and object instantiation, are represented by subset inclusion between type variables. For example, the term `car = new Vehicle(2003)` would produce constraints: `Vehicle` $\leq$ $[\![$`new Vehicle(2003)`$]\!]$ and $[\![$`2003`$]\!] \leq [\![$`x_Vehicle`$]\!]$. The term `x_Vehicle` denotes that `x` belongs to class `Vehicle`. The first constraint ensures that type `Vehicle` is included in the type for $[\![$`new Vehicle(2003)`$]\!]$. The second constraint ensures that the type of $[\![2003]\!]$ is included in the type for $[\![\texttt{x\_Vehicle}]\!]$. We see that the constraints model the flow of type information around the program.

One of the major challenges with type inference for object oriented languages is late binding. This is modelled by *conditional constraints* between type variables. For example, with method call we need to capture the possibility that the receiver *could* be of many different types. For each of the possible types of the receiver, we have a conditional constraint with the constraints for the method body associated with the type of the receiver. For example, method call `car.drive(10)` produces the following constraints (assuming a class `Vehicle` with method `drive` with body e):

$$\mathsf{Vehicle} \in [\![\texttt{car}]\!] \rightarrow [\![10]\!] \leq [\![\texttt{Vehicle\_drive\_x}]\!], [\![\mathsf{e}]\!] \leq [\![\texttt{car.drive}(10)]\!]$$

This ensures that if `Vehicle` is one of the types present in $[\![\texttt{car}]\!]$ (represented by $\mathsf{Vehicle} \in [\![\texttt{car}]\!]$) then $[\![10]\!]$ must be included in the type of the formal parameter of method `drive` from class `Vehicle`, and the type of the method body `drive` must be included in the type of the method call. In [57, 58], one conditional constraint is generated for each of the possible types for the receiver of a method call.

One major limitation of this system is that when dealing with method call only one set of constraints is generated for each method body. Therefore, a method or function invoked with a different receiver or argument types leads to a loss of precision. Consider a function `pair(x,y)` which returns its arguments in a tuple. Consider the following example where two calls are made with different argument types:

```
function pair(x,y) {
  return (x,y);
}

pair(new Vehicle(100), new Vehicle(200));
pair(3,4);
```

The algorithm in [57, 58] would give the following typing for `pair`:

```
function
pair(x:(Vehicle,Int),y:(Vehicle,Int)):((Vehicle,Int),(Vehicle,Int)) {
  return (x,y);
}
```

As more calls with different argument types are made the more conflated the typing for `pair` becomes. We do not require conditional constraints, because our types are structural. When many different kinds of object are passed to a function, our system infers a type for the parameter that includes only those members required by that function. This type will be a supertype of all the types of all the objects passed to that function.

Agesen, Palsberg and Schwartzbach

Using the analysis techniques in [57, 58] on the object based language SELF[6] results in virtually useless typings [4]. This is because in SELF many operations, *e.g.* conditionals, are implemented in terms of more basic constructs.

In [7] the work in [57, 58] is extended to work in a satisfactory manner for SELF[6]. Although SELF is object based, object structure can be derived from the program text. Method call in SELF is based on a lookup algorithm. If an object does not have a method the delegates are searched.

Each occurrence of an object is given a unique token ω. The delegation between objects is represented as constraints between the tokens. The type inference algorithm represents method lookup statically through analysis of the relations between objects. The major differences between our work and this work is that we have to determine the structure of objects and we have no delegation.

Eifrig, Smith and Trifonov

In [31] Eifrig et al. consider type inference for the class-based language *I-LOOP*. The types are *recursively constrained* in that a type is supplemented with a set of constraints $\tau\backslash\mathsf{C}$. They take a different approach to us, by defining type rules that generate constraints and then modifying the rules to make a deterministic and complete inference system. Fields and methods of a type are detailed with constraints of the form $\tau \leq \mathbf{Inst}\ \mathsf{m} : \tau'$ which states that τ has a field m of type τ'. This is similar to our constraint for detailing object type members, $\tau \leq [\mathsf{m}{:}(\tau',\psi)]$, with the exception we have the annotation ψ.

Wang and Smith

In [71] Wang et al. give a type inference system for Java that can statically verify the correctness of downcasts. The types used are based on those in [31]. There are types that describe the structure of objects: **obj** $(\delta, [\overline{\mathtt{l}_i : \tau_i}])$ where δ and $\mathtt{l}_i$ are labels for the class name and field/method names respectively. The structure of object types is derived from the class structure. They do not allocate new type variables for each function and method call site. Instead, the allocation of type variables is done when closing the constraints. By parameterizing closure with a mapping, it is possible to *share* type variables between different invocations of a method.

Other Work

In [55] Palsberg increases the precision of type inference, with respect to method call, by *expanding* by one level the code in order to differentiate call sites. In [60] code is expanded many times, at great computational cost. Another technique, adopted in [61], is to iterate the type inference to control the sharing of type variables; however, there is a reasonable computational overhead and difficulties with sharing type information between iterations with this technique.

In [4, 5] Agesen makes the observation that there is no such thing as a polymorphic call only polymorphic call sites. This gives rise the Cartesian Product Algorithm (CPA) which is a happy medium between expanding the code and iterating the type inference.

5.11 Discussion

We have developed a sound type inference system for JS_0. The main difficulty was handling the changing type of `this` and `x`. Our breakthrough was to use labelled type variables to represent the evolving type. At each program point, each type variable serves as a snapshot of the type of `this` and `x`. This elegantly handles situations where `this` and `x` are aliased, *i.e.* through assignment or as being passed to a function.

Our closure rules represent the flow of type information around the program. They are insensitive to the validity of the constraints; they do not validate any of the satisfaction requirements of the constraints. This is done later with a well-formedness criterion on closed constraints. Well-formedness is expressed in terms of the constraints themselves. A well-formed set of constraints can be satisfied by a solution. We developed a translation from well-formed constraints to types that can be used to generate a solution.

In the next chapter we will describe an implementation of the type inference algorithm presented in this chapter.

6 Implementation

IN THIS CHAPTER we will discuss a prototype implementation of the JS_0 type inference system described in Chapter 5. What we describe here is the current state of the implementation; we will continue to improve and expand it.

6.1 Objectives and Overview of the Implementation

The constraints generation system detailed in Chapter 5 produces a large number of constraints for even small programs. We found closing the constraints by hand error prone; we decided that a prototype implementation should be developed in order to study the system.

The brief was to provide an implementation that would allow the end user to submit a program for checking. We use the term "checking" in this context to mean apply type inference. If the checking is successful, the program should be displayed with the types that have been inferred. If the checking is unsuccessful, the errors should be highlighted.

Developing an implementation gave us an opportunity to extend JS_0 with some features that make programming easier. These include multiple parameters for functions, a string base type, local variable declarations, and statements. These extensions have their formal counterpart in the type system and inference systems. We study, but not prove, these extensions later in Section 6.2.

We developed the code in Python as it allowed us to rapidly prototype and refine the code. It also allowed us to provide a web front-end so that programs could be submitted remotely, and the results displayed in a web browser. Unfortunately, Python does suffer from performance problems and much time was spent optimizing the constraints in order to produce results in a timely fashion. We will discuss these optimizations in Section 6.4.

6.2 Implementation Extensions to JS_0

For the implementation, we extended JS_0 with the following features: multiple parameters for functions, a string base type, local variable declarations and statements. We give the extended syntax of JS_0 in Figure 6.1 where the new parts are highlighted. We now briefly discuss each of the extensions.

$\mathsf{P} \in \mathit{Program}$	::=	F^*	
$\mathsf{F} \in \mathit{FuncDecl}$	::=	`function` f ($i_0 ... i_n$) { s^* }	
$\mathsf{V} \in \mathit{VarDecl}$	::=	`var` i;	
$s \in \mathit{Stat}$	::=	`if` (e) { s } `else` { s }	conditional
		`return` e;	return
		e;	exp
$\mathsf{e} \in \mathit{Exp}$	::=	var	locals
		f	function identifier
		new f($\mathsf{e}_1 ... \mathsf{e}_n$)	object creation
		e; e	sequence
		e.m($\mathsf{e}_1 ... \mathsf{e}_n$)	member call
		e.m	member select
		f($\mathsf{e}_1 ... \mathsf{e}_n$)	global call
		lhs = e	assignment
		e_1 ? e_2 : e_3	conditional exp
		`null`	null
		`n`	integer
$\mathsf{var} \in \mathit{EnvVars}$	::=	`this` | i	
$\mathsf{lhs} \in \mathit{LeftHandSide}$	::=	i | e.m	

Identifiers

$\mathsf{f} \in \mathit{FuncID}$	::=	f | f′ | ...
$\mathsf{m} \in \mathit{MemberID}$	::=	m | m′ | ...
$i \in \mathit{Identifier}$	::=	i | i' | ...

Figure 6.1.: Implementation Syntax of JS_0

6.2.1 Multiple Parameter Functions

In real programming, it is unrealistic to require each function to have exactly one parameter. It is cumbersome to use parameter blocks to model multiple parameters. The implementation allows any number of parameters (including none). Furthermore, the parameters can have any name, unlike in JS$_0$ where the parameter is always x. Therefore, we introduce the identifier, i, which is used to represent the name of variables.

6.2.2 String Base Type

Strings are used in JavaScript code because all the text elements in a webpage are represented as a string (within objects representing the elements). Adding a new base type is relatively straightforward and poses no conceptual difficulties. We must check that strings are not mixed with any of the other types when checking the well-formedness of constraints. One can imagine a new well-formedness rule along the lines of the rules $(wlfMix1)$, $(wlfMix2)$ and $(wlfMix3)$ of Figure 5.9. As in Java, strings are treated as a base type. Note that JavaScript allows access to properties of a string literal, for example, `''Hello''.length` would return 5; we don't allow this.

6.2.3 Local Variable Declarations

As show in Figure 6.1, we allow any number of local variable declarations at the beginning of a function declaration . The name of a local variable is an identifier. The scope of local variable names is shared with the function parameter names. An attempt to define two variables with the same name or define a variable with a name already given as a formal parameter will result in an error.

In the type system we allow local variables to enjoy the same privileges as `this` and x. Therefore, the type of local variables can include potential members that are made definite upon assignment. Thus, the type inference system in the implementation uses labelled versions of all the variables in a function. In terms of the formalism one can think of the *pre-environment* of Chapter 5 as mapping all the variables in a function to a label. The implementation gives a label to variables that also includes the function they are defined in; we found this aids in presenting the results of the type inference.

6.2.4 Statements

Because JavaScript is statement oriented, we extend the syntax to include statements. To return a value from a function one must explicitly use the `return` keyword. Consider the `Date` function from Figure 3.2:

```
function Date(x) {
 this.mSec = x;
 this.add = addFn;
 this
}
```

The last expression is `this` (which will return the newly constructed object). For the implementation the above function would be written as:

```
function Date(x) {
 this.mSec = x;
 this.add = addFn;
 return this
}
```

If a function does not have a return statement then it is considered not to return anything. Hence, as in Java, we give it a `void` type. The type `void` can be considered a base type. Recall the rule $(Prog)$, from Figure 5.6, which generates constraints for each function, where the return value of a function f is represented by a type variable $[\![\texttt{ret\_f}]\!]$. In the implementation each occurrence of **return** e creates a constraint $[\![\mathtt{e}]\!] \leq [\![\texttt{ret\_f}]\!]$. Unlike rule $(Prog)$ the implementation does not create a constraint between the function body and the type variable $[\![\texttt{ret\_f}]\!]$.

We also add conditionals, **if** (e) **then** $\{ s \}$ **else** $\{s \}$, as statements to JS$_0$. Unlike conditional as an expression, $\mathtt{e}_1 ?\, \mathtt{e}_2 : \mathtt{e}_3$, the overall statement has the `void` type.

6.3 System Overview

The system can be broken down into the following parts:

1. Lexer/Parser
2. Constraints Generator
3. Constraint Closer
4. Type Generator

We now discuss each of the components in more detail.

6.3.1 Lexer/Parser

We used a parser generator called PLY by David M. Beazley[1]. PLY makes heavy use of Python's reflection capabilities. Each token for the lexer is specified via a variable or function prefixed by `t_`, *e.g.* for the `this` keyword `t_THIS = 'this'`. For more complex tokens, such as numbers or variable names, a function can be specified to check the validity of the token. For example, with variable names we check that the name is not a reserved keyword:

```
def t_NAME(self,t):
   r'[a-zA-Z_][a-zA-Z_0-9]*'
   t.type = self.reserved.get(t.value,'NAME')
   # Check for reserved words
   return t
```

Similarly, for the parser a function is specified for each of the productions in the grammar. Each function specifies the format of the production as a comment in the function definition. Each of the elements in the production can be referred to in the body of the function; this makes it possible to build an Abstract Syntax Tree (AST). For example, the parser function for member access, e.m, is defined as:

```
def p_expression_memacc(self,t):
   'expression : expression DOT NAME'
   t[0] = MemAcc(t[1],t[3])
```

We see that each part of the production in the comment `'expression : expression DOT NAME'` is referenced through the array `t`. For each production we also specify possible erroneous versions to give better error handling.

6.3.2 Constraints Generator

From the AST the constraint generator creates constraints using the constraint generation rules of Section 5.3. The class `ConGen` implements a visitor[36] that walks the AST generating constraints. Constraints are represented through the class `Cons` and its subclasses: `ConsFlow` for constraints of the form $\tau \leq \sigma$ and `ConsMem` for constraints of the from $\tau \lhd \tau'$. There is no class representing constraints of the form $\tau^{\Box}$; we shall discuss why when we describe the type generator in Section 6.3.4.

The type variables used in the constraints are represented through the class `TypeVar` and its subclasses. Each instance of `TypeVar` maintains a reference to

[1]`http://www.dabeaz.com/ply/`

the expression it represents along with its labelling. The management of the labels (handled by pre-environments γ in the formalism) is through an *associative array* mapping variables to their current label. With assignments to `this` and `x`, and conditionals, see the constraint generation rules of Figures 5.4 and 5.5, we introduce new labels. Labels are represented as integers and are incremented when a fresh one is required.

Constraint generation will fail if:

- a global function call (or new) is used with an incorrect number of parameters, or
- an undefined variable or function is used.

In any of these cases, the exception `ConstraintError` is thrown.

6.3.3 Constraint Closure

Each constraint class, `ConsFlow` and `ConsMem`, has a `close` method that takes another constraint as a parameter. Intuitively, this represents the pair of constraint on the left hand side of each of the closure rules (with the exception of rule ($closeBalanceBranch$)). The result of invoking `close` will be the constraints prescribed by the relevant closure rule. Thus, constraint closure is implemented as invocations of the `close` method between pairs of constraints:

```
c1.close(c2)
```

The overall close procedure is captured by two nested loops that invoke `close` between each of the constraints.

Given that functions can have more than one parameter, unlike the closure rules of Section 5.5, it is possible for `close` to fail. When `close` is applied between two type variables that represent functions, the arity of the functions *must* be the same. We don't model JavaScript's variable arity parameter mechanism (where unspecified parameters are given a default value).

6.3.4 Type Generator

Types are represented by the class `Type` and its subclasses: `TypeObj` for object types, `TypeFunc` for functions, `TypeInt` for integers and `TypeString` for strings. A further class, `TypeMember`, represents the members of an object type. Object and function types have a field `recur` which indicates if the type is recursive; when displaying types this is used to determine if the μ binding should be displayed or not.

The class `TypeMaker` is responsible for generating types from a set of closed constraints. During initialisation an attempt is made to determine what kind of type, by looking at the associate constraints, a type variable represents. For example, if $\tau \leq \mathsf{Int} \in \mathsf{C}$ then τ would be marked as an integer. Type variables that are equivalent, $\tau \leq \tau', \tau' \leq \tau \in \mathsf{C}$, are also recorded.

Once the kind of a type has been determined a check is done to determine if there is any mixing *c.f.* $(wlfMix1)$, $(wlfMix2)$ and $(wlfMix3)$ of Figure 5.9. For example, no type variable is expected to be an integer type *and* object type. Any errors are recorded in the hash table `error` which maps erroneous type variables to the appropriate error text (*e.g.* "`this_f_0 Being used as (Object and Integer)`"). Any type variable that must be of an object type, through constraint $\tau^{\mathbf{obj}}$, are also checked.

Finally, we ensure that global and constructor functions have a receiver with no definite members *c.f.* $(wlfNoDefs)$ of Figure 5.9. Again, any errors are recorded in `error`, *e.g.* error text "`Function f has definite member m`".

6.4 Optimizations

We did two kinds of optimization while developing the implementation. The first kind focused on specific Python optimizations. The second kind focused on the inference system itself.

6.4.1 Improving Python Performance

In the implementation, there are many situations where two objects are compared for equality. For example, when closing constraints we need to know when to stop, *i.e.* the closure rules are not adding any *new* constraints. Thus, the code to compare two constraints (and the type variables that comprise them) needed to be efficient. The default implementation of object equality in Python compares the hash values of objects (the hash value of an object is the unique value given by the Python runtime for each object). This was inadequate because two objects may represent the same constraint but have different hash values. Python allows a class to override how its objects are compared through the method `__eq__`. The implementation of this method for constraints is straightforward, for example, `__eq__` would return true if the left and right hand sides of two flow constraints are the *same*. This involves comparing objects representing type variables and in turn objects representing expressions.

The code to compare two objects was initially a performance bottleneck in the system. We drastically improved performance by associating a *type token* with each object representing constraints, type variables and expressions. The first test in the implementation of `__eq__` is to check that the type tokens are the same. If they are not then the two objects are not representing the same thing. Furthermore, the tokens are represented by integers which can be compared very quickly.

6.4.2 Improving Performance of the Inference System

We now discuss how we reduced the number of constraints generated.

Equality Constraints

The closure rules $(closeCong)$ and $(closeCongFunc)$, from Figure 5.8, generate constraints of the form: $\tau' \leq \tau''$, $\tau'' \leq \tau'$. We decided to replace these pairs of constraints with one new kind of "equality" constraint:

$$\tau = \tau'$$

We modified the code implementing the closure rules $(closeCong)$ and $(closeCongFunc)$; instead of a pair of constraints we generate a single equality constraint (see the closure rules $(closeCongEq)$ and $(closeCongFuncEq)$ shown in Figure 6.2). The implementation ignores flow and equality constraints between the same type variable, *e.g.* we ignore the constraint $\tau = \tau$.

Reducing Member Constraints

Recall that members of an object type are detailed with constraints of the form $\tau \leq [\mathsf{m} : (\tau', \psi)]$. It may be that several constraints for the same member are generated. We reduced this to a minimum by giving each object representing a type variable a list of the members it has. The code for closure rules $(closeTrans)$, $(closeBalanceMem)$, $(closeBalanceMems)$ and $(closeBalanceBranch)$ was modified to check whether a type variable already has a member constraint before

$$\frac{}{\tau \;\leq\; [\mathsf{m} : (\tau', \_)],\; \tau \;\leq\; [\mathsf{m} : (\tau'', \_)] \;\longrightarrow\; \tau' \;= \tau''}\;(closeCongEq)$$

$$\frac{}{\left.\begin{array}{l}\tau \;\leq\; (\tau_1 \times \tau_2 \to \tau_3),\\ \tau \;\leq\; (\tau_1' \times \tau_2' \to \tau_3')\end{array}\right\} \;\longrightarrow\; \begin{array}{l}\tau_1' \;= \tau_1,\\ \tau_2' \;= \tau_2,\\ \tau_3 \;= \tau_3',\end{array}}\;(closeCongFuncEq)$$

Figure 6.2.: Closure rules with equality constraints

adding it. There are two outstanding issues with this optimization: congruence of members *not* added and the annotations of members.

An attempt to add the same member twice to a type variable will result in the annotation being updated, if the new member has an annotation that is subtype of the current one. For example, consider closing these constraints:

$$\tau \;\leq\; \tau',\; \tau' \;\leq\; [\mathsf{m} : (\tau'', \bullet)]$$

If τ already has a member m with annotation $\bullet$ then we do not update the member or add the constraint $\tau \;\leq\; [\mathsf{m} : (\tau'', \bullet)]$. However, if τ has a member m with annotation $\circ$, then we must add the constraint $\tau \;\leq\; [\mathsf{m} : (\tau'', \bullet)]$ to ensure τ gets the correct annotation and update the annotation of the member.

To ensure that this optimization is sound, when a member is *not* added (because it is already there) we still have to equate the type variables of those members. Intuitively, we are doing what closure $(closeCong)$ would do had the constraint for that member been added. Returning to the example above, if we do not add constraint $\tau \;\leq\; [\mathsf{m} : (\tau'', \bullet)]$, then we add constraint $\tau''' \;= \tau''$.

6.5 Examples

We have applied our implementation to the two examples in Chapter 3 (Figures 3.2 and 3.5). We added a new module that outputs the constraints and types as Latex code. The constraints (unclosed) for the examples are presented in Appendices A.1 and A.2. We now present the inferred types. In order to make the presentation readable we format object types into columns. For example:

$$\left[\ \texttt{submit} : (\left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet)\end{array}\right], \bullet)\ \right]$$

We also format function types into columns where all the parameters (including `this`) are grouped together, { }, followed by the return type. For example:

$$\left\{\begin{array}{l}\left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\ \mathsf{Int}\end{array}\right\} \rightarrow \left[\ \texttt{submit} : (\left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet)\end{array}\right], \bullet)\ \right]$$

6.5.1 Date Example

Running the implementation on the code in Figure 3.2 infers annotations for functions `Date`, `addFn` and `main` as follows (the types of the type variables are shown in Figure 6.3):

$$\texttt{function Date}(\texttt{x}) : ([\![\texttt{this\_Date}]\!] \times [\![\texttt{ret\_Date}]\!] \rightarrow [\![\texttt{ret\_Date}]\!])\ \{...$$
$$\texttt{function addFn}(\texttt{x}) : ([\![\texttt{this\_addFn}]\!] \times [\![\texttt{x\_addFn}]\!] \rightarrow [\![\texttt{ret\_addFn}]\!])\ \{...$$
$$\texttt{function main}(\texttt{x}, \texttt{y}) : ([\![\texttt{this\_main}]\!] \times [\![\texttt{x\_main}]\!] \times [\![\texttt{y\_main}]\!] \rightarrow [\![\texttt{ret\_main}]\!])\ \{...$$

6.5.2 Web Example

Running the implementation on the code in Figure 3.5 infers annotations for functions `checkform`, `input`, `onSubmit`, `form` and `main` as follows (the types of the type variables are shown in Figures 6.4 and 6.5):

$$\texttt{function checkform}(\texttt{theform}) :$$
$$([\![\texttt{this\_checkform}]\!] \times [\![\texttt{theform\_checkform}]\!] \rightarrow [\![\texttt{ret\_checkform}]\!])\ \{...$$
$$\texttt{function input}(\texttt{value}) : ([\![\texttt{this\_input}]\!] \times [\![\texttt{value\_input}]\!] \rightarrow [\![\texttt{ret\_input}]\!])\ \{...$$
$$\texttt{function onSubmit}() : [\![\texttt{this\_onSubmit}]\!] \rightarrow [\![\texttt{ret\_onSubmit}]\!]\ \{...$$
$$\texttt{function form}() : [\![\texttt{this\_onSubmit}]\!] \rightarrow [\![\texttt{ret\_onSubmit}]\!]\ \{...$$
$$\texttt{function main}(\texttt{htmlform\_main}, \texttt{htmlinput\_main}) :$$
$$([\![\texttt{this\_main}]\!] \times [\![\texttt{htmlinput\_main}]\!] \times [\![\texttt{htmlform\_main}]\!] \rightarrow [\![\texttt{ret\_main}]\!])\ \{...$$

$$
\begin{array}{rcl}
\llbracket \texttt{this\_Date} \rrbracket & : & \left[\begin{array}{l} \texttt{add} : (\left\{ \begin{array}{c} \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\ \\ \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \end{array} \right\} \to [], \circ) \\ \texttt{mSec} : (\mathsf{Int}, \circ) \end{array} \right] \\
\llbracket \texttt{x\_Date} \rrbracket & : & \mathsf{Int} \\
\llbracket \texttt{ret\_Date} \rrbracket & : & \left[\begin{array}{l} \texttt{add} : (\left\{ \begin{array}{c} \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\ \\ \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \end{array} \right\} \to [], \bullet) \\ \texttt{mSec} : (\mathsf{Int}, \bullet) \end{array} \right] \\
\\
\llbracket \texttt{this\_addFn} \rrbracket & : & \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\
\llbracket \texttt{x\_addFn} \rrbracket & : & \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\
\llbracket \texttt{ret\_addFn} \rrbracket & : & [] \\
\\
\llbracket \texttt{this\_main} \rrbracket & : & [] \\
\llbracket \texttt{x\_main} \rrbracket & : & \left[\begin{array}{l} \texttt{add} : (\left\{ \begin{array}{c} \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\ \\ \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \end{array} \right\} \to [], \bullet) \\ \texttt{mSec} : (\mathsf{Int}, \bullet) \end{array} \right] \\
\llbracket \texttt{y\_main} \rrbracket & : & \left[\ \texttt{mSec} : (\mathsf{Int}, \bullet)\ \right] \\
\llbracket \texttt{ret\_main} \rrbracket & : & []
\end{array}
$$

Figure 6.3.: Inferred Types for Date Example

$$
\begin{array}{rcl}
[\![\texttt{this\_checkform}]\!] & : & [\,] \\[1em]
[\![\texttt{theform\_checkform}]\!] & : & \left[\ \texttt{submit} : (\left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet)\end{array}\right], \bullet)\ \right] \\[1em]
[\![\texttt{ret\_checkform}]\!] & : & \mathsf{String} \\[2em]
[\![\texttt{this\_input}]\!] & : & \left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \circ) \\ \texttt{value} : (\mathsf{String}, \circ)\end{array}\right] \\[1em]
[\![\texttt{value\_input}]\!] & : & \mathsf{String} \\[1em]
[\![\texttt{ret\_input}]\!] & : & \left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet)\end{array}\right] \\[2em]
[\![\texttt{this\_onSubmit}]\!] & : & \left[\ \texttt{submit} : (\left[\begin{array}{l}\texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet)\end{array}\right], \bullet)\ \right] \\[1em]
[\![\texttt{ret\_onSubmit}]\!] & : & \mathsf{String}
\end{array}
$$

Figure 6.4.: Inferred Types for Web Example

$$
\begin{array}{rcl}
[\![\texttt{this\_form}]\!] & : & \left[\begin{array}{l}
\texttt{onSubmit} : \left(\left\{ \left[\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \bullet\right) \right] \right\} \rightarrow \mathsf{String}, \circ\right) \\
\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \circ\right)
\end{array}\right] \\
[\![\texttt{ret\_form}]\!] & : & \left[\begin{array}{l}
\texttt{onSubmit} : \left(\left\{ \left[\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \bullet\right) \right] \right\} \rightarrow \mathsf{String}, \bullet\right) \\
\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \circ\right)
\end{array}\right] \\
[\![\texttt{this\_main}]\!] & : & [] \\
[\![\texttt{htmlinput\_main}]\!] & : & \left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right] \\
[\![\texttt{htmlform\_main}]\!] & : & \left[\begin{array}{l}
\texttt{onSubmit} : \left(\left\{ \left[\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \bullet\right) \right] \right\} \rightarrow \mathsf{String}, \bullet\right) \\
\texttt{submit} : \left(\left[\begin{array}{l} \texttt{disabled} : (\mathsf{Int}, \bullet) \\ \texttt{value} : (\mathsf{String}, \bullet) \end{array}\right], \circ\right)
\end{array}\right] \\
[\![\texttt{ret\_main}]\!] & : & \mathsf{String}
\end{array}
$$

Figure 6.5.: Inferred Types for Web Example Cont.

6.6 Discussion

We have developed a prototype implementation of the type inference algorithm presented in the previous chapter. We extended the language with more practical features. We developed optimizations to reduce the number of constraints generated. This involved finding new ways to represent the members constraints of type variables and the equivalence between type variables. These optimizations greatly improved the performance of the implementation.

7 Conclusions and Future Work

THE GOAL OF this thesis was to develop type inference system for JavaScript. Why did we decide to do this? A good starting point is to look at JavaScript and how it is used in practice. There is no doubt that JavaScript has become incredibly popular; its support for rapid development of software make it a good choice for many developers. Its inclusion in web browsers has been a mixed blessing. While this has certainly contributed to its popularity, bad code and browser differences have had it vilified. Many errors are easily avoidable, namely syntax errors and variable misspellings; the tool jslint[1] can find them. But there is a kind of error that remains a problem: attempting to access undefined fields or methods of objects. Our type inference system can detect this kind of error. Therefore, we see our work as a first step towards making JavaScript code more robust.

We started by developing a formalised subset of JavaScript: JS_0. This included an operational semantics which describes execution of JS_0 programs. The features and behaviour we gave to JS_0 are a tradeoff between real JavaScript and the aim of doing type inference.

We developed a type system for JS_0 that allowed us to give a type annotated version of JS_0: $\mathrm{JS}_0^{\mathsf{T}}$. It was clear that nominal types were limited for modelling many of the characteristics of JavaScript programs; this was demonstrated by our earlier experience in this area [11, 12]. We did not want the programmer to have to refactor code in order to get it to type check. We overcome the limitations of this earlier work by using structural types with structural subtyping. We were able to track statically the evolution of objects by classifying members into potential and definite. We proved the soundness of our type system and thus achieved the safety guarantees we required.

We developed a type inference system that removes from the programmer the burden of *actually* writing types in the code. Our system takes a JS_0 program and generates a set of constraints. By closing the constraints, we could determine

[1] `www.jslint.com`

whether they were satisfiable. We proved the soundness of the constraint system with respect to the type system; this formally shows the correspondence between satisfied constraints and our type system. We showed how well-formed constraints can be converted into a solution that satisfies those constraints.

7.1 Limitations

JavaScript was not designed with static typing in mind (as shown in Section 3). This had clear consequences for our work. We could have gone in two directions with the type system. First, develop a type system that gives some safety guarantees but does not restrict the language. Second, develop a sound type system but sacrifice some of the language features.

The first approach was used in JavaScript 2 (discussed in Section 2.7). This work has had limited success: no current web browser supports it. One of the major criticisms was that by adding classes they were mixing Java style programming with JavaScript. Finally, JavaScript 2 has limitations in terms of the safety guarantees it offers.

We took the second approach in this thesis. Our aim was to provide a sound type system for JavaScript. The guarantees this gives are familiar to Java developers, but probably not to JavaScript developers. This led us to define a subset of JavaScript that allowed us to develop a sound type system. Naturally, sacrifices had to be made in the language in order to achieve this. Some might argue that the type system is too conservative and that some of the choices we made are unrealistic, *e.g.* making access to non-existent members a type error. We feel that the road to better programming starts with choosing which members an object will have: types help to enforce those choices. It is better to have a sound system, even with its restrictions, than a half attempt that gives no real guarantees.

There are also the obvious limitations involved with any formalisation of a system. To capture the essence of the problem we have to reduce some of the practical features. For example, only one function parameter, only one base type and no input/output functionality. The implementation attempts to compensate for some of these limitations.

To appreciate fully the limitations of a system one must evaluate how the system performs in the *real world*. The implementation was essential for this. We first took small examples and ran them through the implementation. Naturally, as they were engineered by us, we were clear as to how they should behave. Our next line of evaluation was to look at JavaScript code in web pages. We wanted to analyse pages that had errors in them to see if our system would find them. This posed some problems. First, we found that code in web pages makes use of the Document Object Model (DOM) which is akin to the standard library in C++. Any tests we did perform on code in web pages required the construction of code to represent the objects in the page. This was demonstrated in our second example of Chapter 3. Second, we had to find pages with errors that were neither syntactic or because of browser differences in representing the DOM. This was further compounded by most modern browsers simply ignoring errors in order to give a better browsing experience.

Those pages with errors that we did find were mostly attempts to dereference the undefined value (as a result of access to an undefined member of an object) as was shown in the second example of Chapter 3, or calling an undefined method on an object. Finally, we were unable to test pages that included language constructs we did not consider. The development of a full representation of the DOM will be helpful in further evaluating the system.

7.2 Future Work

7.2.1 Mixed Mode System

It would be nice if programmers could specify some of the types in their JS_0 programs. This is especially relevant to library code where the types can act as documentation. We now outline a possible solution using the work developed in this thesis.

To support a mixed mode system the type inference algorithm would have to be extended to take into account types already in the code. This could be achieved by expressing user given types as constraints. We now return to the Date example of Chapters 3 and 4 (the code is shown in Figure 7.1). Note that we have removed variable `y` and the annotation on variable `x`. Recall that we use $\mathtt{t}_1$ for type $[\mathtt{mSec} : (\mathsf{Int}, \circ), \ \mathtt{add} : ((\mathtt{t}_2 \times \mathtt{t}_2 \rightarrow \mathtt{t}_2), \circ)]$ and $\mathtt{t}_2$ for type

$\mu\ \alpha.[\texttt{mSec} : (\mathsf{Int}, \bullet),\ \texttt{add} : ((\alpha \times \alpha \to \alpha), \bullet)]$. We could represent type $\texttt{t}_2$ as the following constraints:

$$\begin{aligned}
[\![\texttt{t\_2}]\!] &\leq [\texttt{mSec} : ([\![\texttt{t\_2.mSec}]\!], \bullet)] \\
[\![\texttt{t\_2.mSec}]\!] &\leq \mathsf{Int} \\
[\![\texttt{t\_2}]\!] &\leq [\texttt{add} : ([\![\texttt{t\_2.add}]\!], \bullet)] \\
\texttt{t\_2.add} &\leq ([\![\texttt{t\_2}]\!] \times [\![\texttt{t\_2}]\!]) \to [\![\texttt{t\_2}]\!]
\end{aligned}$$

Note the use of extra type variables $[\![\texttt{t\_2}]\!]$, $[\![\texttt{t\_1.mSec}]\!]$ and $[\![\texttt{t\_1.add}]\!]$ representing the 'type' of type $\texttt{t}_2$ and members `add` and `mSec` respectively. Similarly, for type $\texttt{t}_1$ we could use the following constraints:

$$\begin{aligned}
[\![\texttt{t\_1}]\!] &\leq [\texttt{mSec} : ([\![\texttt{t\_1.mSec}]\!], \circ)] \\
[\![\texttt{t\_1.mSec}]\!] &\leq \mathsf{Int} \\
[\![\texttt{t\_1}]\!] &\leq [\texttt{add} : ([\![\texttt{t\_1.add}]\!], \circ)] \\
\texttt{t\_1.add} &\leq ([\![\texttt{t\_2}]\!] \times [\![\texttt{t\_2}]\!]) \to [\![\texttt{t\_2}]\!]
\end{aligned}$$

We have simplified here by using type variable $[\![\texttt{t\_2}]\!]$ to represent the *use* of $\texttt{t}_2$ in $\texttt{t}_1$. Had we not done this we would have constraints for *three* object types all representing $\texttt{t}_2$.

Once the constraints for the types have been created they can be used to create constraints to represent the annotations of a function (if it has any). For function `Date` we can express annotation $(\texttt{t}_1 \times \mathsf{Int} \to \texttt{t}_2)$ as constraints:

$$\begin{aligned}
[\![\texttt{this\_Date}]\!] &\leq [\![\texttt{t\_1}]\!] \\
[\![\texttt{x\_Date}]\!] &\leq \mathsf{Int} \\
[\![\texttt{ret\_Date}]\!] &\leq [\![\texttt{t\_2}]\!]
\end{aligned}$$

Note that we have simplified again by using type variable $[\![\texttt{t\_1}]\!]$ and $[\![\texttt{t\_2}]\!]$ for all occurrences of $\texttt{t}_1$ and $\texttt{t}_2$ respectively.

Once all the constraint for the types and annotations in the program have been created, the program is stripped of annotations and submitted for constraint generation. All the constraints are combined together and closed. Any inconsistencies between the annotated types and the program will be discovered. For example, if `Date` were used with a parameter that was not of type integer, this would clash with the constraint inferred by the annotation $[\![\texttt{t\_1.mSec}]\!] \leq \mathsf{Int}$.

```
function Date(x):$(t_1 \times \mathsf{Int} \to t_2)$ {
 this.mSec = x;
 this.add = addFn;
 this
}

function addFn(x):$(t_2 \times t_2 \to t_2)$ {
 this.mSec = this.mSec + x.mSec; this;
}
//Main
x = new Date(1000);
```

Figure 7.1.: Typed JS_0 Date Example.

7.2.2 Implementation

The implementation is still a prototype and we would like to include the following features: arrays, wrapped objects representing base types, coercions and functions as objects. We would also like to improve the performance of the implementation; Python can become very slow when there are a large number of objects active in the virtual machine. We could reimplement the code in C++ or Java.

Another interesting avenue would be to automate the translation of a web page into JS_0 code representing the DOM elements. In the second example in Chapter 3 we gave a taste of what the code might look like for a particular webpage; we wrote the code representing the `form` and `input` elements by hand.

7.2.3 Prototyping

JavaScript has a prototyping mechanism, as discussed in Chapter 3, which allows objects to delegate execution of a method to another object. Our work in [9, 26] for typing δ could provide the foundation for adding this to JS_0. We could allow each object to have a special prototype field; object types would include a prototype member which itself was an object type. It is still unclear exactly how the type inference would work with these prototype fields. Recall that with delegation the receiver of a method call remains the same (even when another object is executing the code on its behalf).

7.2.4 Formal Work

If we were able to prove that the type system has principal types, and that the inference algorithm finds those types, we would have a compositional analysis.

This would allow different pieces of code to be analysed in isolation and then used together.

We would like to prove Conjectures 1 and 2. We have attempted many formulations of the conjectures with a view of constructing an inductive proof. Each formulation would either permit proof of the base cases but fail on the inductive cases or visa versa. We believe the central issue is that the substitution that occurs upon member selection (recall Section 4.1.3) can produce a type that cannot be generated by the translation relation. We would like to look at other proof techniques, such as bisimulation, and reformulate the conjectures appropriately.

7.2.5 Applicability To Other Work

Our work could be applied to languages similar to JavaScript; one candidate is Python. In Python there are classes but they don't fully describe objects. Members can be added (or removed) after an object has been created. The types we have developed for JS_0 could be applied to Python; classes would be treated as constructor functions and methods could be represented as functions that are assigned to fields.

Our work could also form the basis of a development environment. A browser could indicate the types of variables in the code and detail the members of objects.

Bibliography

[1] Martín Abadi and Luca Cardelli. An imperative object calculus. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 471–485, London, UK, 1995. Springer-Verlag.

[2] Martín Abadi and Luca Cardelli. An imperative object calculus. *Theor. Pract. Object Syst.*, 1(3):151–166, 1995.

[3] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

[4] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP*, pages 2–26, 1995.

[5] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford, CA, USA, 1996.

[6] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Holzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The SELF 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc. and Stanford Unversity, 1995. `http://research.sun.com/self/`.

[7] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 247–267. Springer-Verlag, 1993.

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[9] Christopher Anderson, Ferruccio Barbanera, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. Can Addresses be Types? (a case study: objects with delegation). In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.

[10] Christopher Anderson and Sophia Drossopoulou. δ an imperative object based calculus. Presented at the workshop USE in 2002, Malaga, 2002.

[11] Christopher Anderson and Sophia Drossopoulou. BabyJ - From object based to class based programming via types. In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.

[12] Christopher Anderson and Sophia Drossopoulou. Type Inference for JavaScript. Internal Report `http://www.binarylord.com/work/babyjinfer.pdf`, 2003.

[13] Christopher Anderson and Paola Giannini. Type checking for Javascript. In *WOOD '04*, volume WOOD of *ENTCS*. Elsevier, 2004. `http://www.binarylord.com/work/js0wood.pdf`.

[14] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Type Inference for Scripting Languages. In *Proc. ECOOP'05*, volume 3586 of *Lecture Notes in Computer Science*, pages 441–469. Springer, 2005.

[15] Stig Sther Bakken. PHP 4 Manual. `http://www.php.net/manual/en/`.

[16] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., 1994.

[17] Viviana Bono, Ferruccio Damiani, and Paola Giannini. A calculus for "environment-aware" computation. In *F-WAN'02*, volume 66.3 of *ENTCS*. Elsevier, 2002.

[18] Viviana Bono and Kathleen Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP'98*, volume 1445 of *LNCS*, pages 462–497, 1998. A preliminary version already appeared in Proc. of 5th Annual FOOL Workshop.

[19] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[20] Kim Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, 952:27–51, 1995.

[21] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[22] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report, SRC-31*. Digital Systems Research Center, 1988.

[23] Luca Cardelli, Jim E. Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, 1989.

[24] Craig Chambers. The Cecil language: Specification and rationale. Technical Report TR-93-03-05, 1993.

[25] Andrew A. Chien and Willaim J. Dally. Concurrent aggregates (CA). In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 187–196, New York, NY, USA, 1990. ACM Press.

[26] Ferruccio Damiani and Paola Giannini. Alias types for "environment-aware" computations. In *WOOD'03*, volume 82.8 of *ENTCS*. Elsevier, 2003.

[27] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 389–418. Springer-Verlag, June 1997.

[28] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[29] ECMA International, ECMA-262, 3rd Edition. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.htm`, December 1999.

[30] ECMA International, ECMA-334, 3rd Edition. C♯ Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, June 2005.

[31] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Conference on Object-Oriented*, pages 169–184, 1995.

[32] Arnaud Le Hors et al. Document Object Model (DOM) Level 3 Core Specification. Technical report, October 1998. `http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107`.

[33] Kathleen Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996. Available as Stanford Computer Science Technical Report number STAN-CS-TR-98-1602.

[34] Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*. Springer LNCS, 1995.

[35] David Flanagan. *JavaScript - The Definitive Guide*. O'Reilly, 1998.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: abstraction and reuse of object-oriented design. 2002.

[37] Adele Goldberg and David Robson. *SmallTalk-80 The Language and its Implementation*. ADDWES, 1983.

[38] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[39] Paul Graham. *ANSI Common Lisp*. Prentice-Hall, Inc., 1995.

[40] Waldemar Horwat. Javascript 2.0: Evolving a language for evolving systems. `http://www.mozilla.org/js/language/evolvingJS.pdf`.

[41] Simon Peyton Jones and John Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://haskell.org/definition/haskell98-report.pdf`, February 1999.

[42] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Inc., 1988.

[43] Xavier Leroy. *The Objective Caml system*. Institut National de Recherche en Informatique et en Automatique, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[44] Luigi Liquori. Bounded Polymorphism for Extensible Objects. In *Proc. of Types, International Workshop on Types for Proof and Programs*, volume 1657 of *Lecture Notes in Computer Sciences*, pages 149–163. Springer Verlag, 1998. `http://www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/types-98.ps.gz`.

[45] Luigi Liquori. On Object Extension. In *Proc. of ECOOP, European Conference on Object Oriented Programming*, volume 1445 of *Lecture Notes in Computer Sciences*, pages 498–552. Springer Verlag, 1998. `http://www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/ecoop-98.ps.gz`.

[46] Mark Lutz. *Programming Python*. O'Reilly, 2001.

[47] Yukihiro Matsumoto. Ruby Reference Manual. `http://www.ruby-lang.org/en/`.

[48] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, 1993.

[49] Bertrand Meyer. Eiffel: a language and environment for software engineering. *J. Syst. Softw.*, 8(3):199–246, 1988.

[50] Microsoft. .NET Platfrom. `http://msdn.microsoft.com/`.

[51] Microsoft. JScript .NET Language Specification. `http://msdn.microsoft.com/`, 2003.

[52] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990.

[53] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[54] Netscape. ECMAScript Edition 4. `http://www.mozilla.org/js/language/es4/`.

[55] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.

[56] Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123(2):198–209, 1995.

[57] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY, 1991. ACM Press.

[58] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., 1994.

[59] Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Inf. Comput.*, 189(1):54–86, 2004.

[60] G. Phillips and T. Shepard. Static typing without explicit types (unpublished report), 1994.

[61] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, New York, NY, USA, 1994. ACM Press.

[62] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[63] Mike Salib. Faster than C: Static Type Inference with Starkiller. `http://www.python.org/pycon/dc2004/papers/1/paper.pdf`, 2004.

[64] Tim Sheard. Accomplishments and research challenges in meta-programming. *Lecture Notes in Computer Science*, 2196, 2001.

[65] Frederick Smith, David Walker, and Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

[66] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, pages 408–422, 2005.

[67] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

[68] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.

[69] David Walker and Gregory Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:117–146, 2001.

[70] Larry Wall. *Programming Perl*. O'Reilly and Associates, Inc.

[71] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 99–117. Springer-Verlag, 2001.

[72] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Constraints

A.1 Date Example

$$
\begin{array}{l}
\llbracket \mathtt{addFn} \rrbracket \le \llbracket \mathtt{this_2.add} \rrbracket \\
\llbracket \mathtt{this_4} \rrbracket \le [\mathsf{mSec} : (\llbracket \mathtt{this_4.mSec} \rrbracket, \bullet)] \\
\llbracket \mathtt{call_ret_1} \rrbracket \le \llbracket \mathtt{x_main_0.add(y_main_0)} \rrbracket \\
\llbracket \mathtt{1000} \rrbracket \le \llbracket \mathtt{x_Date_Date} \rrbracket \\
\llbracket \mathtt{this_2} \rrbracket \le [\mathsf{add} : (\llbracket \mathtt{this_2.add} \rrbracket, \bullet)] \\
\llbracket \mathtt{this_1} \rrbracket \le [\mathsf{mSec} : (\llbracket \mathtt{this_1.mSec} \rrbracket, \bullet)] \\
\llbracket \mathtt{x_Date_0} \rrbracket \le \llbracket \mathtt{this_1.mSec} \rrbracket \\
\llbracket \mathtt{x_main_0.add(y_main_0)} \rrbracket \le \left\llbracket \begin{array}{l} \mathtt{x_main_0} = \mathtt{newDate(1000);} \\ \mathtt{y_main_0} = \mathtt{newDate(100);} \\ \mathtt{x_main_0.add(y_main_0)} \end{array} \right\rrbracket \\
\llbracket \mathtt{this_3.mSec} + \mathtt{x_addFn_0.mSec} \rrbracket \le \llbracket \mathtt{this_4.mSec} = \mathtt{this_3.mSec} + \mathtt{x_addFn_0.mSec} \rrbracket \\
\llbracket \mathtt{newDate(100)} \rrbracket \le \llbracket \mathtt{y_main_0} \rrbracket \\
\llbracket \mathtt{100} \rrbracket \le \mathsf{Int} \\
\left\llbracket \begin{array}{l} \mathtt{this_1.mSec} = \mathtt{x_Date_0;} \\ \mathtt{this_2.add} = \mathtt{addFn;} \\ \mathtt{this_2} \end{array} \right\rrbracket \le \llbracket \mathtt{ret_Date} \rrbracket \\
\llbracket \mathtt{x_main_0} \rrbracket \le \llbracket \mathtt{call_this_1} \rrbracket \\
\llbracket \mathtt{newDate(1000)} \rrbracket \le \llbracket \mathtt{x_main_0} \rrbracket \\
\llbracket \mathtt{this_Date} \rrbracket = \llbracket \mathtt{this_0} \rrbracket \\
\left\llbracket \begin{array}{l} \mathtt{x_main_0} = \mathtt{newDate(1000);} \\ \mathtt{y_main_0} = \mathtt{newDate(100);} \\ \mathtt{x_main_0.add(y_main_0)} \end{array} \right\rrbracket \le \llbracket \mathtt{ret_main} \rrbracket \\
\llbracket \mathtt{this_3} \rrbracket \le [\mathsf{mSec} : (\llbracket \mathtt{this_3.mSec} \rrbracket, \circ)] \\
\llbracket \mathtt{this_main} \rrbracket = \llbracket \mathtt{this_5} \rrbracket \\
\llbracket \mathtt{addFn} \rrbracket \le \llbracket \mathtt{this_2.add} = \mathtt{addFn} \rrbracket \\
\llbracket \mathtt{this_1} \rrbracket \le [\mathsf{add} : (\llbracket \mathtt{this_1.add} \rrbracket, \circ)] \\
\llbracket \mathtt{this_addFn} \rrbracket = \llbracket \mathtt{this_3} \rrbracket \\
\llbracket \mathtt{x_Date_0} \rrbracket \le \llbracket \mathtt{this_1.mSec} = \mathtt{x_Date_0} \rrbracket \\
\llbracket \mathtt{x_main_0} \rrbracket \le [\mathsf{add} : (\llbracket \mathtt{x_main_0.add} \rrbracket, \bullet)] \\
\llbracket \mathtt{x_Date_Date} \rrbracket = \llbracket \mathtt{x_Date_0} \rrbracket \\
\llbracket \mathtt{100} \rrbracket \le \llbracket \mathtt{x_Date_Date} \rrbracket \\
\llbracket \mathtt{this_4} \rrbracket \lhd_{\mathsf{mSec}} \llbracket \mathtt{this_3} \rrbracket \\
\llbracket \mathtt{1000} \rrbracket \le \mathsf{Int} \\
\llbracket \mathtt{addFn} \rrbracket \le \left\{ \begin{array}{l} \llbracket \mathtt{this_addFn} \rrbracket \\ \llbracket \mathtt{x_addFn_addFn} \rrbracket \end{array} \right\} \to \llbracket \mathtt{ret_addFn} \rrbracket \\
\llbracket \mathtt{this_4} \rrbracket \le \left\llbracket \begin{array}{l} \mathtt{this_4.mSec} = \mathtt{this_3.mSec} + \mathtt{x_addFn_0.mSec;} \\ \mathtt{this_4} \end{array} \right\rrbracket \\
\llbracket \mathtt{this_3.mSec} \rrbracket \le \mathsf{Int} \\
\llbracket \mathtt{this_0} \rrbracket \le [\mathsf{mSec} : (\llbracket \mathtt{this_0.mSec} \rrbracket, \circ)] \\
\llbracket \mathtt{this_2} \rrbracket \lhd_{\mathsf{add}} \llbracket \mathtt{this_1} \rrbracket \\
\llbracket \mathtt{y_main_0} \rrbracket \le \llbracket \mathtt{call_param_2} \rrbracket \\
\left\llbracket \begin{array}{l} \mathtt{this_4.mSec} = \mathtt{this_3.mSec} + \mathtt{x_addFn_0.mSec;} \\ \mathtt{this_4} \end{array} \right\rrbracket \le \llbracket \mathtt{ret_addFn} \rrbracket
\end{array}
$$

$$
\begin{array}{l}
[\![\ \texttt{this\_3.mSec} + \texttt{x\_addFn\_0.mSec}\ ]\!] \ \leq\ \mathsf{Int} \\
[\![\ \texttt{this\_2}\ ]\!] \ \leq\ \left[\!\!\left[\ \begin{array}{l}\texttt{this\_1.mSec} = \texttt{x\_Date\_0;} \\ \texttt{this\_2.add} = \texttt{addFn;} \\ \texttt{this\_2}\end{array}\ \right]\!\!\right] \\
[\![\ \texttt{newDate(100)}\ ]\!] \ \leq\ [\![\ \texttt{y\_main\_0} = \texttt{newDate(100)}\ ]\!] \\
[\![\ \texttt{newDate(1000)}\ ]\!] \ \leq\ [\![\ \texttt{x\_main\_0} = \texttt{newDate(1000)}\ ]\!] \\
[\![\ \texttt{x\_main\_0.add}\ ]\!] \ \leq\ \left\{\begin{array}{l}[\![\ \texttt{call\_this\_1}\ ]\!] \\ [\![\ \texttt{call\_param\_2}\ ]\!]\end{array}\right\} \to [\![\ \texttt{call\_ret\_1}\ ]\!] \\
[\![\ \texttt{x\_addFn\_addFn}\ ]\!] \ =\ [\![\ \texttt{x\_addFn\_0}\ ]\!] \\
[\![\ \texttt{x\_addFn\_0.mSec}\ ]\!] \ \leq\ \mathsf{Int} \\
[\![\ \texttt{ret\_Date}\ ]\!] \ \leq\ [\![\ \texttt{newDate(1000)}\ ]\!] \\
[\![\ \texttt{ret\_Date}\ ]\!] \ \leq\ [\![\ \texttt{newDate(100)}\ ]\!] \\
[\![\ \texttt{this\_1}\ ]\!] \ \lhd_{\texttt{mSec}}\ [\![\ \texttt{this\_0}\ ]\!] \\
[\![\ \texttt{this\_3.mSec} + \texttt{x\_addFn\_0.mSec}\ ]\!] \ \leq\ [\![\ \texttt{this\_4.mSec}\ ]\!] \\
[\![\ \texttt{this\_3}\ ]\!] \ \leq\ [\texttt{mSec} : ([\![\ \texttt{this\_3.mSec}\ ]\!], \bullet)] \\
[\![\ \texttt{x\_addFn\_0}\ ]\!] \ \leq\ [\texttt{mSec} : ([\![\ \texttt{x\_addFn\_0.mSec}\ ]\!], \bullet)]
\end{array}
$$

A.2 Web Example

$$\llbracket\ \texttt{this\_1}\ \rrbracket \leq [\texttt{value} : (\llbracket\ \texttt{this\_1.value}\ \rrbracket, \bullet)]$$
$$\llbracket\ \texttt{checkform(this\_5)}\ \rrbracket \leq \llbracket\ \texttt{checkform(this\_5)}\ \rrbracket$$
$$\llbracket\ \texttt{htmlinput\_main\_0}\ \rrbracket \leq \llbracket\ \texttt{htmlform\_main\_1.submit}\ \rrbracket$$
$$\llbracket\ \texttt{ret\_input}\ \rrbracket \leq \llbracket\ \texttt{newinput("Infer")}\ \rrbracket$$
$$\llbracket\ \texttt{this\_2}\ \rrbracket \leq [\texttt{disabled} : (\llbracket\ \texttt{this\_2.disabled}\ \rrbracket, \bullet)]$$
$$\llbracket\ \texttt{theform\_checkform\_checkform}\ \rrbracket = \llbracket\ \texttt{theform\_checkform\_0}\ \rrbracket$$
$$\llbracket\ \texttt{ret\_form}\ \rrbracket \leq \llbracket\ \texttt{newform()}\ \rrbracket$$
$$\llbracket\ 1\ \rrbracket \leq \llbracket\ \texttt{theform\_checkform\_0.submit.disabled} = 1\ \rrbracket$$
$$\llbracket\ \texttt{this\_form}\ \rrbracket = \llbracket\ \texttt{this\_3}\ \rrbracket$$
$$\llbracket\ \texttt{this\_3}\ \rrbracket \leq [\texttt{onSubmit} : (\llbracket\ \texttt{this\_3.onSubmit}\ \rrbracket, \circ)]$$
$$\llbracket\ \texttt{this\_checkform}\ \rrbracket = \llbracket\ \texttt{this\_6}\ \rrbracket$$
$$\llbracket\ \texttt{value\_input\_0}\ \rrbracket \leq \llbracket\ \texttt{this\_1.value}\ \rrbracket$$
$$\llbracket\ \texttt{htmlform\_main\_1}\ \rrbracket \leq \llbracket\ \texttt{call\_this\_1}\ \rrbracket$$
$$\llbracket\ 0\ \rrbracket \leq \llbracket\ \texttt{this\_2.disabled}\ \rrbracket$$

$$\llbracket\ \texttt{theform\_checkform\_0.submit.value = "Calculating..."}\ \rrbracket \leq \left\llbracket\ \begin{array}{l}\texttt{theform\_checkform\_0.submit.disabled} = 1;\\ \texttt{theform\_checkform\_0.submit.value = "Calculating..."}\end{array}\ \right\rrbracket$$

$$\llbracket\ 0\ \rrbracket \leq \mathsf{Int}$$
$$\llbracket\ 0\ \rrbracket \leq \llbracket\ \texttt{this\_2.disabled} = 0\ \rrbracket$$
$$\llbracket\ \texttt{onSubmit}\ \rrbracket \leq \llbracket\ \texttt{this\_4.onSubmit}\ \rrbracket$$
$$\llbracket\ \texttt{htmlform\_main\_1.onSubmit}\ \rrbracket \leq \{\ \llbracket\ \texttt{call\_this\_1}\ \rrbracket\ \} \rightarrow \llbracket\ \texttt{call\_ret\_1}\ \rrbracket$$
$$\llbracket\ \texttt{theform\_checkform\_0.submit}\ \rrbracket \leq [\texttt{disabled} : (\llbracket\ \texttt{theform\_checkform\_0.submit.disabled}\ \rrbracket, \bullet)]$$
$$\llbracket\ \texttt{this\_1}\ \rrbracket \leq [\texttt{disabled} : (\llbracket\ \texttt{this\_1.disabled}\ \rrbracket, \circ)]$$
$$\llbracket\ \texttt{newinput("Infer")}\ \rrbracket \leq \llbracket\ \texttt{htmlinput\_main\_0}\ \rrbracket$$
$$\llbracket\ \texttt{htmlform\_main\_1}\ \rrbracket \lhd_{\texttt{submit}} \llbracket\ \texttt{htmlform\_main\_0}\ \rrbracket$$
$$\llbracket\ \texttt{htmlinput\_main\_0}\ \rrbracket \leq \llbracket\ \texttt{htmlform\_main\_1.submit} = \texttt{htmlinput\_main\_0}\ \rrbracket$$
$$\left\llbracket\ \begin{array}{l}\texttt{this\_1.value} = \texttt{value\_input\_0};\\ \texttt{this\_2.disabled} = 0;\\ \texttt{this\_2}\end{array}\ \right\rrbracket \leq \llbracket\ \texttt{ret\_input}\ \rrbracket$$
$$\llbracket\ \texttt{"Calculating..."}\ \rrbracket \leq \mathsf{String}$$
$$\llbracket\ \texttt{this\_4}\ \rrbracket \leq [\texttt{onSubmit} : (\llbracket\ \texttt{this\_4.onSubmit}\ \rrbracket, \bullet)]$$
$$\llbracket\ \texttt{this\_4}\ \rrbracket \lhd_{\texttt{onSubmit}} \llbracket\ \texttt{this\_3}\ \rrbracket$$

$$\llbracket \texttt{theform\_checkform\_0.submit} \rrbracket \leq [\texttt{value} : (\llbracket \texttt{theform\_checkform\_0.submit.value} \rrbracket, \bullet)]$$
$$\llbracket \texttt{this\_onSubmit} \rrbracket = \llbracket \texttt{this\_5} \rrbracket$$
$$\llbracket \texttt{value\_input\_0} \rrbracket \leq \llbracket \texttt{this\_1.value} = \texttt{value\_input\_0} \rrbracket$$
$$\llbracket \texttt{"Calculating..."} \rrbracket \leq \llbracket \texttt{theform\_checkform\_0.submit.value} = \texttt{"Calculating..."} \rrbracket$$
$$\llbracket \texttt{1} \rrbracket \leq \mathsf{Int}$$
$$\llbracket \texttt{checkform(this\_5)} \rrbracket \leq \llbracket \texttt{ret\_onSubmit} \rrbracket$$
$$\llbracket \texttt{this\_main} \rrbracket = \llbracket \texttt{this\_7} \rrbracket$$
$$\llbracket \texttt{newinput("Infer")} \rrbracket \leq \llbracket \texttt{htmlinput\_main\_0} = \texttt{newinput("Infer")} \rrbracket$$
$$\llbracket \texttt{htmlform\_main\_1} \rrbracket \leq [\texttt{onSubmit} : (\llbracket \texttt{htmlform\_main\_1.onSubmit} \rrbracket, \bullet)]$$
$$\llbracket \texttt{this\_2} \rrbracket \leq \left\llbracket \begin{array}{l} \texttt{this\_1.value} = \texttt{value\_input\_0;} \\ \texttt{this\_2.disabled} = \texttt{0;} \\ \texttt{this\_2} \end{array} \right\rrbracket$$
$$\llbracket \texttt{ret\_checkform} \rrbracket \leq \llbracket \texttt{checkform(this\_5)} \rrbracket$$
$$\llbracket \texttt{1} \rrbracket \leq \llbracket \texttt{theform\_checkform\_0.submit.disabled} \rrbracket$$
$$\llbracket \texttt{this\_4} \rrbracket \leq \left\llbracket \begin{array}{l} \texttt{this\_4.onSubmit} = \texttt{onSubmit;} \\ \texttt{this\_4} \end{array} \right\rrbracket$$
$$\llbracket \texttt{newform()} \rrbracket \leq \llbracket \texttt{htmlform\_main\_0} = \texttt{newform()} \rrbracket$$
$$\llbracket \texttt{htmlform\_main\_1} \rrbracket \leq [\texttt{submit} : (\llbracket \texttt{htmlform\_main\_1.submit} \rrbracket, \bullet)]$$
$$\llbracket \texttt{this\_5} \rrbracket \leq \llbracket \texttt{theform\_checkform\_checkform} \rrbracket$$
$$\left\llbracket \begin{array}{l} \texttt{this\_4.onSubmit} = \texttt{onSubmit;} \\ \texttt{this\_4} \end{array} \right\rrbracket \leq \llbracket \texttt{ret\_form} \rrbracket$$
$$\llbracket \texttt{this\_1} \rrbracket \lhd_{\texttt{value}} \llbracket \texttt{this\_0} \rrbracket$$
$$\llbracket \texttt{onSubmit} \rrbracket \leq \llbracket \texttt{this\_4.onSubmit} = \texttt{onSubmit} \rrbracket$$
$$\llbracket \texttt{"Infer"} \rrbracket \leq \mathsf{String}$$
$$\llbracket \texttt{"Calculating..."} \rrbracket \leq \llbracket \texttt{theform\_checkform\_0.submit.value} \rrbracket$$
$$\left\llbracket \begin{array}{l} \texttt{theform\_checkform\_0.submit.disabled} = \texttt{1;} \\ \texttt{theform\_checkform\_0.submit.value} = \texttt{"Calculating..."} \end{array} \right\rrbracket \leq \llbracket \texttt{ret\_checkform} \rrbracket$$
$$\left\llbracket \begin{array}{l} \texttt{htmlform\_main\_0} = \texttt{newform();} \\ \texttt{htmlinput\_main\_0} = \texttt{newinput("Infer");} \\ \texttt{htmlform\_main\_1.submit} = \texttt{htmlinput\_main\_0;} \\ \texttt{htmlform\_main\_1.onSubmit()} \end{array} \right\rrbracket \leq \llbracket \texttt{ret\_main} \rrbracket$$
$$\llbracket \texttt{newform()} \rrbracket \leq \llbracket \texttt{htmlform\_main\_0} \rrbracket$$
$$\llbracket \texttt{onSubmit} \rrbracket \leq \{ \llbracket \texttt{this\_onSubmit} \rrbracket \} \rightarrow \llbracket \texttt{ret\_onSubmit} \rrbracket$$

$$\llbracket\ \texttt{theform\_checkform\_0}\ \rrbracket \leq [\texttt{submit} : (\llbracket\ \texttt{theform\_checkform\_0.submit}\ \rrbracket, \bullet)]$$
$$\llbracket\ \texttt{this\_0}\ \rrbracket \leq [\texttt{value} : (\llbracket\ \texttt{this\_0.value}\ \rrbracket, \circ)]$$
$$\llbracket\ \texttt{htmlform\_main\_1.onSubmit()}\ \rrbracket \leq \left\llbracket \begin{array}{l} \texttt{htmlform\_main\_0} = \texttt{newform();} \\ \texttt{htmlinput\_main\_0} = \texttt{newinput("Infer");} \\ \texttt{htmlform\_main\_1.submit} = \texttt{htmlinput\_main\_0;} \\ \texttt{htmlform\_main\_1.onSubmit()} \end{array} \right\rrbracket$$
$$\llbracket\ \texttt{"Infer"}\ \rrbracket \leq \llbracket\ \texttt{value\_input\_input}\ \rrbracket$$
$$\llbracket\ \texttt{this\_2}\ \rrbracket \lhd_{\texttt{disabled}} \llbracket\ \texttt{this\_1}\ \rrbracket$$
$$\llbracket\ \texttt{this\_input}\ \rrbracket = \llbracket\ \texttt{this\_0}\ \rrbracket$$
$$\llbracket\ \texttt{htmlform\_main\_0}\ \rrbracket \leq [\texttt{submit} : (\llbracket\ \texttt{htmlform\_main\_0.submit}\ \rrbracket, \circ)]$$
$$\llbracket\ \texttt{value\_input\_input}\ \rrbracket = \llbracket\ \texttt{value\_input\_0}\ \rrbracket$$
$$\llbracket\ \texttt{call\_ret\_1}\ \rrbracket \leq \llbracket\ \texttt{htmlform\_main\_1.onSubmit()}\ \rrbracket$$

B Proof Attempts

We now outline two avenues we explored when attempting to prove Conjecture 3. Recall that Conjecture 3 asserts that a generated solution from a well-formed set of constraints will satisfy those constraints.

B.1 Proving Conjectures 1 and 2

In an attempt to prove Conjectures 1 and 2 we formulated Lemma 7. A proof of the conjectures would follow from this lemma.

Lemma 7 *For constraints* C*, type variable mappings* V_1*,* V_2 *type variables* τ_1, τ_2, τ*, if*

$$\vdash \mathsf{C}\diamond \quad (1)$$
$$\mathsf{C} \vdash \mathsf{V}_1 \simeq_\beta \mathsf{V}_2 \quad (2)$$
$$\mathsf{C}, \mathsf{V}_1, \tau_1 \rightarrow \mathsf{tp}_1 \quad (3)$$
$$\mathsf{V}_2' = \mathsf{V}_2 \bullet_{\mathsf{C}} (\tau_2, \alpha) \quad (4)$$
$$\mathsf{C}, \mathsf{V}_2', \tau_2 \rightarrow \mathsf{tp}_2 \quad (5)$$
$$\mathsf{C}, \mathsf{V}_\emptyset, \tau \rightarrow \mathsf{t} \quad (6)$$
$$\tau_1 \leq \tau_2, \tau_2 \leq \tau_1 \in \mathsf{C} \quad (7)$$

Then

$$\mathsf{tp}_1[\alpha/\mathsf{t}] \equiv_\beta \mathsf{tp}_2[\alpha/\mathsf{t}] \quad (8)$$

We now outline our attempt at proving Lemma 7. We proceed by induction on the derivations (3) and (5). In the case where $\mathsf{tp}_1 = \alpha_1$, $\mathsf{tp}_2 = \alpha_2$, $\alpha \neq \alpha_1$, and $\alpha \neq \alpha_2$ we can trivially establish $\alpha_1 \equiv_\beta \alpha_2$ with (2) and (7). In the case where tp_1 and tp_2 are object types, we can apply the inductive hypothesis to show that each member in tp_1 is congruent to tp_2. The same reasoning applies when tp_1 and tp_2 are function types. The difficulty arises in the cases where $\mathsf{tp}_2 = \alpha$ and tp_1 is a function or object type. Recall that α *cannot* be free in tp_2, thus we have to establish that $\mathsf{t} \equiv_\beta \mathsf{tp}_2$. We were unable to achieve is. The reason being that we cannot apply the inductive hypothesis, because we are unable to establish $\mathsf{C} \vdash \mathsf{V}_1 \bullet_{\mathsf{C}} (\tau_1, \alpha) \simeq_{\beta'} \mathsf{V}_2$, where β' is an extension of β. Intuitively, there is an *inbalance* between the starting environments (V_1 and V_2 in Lemma 7) that disallows the application of the

$$
\begin{array}{lcl}
\mathsf{tt} \in \mathit{ExtType} & ::= & (\alpha, \tau) \mid (\mathsf{OT}, \tau) \mid (\mathsf{GT}, \tau) \mid (\mathsf{Int}, \tau) \\
\mathsf{OT} \in \mathit{ObjExtType} & ::= & \mu\, \alpha.\mathsf{MT} \mid \mathsf{MT} \\
\mathsf{GT} \in \mathit{FuncExtType} & ::= & \mu\, \alpha.\mathsf{RT} \mid \mathsf{RT} \\
\mathsf{MT} \in \mathit{ExtObjMembers} & ::= & [(\mathsf{m} : \mathsf{ttm})^*] \\
\mathsf{ttm} \in \mathit{MemberExtType} & ::= & (\mathsf{tt}, \psi) \\
\mathsf{RT} \in \mathit{ExtFuncRow} & ::= & ((\mathsf{OT}, \tau) \times \mathsf{tt} \to \mathsf{tt})
\end{array}
$$

Figure B.1.: Syntax of Extended Types

inductive hypothesis, *i.e.* between the sub-derivations relating to the translation of τ_1 with the translation of τ_2.

B.2 Extended Types

In this attempt we introduced the concept of *extended types*, tt, defined in Figure B.1, which are pairs of pre-types and type variables. Intuitively, extended types associate a type variable with each type. Figure B.2 shows the translation relation for extended types. The definition of free variables is extended so that a free variable is an extended type comprised of a variable and a type variable: (α, τ). Function $\mathcal{R}$ is used to remove the type variables from an extended type to give a type. We omit the definitions of free variables and function $\mathcal{R}$ as they are straightforward.

We now outline some of the properties we proved using extended types. With Lemma 8 we prove an *extended* version of Lemma 5. As well as showing the translation is deterministic, we also assert that if there are free variables in tt, then there are *corresponding* free variable in tt′. Furthermore, the type variables associated with the free variables, τ_1, τ_2, are equivalent: $\tau_1 \leq \tau_2, \tau_2 \leq \tau_1 \in \mathsf{C}$.

Lemma 8 *For constraints* C*, type variables* τ, τ'*, type variable functions* V, V' *if:*

$$\vdash \mathsf{C}\diamond \tag{1}$$
$$\tau \leq \tau', \tau' \leq \tau \in \mathsf{C} \tag{2}$$
$$\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tt} \tag{3}$$
$$\mathsf{C}, \mathsf{V}', \tau' \to \mathsf{tt}' \tag{4}$$
$$\mathsf{C} \vdash \mathsf{V} \simeq_\beta \mathsf{V}' \tag{5}$$

Then for all τ_1*:*

$$\mathcal{R}(\mathsf{tt}) \equiv_\beta \mathcal{R}(\mathsf{tt}') \tag{6}$$
$$(\mathsf{V}(\tau_1), \tau_1) \in \mathcal{FV}(\mathsf{tt}, \mathsf{C}) \implies \begin{array}{c} \exists\, \tau_2 : ((\mathsf{V}'(\tau_2), \tau_2) \in \mathcal{FV}(\mathsf{tt}', \mathsf{C}) \wedge \\ \tau_1 \leq \tau_2, \tau_2 \leq \tau_1 \in \mathsf{C}) \end{array} \tag{7}$$

$$\frac{\mathsf{V}(\tau) = \alpha}{\mathsf{C}, \mathsf{V}, \tau \to (\alpha, \tau)}\,(var) \qquad \frac{\tau \ \leq\ \mathsf{Int} \in \mathsf{C}}{\mathsf{C}, \mathsf{V}, \tau \to (\mathsf{Int}, \tau)}\,(int) \qquad \frac{\begin{array}{l} \tau \ \leq\ [\mathsf{m} : \_\,] \notin \mathsf{C} \\ \tau \ \leq\ (\_ \times \_ \to \_) \notin \mathsf{C} \\ \tau \ \leq\ \mathsf{Int} \notin \mathsf{C} \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to (\mu\, \alpha.[\,], \tau)}\,(unknown)$$

$$\frac{\begin{array}{l} n \geq 1 \\ \{\mathsf{m}_1 ... \mathsf{m}_n\} = \{\mathsf{m} \mid \tau \ \leq\ [\mathsf{m} : (\_, \_)] \ \in\ \mathsf{C}\ \} \\ \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha)\ \ (\alpha \notin Range(\mathsf{V})) \\ \tau \ \leq\ [\mathsf{m}_i : (\tau_i, \_)] \ \in\ \mathsf{C}\ (for\ i\ \in 1...n) \\ \mathsf{C}, \mathsf{V}_0, \tau_i \to \mathsf{tt}_i \\ \psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to (\mu\, \alpha.[\mathsf{m}_1 : (\mathsf{tt}_1, \psi_1) ... \mathsf{m}_{\mathtt{n}} : (\mathsf{tt}_{\mathtt{n}}, \psi_{\mathtt{n}})], \tau)}\,(objtype)$$

$$\frac{\begin{array}{l} \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \bullet_{\mathsf{C}} (\tau, \alpha)\ \ (\alpha \notin Range(\mathsf{V})) \\ \tau \ \leq\ (\tau_1 \times \tau_2 \to \tau_3) \in \mathsf{C} \\ \mathsf{C}, \mathsf{V}_0, \tau_i \to \mathsf{tt}_i\ \ \ (for\ i\ \in 1...3) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to (\mu\, \alpha.(\mathsf{tt}_1 \times \mathsf{tt}_2 \to \mathsf{tt}_3), \tau)}\,(functype)$$

Figure B.2.: Translation from Type Variables to Extended Types

Lemma 9 asserts that if two pre-types are congruent up to a renaming, then we can replace occurrences of free variables with other congruent types.

Lemma 9 *For type,* t, t'*, pre-types* $\mathsf{tp}, \mathsf{tp}'$ *, variable renaming* β*, variables* α*,* α'*, if:*

$$\mathsf{t} \equiv \mathsf{t}' \qquad (1)$$
$$\mathsf{tp} \equiv_{\beta,\alpha\mapsto\alpha'} \mathsf{tp}' \qquad (2)$$
$$\alpha \in \mathcal{FV}(\mathsf{tp}) \qquad (3)$$
$$\alpha' \in \mathcal{FV}(\mathsf{tp}') \qquad (4)$$

Then

$$\mathsf{tp}[\alpha/\mathsf{t}] \equiv \mathsf{tp}'[\alpha'/\mathsf{t}'] \qquad (5)$$

Proof: The proof is a straightforward induction on $\mathsf{t} \equiv \mathsf{t}'$. □

Lemma 10 asserts that all the type variables associated with occurrences of a free variable in an extended type are equivalent.

Lemma 10 *For constraints* C*, type variable function* V*, type variables* τ, τ'*, if:*

$$\mathsf{C} \vdash \mathsf{V} \qquad (1)$$
$$(\mathsf{V}(\tau'), \tau') \in \mathcal{FV}(\mathsf{tt}, \mathsf{C}) \qquad (2)$$
$$\mathsf{V}(\tau') = \mathsf{V}(\tau) \qquad (3)$$

Then

$$\tau \leq \tau', \tau' \leq \tau \in \mathsf{C} \qquad (4)$$

Proof: The proof follows from Definition 7. □

Using the properties above, we are able to prove all the cases of Conjecture 3 with the exception of cases $\mathsf{C} = \tau \leq [\mathsf{m} : (\tau', \psi)]$ and $\mathsf{C} = \mathsf{c} = \tau \leq (\tau_1 \times \tau_2 \to \tau_3)$. In the cases for constraints of the form $\tau \leq \tau'$, $\tau \lhd_{\mathsf{m}} \tau'$, and $\tau \lhd_{*} \tau'$ we can use Lemmata 8, 10, and 9. In the cases for constraints of the form $\mathsf{c} = \tau^{\circ}$ and $\tau^{\mathbf{obj}}$ the proof is straightforward. However, in the cases for constraints of the form $\mathsf{C} = \tau \leq [\mathsf{m} : (\tau', \psi)]$ and $\mathsf{C} = \mathsf{c} = \tau \leq (\tau_1 \times \tau_2 \to \tau_3)$ we were unable to use any of the lemmata above. In the former case we are unable to establish agreement between the environments used to translate τ and τ'. Similarly, for the later case, we are unable to show agreement between the environments used to translate τ, τ_1, τ_2 and τ_3. Therefore, we are unable to make use of Lemma 8 in these cases.