Dynamic type inference for JavaScript

by **Nico Naus**
ICA-3472353

August 31, 2015

Supervisor Freiburg University
Prof. Dr. P. Thiemann
First Supervisor Utrecht University
Dr. A. Dijkstra
Second Supervisor Utrecht University
Prof. Dr. J.T. Jeuring

Department of Computing Science, Utrecht University, The Netherlands

Abstract

With this master thesis, we have shown that Dynamic Type Inference is a feasible method for doing type inference for JavaScript. The idea of Dynamic Type Inference is to observe the program at run-time, and generate type constraints from the observations. With these constraints, types for the program can be inferred.
To demonstrate this, we have developed both a formal and practical implementation. Our formal system consists of a core JavaScript language and an instrumented semantics. The semantics collects the constraints and infers the types for us. The types inferred by this system are shown to be sound.
Our practical implementation does the same as our formal system, but works on full JavaScript. We have evaluated this system by running it on the SunSpider benchmark. This resulted in useful types, type errors and type warnings.

Acknowledgements

First and foremost, I want to thank Professor Peter Thiemann from Freiburg University. I am very thankful to him for providing me with the opportunity to write my master thesis under his supervision. He was able to challenge me and supported me where needed. Our weekly meetings and his insights were essential to this thesis.
Secondly, I want to thank my colleagues from the Progλang group: Luminous Fennell, Manuel Geffken, Robert Jakob and Matthias Keil, for all their interesting insights, challenging questions, proofreading and daily support.
I also want to thank Dr. Atze Dijkstra for his supervision and useful questions and remarks on this work.

Lastly, I want to thank my parents, friends and boyfriend for their support during my stay in Freiburg, Germany.

The Erasmus+ program supported this master thesis with a generous grant.

Contents

1 Introduction

JavaScript (JS) has been around for almost 20 years. The initial version was created by Brendan Eich in only 10 days, introduced by Netscape in 1995. It was standardized as ECMAscript in 1997. It was intended to be used as a "glue language", but today, it has become much more than that. JavaScript is used on almost all modern websites, some of which have become full-blown applications.
Since the use of JavaScript is so omnipresent, it is important to have good development tools. These tools can support programmers and help them write sound JavaScript code. One of these tools is type analysis.

1.1 Research Question

The aim of this Master Thesis is to develop a type inference algorithm that can be used on JavaScript programs. We allow programmers to supply type annotations for their program code. The algorithm will then verify if these annotations coincide with the types found in the actual program. If no type annotations are provided, the algorithm will infer types to discover type inconsistencies.

1.2 Related work

Quite some work has been done on bringing type checking and type inference to object oriented dynamic languages such as Ruby, Python and JavaScript.

Anderson and Giannini describe a formal static type system for JavaScript [7]. This work builds upon previous work by the same authors together with Drossopoulou [8]. They use a core language with very limited syntax and construct a type system for it. They show that this type system is sound. Unfortunately, their work does not include a practical implementation.

Thiemann lays the groundwork for a static type system for JavaScript [19]. In this work, he too presents a core JavaScript language. This represents a restricted version of JavaScript. For this core JavaScript, types and a syntax for typing are defined. This core JavaScript and typing rules are used in later work by him together with Jensen and Møller [15]. In this work, the actual static analysis algorithm is constructed. This system is proven to be a complete and sound type analysis for JavaScript. Their method is based on the monotone framework. Flow graphs are constructed and analysis lattices and transfer functions are presented. The downside of their method is that it is quite intricate, and therefore hard to implement.

Furr et al. introduce a static type inference algorithm for Ruby [10]. Their method, called DRuby, is similar in complexity to the aforementioned systems for JavaScript. The authors have slightly reduced the burden for their implementation by compiling Ruby to an intermediate language, which has an explicit flow. The authors also created

trusted type annotations for the core library of Ruby and allow programmers to provide trusted type annotations to their programs. DRuby also supports intersection types, varargs, union types, parametric polymorphism, and several other complex types.

For Python, Michael Salib developed Starkiller, a comprehensible static type inference system [17]. Starkiller aims to remove the burden of constantly checking types at runtime before any operation is done. His method is based on work by Agesen for the programming language Self, influenced by Smalltalk [4].

All these approaches are however static analyses. JavaScript is a dynamic language, and a lot of properties of programs including types are only known at runtime. As noted by Jakobs et al. [14], static analysis either yields many false positives or restricts the expressiveness of the language. Cartwright and Fagan introduce the concept of Soft Typing to overcome these limitations [9]. They argue that both static and dynamic typing have their drawbacks and that soft typing could potentially combine the best of both. The idea is, in order to inference types for a dynamically typed language, to do some static type inference using Hindly-Milner first and insert dynamic checks in cases where static inference falls short. Cartwright together with Wright, implemented such a system for Scheme [20].

Soft typing has also been applied to JavaScript. Hackett and Guo present an implementation of soft typing for JavaScript in SpiderMonkey [12]. Their hybrid inference algorithm first performs a static "may have type" analysis on the program. This analysis generates constraints and identifies at what points in the program the constraints may be incomplete. Then using this information, type barriers are inserted in the program. During the execution, a "must have type" analysis is performed, using the previously inserted information. The type information is used to reduce the running time of the program. Since the type of a variable is known in advance, you can potentially omit runtime type checks. The only information reported back to the programmer is how many times a dynamic check was needed, this could possibly indicate weak code.
An obvious downside to hybrid approaches like soft typing is that you still have to develop a very complex static type inference system.

Currently, there is only one paper available on purely dynamic type inference algorithms for JavaScript. Pradel et al. [16] present a dynamic type inconsistency analysis for JavaScript, called TypeDevil. Their system is an implementation on top of the Dynamic Analysis Framework Jalangi [18], and checks JavaScript programs for inconsistent types, where inconsistent is defined as a property having more than one type. These warnings are then pruned based on some heuristic the authors came up with. They don not present a complete type inference system however, and have only developed a practical implementation. An et al. do present a complete dynamic inference algorithm with a formal development, not for JavaScript, but for Ruby [6]. They note that doing a dynamic analysis has several benefits. Implementing such an analysis is much easier and

less error prone than a static or hybrid one, since one does not have to capture the whole language and every possible flow. Furthermore, the result we obtain is more precise.

1.3 Approach

Instead of doing a static or hybrid analysis, we develop a dynamic analysis. This approach is inspired by previous work for Ruby [6]. In this work, the authors developed a dynamic type inference algorithm for said programming language. By running test inputs and inspecting the program behavior, they infer the types of the program.
Their method first instruments the program code in such a way that it can be run, given some test set, and so that constraints can be extracted from these runs. The instrumented code is then actually being run and these constraints are then collected. Lastly, the algorithm uses these constraints to infer the types for the program.

This approach has several benefits over previous work. Static analyses are only able to provide limited information. They can result in too general types since they do not consider how code is actually being used. Furthermore, a complete static analysis is hard to implement. You have to completely model the whole flow of the language, which can become very complex really fast, as shown by both Thiemann (et al.) and Furr (et al.). It took two papers with four years in between to develop a full static analysis implementation prototype. As mentioned, hybrid - soft typing - methods have been used to overcome the first shortcoming, the imprecise results. But this solution is still very complex since a static analysis needs to be performed first. Furthermore, the implementation by Hackett is only of limited use to the programmer, since it only reports where types cannot be inferred statically, not the types itself.

We develop an adaptation of the previously mentioned dynamic type inference algorithm for Ruby. By doing the whole analysis dynamically, we get more precise results and overcome the burden of having to define the complete flow of the language.

First we will discuss what features we would like our implementation to have. Afterwards, we will discuss how we got there, first formally and then in practical implementation. Lastly, we will describe how to validate our work.

1.3.1 Features

We want our algorithm to return two kinds of types. In the first place, we want to infer basic types. These will include: `Undefined`, `Null`, `Bool`, `String`, `Number`, `Function` and `Object`. The last type, `Object`, is a structural type. This type describes what properties an object must have.

Furthermore, we suspect that the values that are being passed to a method or function can influence the behavior of a function. Values like `0`, `false` or `""` could potentially result in aberrant behavior. Therefore, we would also like to track values, up to a certain

level. It is evident that if both `true` and `false` are passed as an argument, the type `Bool` is sufficient. These singleton types were taken from Thiemann [19].

The programmer will be able to annotate his code with types, so that our algorithm can verify these types for the programmer. This prevents unintended, incorrect use of code and improve software quality. We want our program to report type errors and inconsistency warnings, if there are any. Otherwise, we expect our analysis to give the inferred types for the code.

Both the singleton types and the type annotations will only be implemented in the practical system.

1.3.2 Formal development

In order to be able to reason about the algorithm we want to implement, we first develop it formally. We define a core JavaScript language to make reasoning more comprehensible. This core language is based on λ_{JS} developed by Guha et al [11]. λ_{JS} is a very complete core language, and includes language concepts that are not essential for a basic implementation. For example, while, arrays, let, deletion of properties, prototypes, etc. We will omit these expressions from the core language. The very basic core language will be similar to core languages used by Anderson [7], Heidegger [13] and Thiemann [19].

We will extend the regular language semantics as stated in the λ_{JS} paper, to yield type constraints. This is similar to the work done for Ruby [6]. These constraints are then used to infer types for our program.

1.3.3 Implementation

For the practical implementation, we will develop a system similar to TypeDevil[16]. First, we instrument the code with constraint collection. Then constraints will be generated by running the instrumented code. Lastly, we will infer types from these constraints and report back to the user. We extend upon the aforementioned work by including a way for developers to annotate their program with types. This allows us to not only return warnings and the inferred types, but also type errors. We implement our system using Jalangi2, a dynamic analysis framework for JavaScript.

1.3.4 Validation

Since we have produced two systems, a formal one and a practical implementation, we will need to validate both.

For our formal system, we will show that soundness holds. This approach is also used by several other authors [6, 7, 15, 19].

The implementation of the algorithm will be tested by running benchmark programs. This approach is also similar to the approach taken in other work [6, 15].
The most significant result of the system will be the resulting types. We assume that there is no type information available for our benchmark programs. Therefore, we have to manually add type annotations to validate that type verification against trusted annotations works properly.

2 Formal system

In this chapter, a formal dynamic type inference system for JavaScript will be presented. The goal of this system is to show how dynamic type inference would operate and allow logic reasoning about such a system. It allows us to prove correctness of our approach. Our formal system will consist of an augmented semantics for a core JavaScript language. This semantics will collect constraints and in the end use these constraints to infer types.

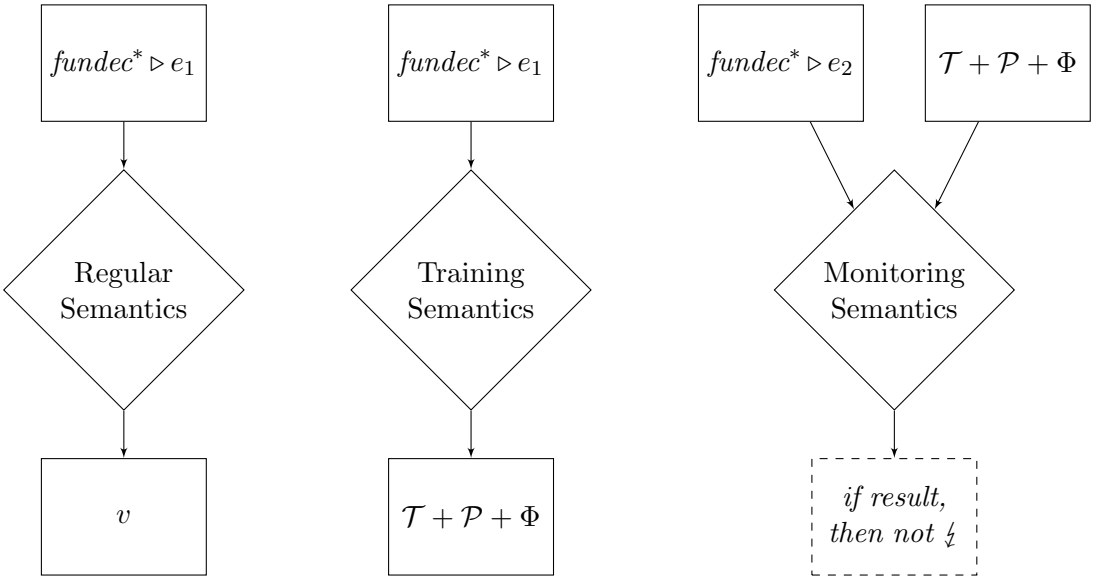

Figure 1: Schematic overview of our formal system

Figure 1 gives an overview of this formal system. On the left, a schematic view of the normal execution of a program is shown. We have our program at the top, and then using some semantics we evaluate the program to get back a result. Our dynamic type inference system is shown in the middle. Here, we have augmented the semantics to also collect constrains. Besides just the result, we also get back the inferred types and the paths of the functions in the program. Note that we do not alter the execution with respect to the regular semantics, we just observe.
Then lastly, on the right side, an overview of the monitoring semantics is given. The purpose of the monitoring semantics is to prove soundness of the types generated by the training semantics. It takes the same program, but with a different input, simulated by the top level expression e_2, and the coverage of the training semantics. Then we will prove that if these semantics give back a result, this result is not crash. Thereby we prove soundness.

Figure 2 shows an example program, written in the Core JavaScript language that will be defined in the next section. On the right are the constraints generated by the

```
1  function test(x){,
2    return{
3      if(x.val)                          1_arg<=[val:1_arg_val]
4        then x.val = inc(x.val)
5        else x.val = 1                   1_arg<=[val:1_arg_val],Num<=1_arg_val
6    }
7  }
8  function inc(x){,
9    return x+1
10 }
11 function main(x){var b c d,
12   return {
13     b = new null;
14     b.val = 0;                         13<=[val:13_val], Num<=13_val
15     c = new b;                         15<=13
16     c.b = new b;                       15<=[b:15_b], 16<=15_b, 16<=13
17     d = test(b);                       1<=1_arg->1_ret, 13<=1_arg, Num<=1_ret
18     inc(d);}}                          8<=8_arg->8_ret, 1_ret<=8_arg, Num<=8_ret
-------------------------------------------------------------------------------
e = main();                             11<=11_arg->11_ret, ()<= 11_arg, 8_ret<=11_ret
```

Figure 2: Example program written in our CoreJS language, with generated constraints

type inference system. Objects and functions are uniquely identified with the line number they were created on. For example, on line 13 we create a new object and store it in variable `b`. When we assign to it, we see that the constraints in line 14 in the right column refer to the object as "13".

Constraints are generated at the following points.

Function calls such as the ones at line 17 and line 18, constrain the object to be a function, the argument type to be a subtype of the function's argument type and the return type to be a subtype of the function's return type.

New object created from a prototype, at line 15 for example. The type variable of the new object must be a subtype of the type variable of the prototype object. That is why we do not create a constraint at line 13, there is no prototype object.

Property read and writes also generate constraints, as shown on line 3, 5 and 16.

Note that the function `inc` does not generate any constraints, since it only accesses its local variables. The function `test` only generates constraints for the else branch, since we do not visit the true branch.
After execution, you can infer the type of every object using these constraints.

In the coming section, we will formally define the core JavaScript language. This represents a subset of JavaScript. This is followed by a type language. Then, semantic objects are introduced, together with the semantics itself. We will present the three

Expressions

$e ::= c \mid x \mid fundec^{\ell} \mid pr(e) \mid \texttt{new}^{\ell}\ e \mid x = e \mid e.n \mid e; e$

$\mid e.n = e \mid \texttt{if}^{\ell}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e(e)$

Constants

$c ::= num$	number
$\mid str$	string
$\mid bool$	boolean
$\mid \texttt{null}$	null
$\mid \texttt{udf}$	undefined

Function declaration

$fundec^{\ell} ::= \texttt{fun}^{\ell}\ f(x)\{\texttt{var}\ y^*, \texttt{return}\ e\}$

Names

$x, f, n \in$ set of names

Program

$prog ::= fundec^* \triangleright e$

Figure 3: Syntax of our core JavaScript language

different semantics as shown in Figure 1. A regular semantics for standard execution, a training semantics that dynamically infers types and a monitoring semantics that will be used to prove soundness of the training semantics. Lastly, a soundness theorem and proof is presented for this system.

2.1 Core JavaScript Language

We desire from our core JavaScript language that it is easily comprehensible, compact, and as similar as possible to actual JavaScript. The syntax for our core JS language is listed in figure 3. The language consists of expressions. These expressions can either be a constant value c, variable read, variable write, function declaration, primitive operation, new, property read, property write, sequence, if-then-else or function application. A program is defined as a set of top level function bindings together with some initial expression.

Later on, we want to be able to record what path the execution takes. In order to do this, we attach a unique label ℓ to every if-then-else statement. How this will work exactly will become clear in the following sections.
We also attach this unique label ℓ to new expressions and function declarations. We do this to track the creation site of objects and functions. This is used later by our semantics.

Sumtypes

$$\tau ::= \sum_{i \in T, T \subseteq \{\bot, u, b, s, n, o, f\}} \varphi_i$$

Rows

$$\varrho ::= \ str : \tau, \varrho \mid \ str : \tau$$

Type summands

$$\varphi_\bot ::= \texttt{Udf}$$
$$\varphi_u ::= \texttt{Null}$$
$$\varphi_b ::= \texttt{Bool}$$
$$\varphi_s ::= \texttt{String}$$
$$\varphi_n ::= \texttt{Number}$$
$$\varphi_f ::= \texttt{Function}(\tau \to \tau)$$
$$\varphi_o ::= \ \texttt{Obj}(\varrho)$$

Figure 4: Definition of Types

Work by Guha[11], Heidegger et al.[13], Anderson[7] and Thiemann[19] has inspired this particular core JavaScript language. In order to keep the formal system to be as simple as possible, only essential elements of full JavaScript are present in this core language.

The biggest omission from JavaScript is the bracket notation for property access. Properties can only be accessed by $e.n$, where n is a property name, predefined in the source code of the program. In full JavaScript, programmers are also able to access properties by writing $e[x]$, where x is a variable. We omit this notation to get our dynamic type inference off the ground. This means that property names must be predefined in the source code. Variable strings are not allowed.

Some other notable exclusions are while-statements, arrays, let-statements and deletion of properties.

2.2 Type Language

The language of types, shown in Figure 4, consists of a set of base types and row types. The row types are used as structural types for objects. This type language an adaptation of the type language used by Thiemann[19]. Our types are very straight forward. We have the basic types that are also present in JavaScript: `Udf`, `Null`, `Bool`, `String` and `Number`. Then we have Function types, defined as holding two inner types. Finally we have Object types, which are defined as a row type. A row type consists of one or more property names paired with a type. An example of an object type can be $\texttt{Obj}(\text{"}a\text{"} : \texttt{Number}, \text{"}b\text{"} : \texttt{String})$.

2.3 Semantics

In this section, we will introduce three kinds of semantics for our core JavaScript language. First, a regular, untyped, big step semantics. Then in the next section, this semantics will be augmented with constraint collection. Lastly, a monitoring semantics

$$
\begin{aligned}
\text{heaps } H &::= (l \mapsto \text{obj})^* \\
\text{activation record } S &::= (x \mapsto v)^* \\
\text{values } v &::= l \mid c \\
\text{object obj} &::= (v, (n \mapsto v)^*) \\
\text{wrapped values } \omega &::= v : \bar{\tau} \\
\text{abstract types } \bar{\tau} &::= \tau \mid \alpha \\
\text{paths } \Phi &::= \phi^* \\
\text{path } \phi &::= p^* \\
\text{literal } p &::= \ell \mid \neg\ell \\
\text{constraints } C &::= (\tau \leq \tau')^* \\
\text{Falsey} &::= \texttt{udf} \mid \texttt{null} \mid 0 \mid \text{""} \mid \texttt{false}
\end{aligned}
$$

$l \in$ Heap address
$\alpha \in$ Type variable
$n \in$ Property names

Figure 5: Semantic objects for our core JavaScript language

will be introduced. But first, we need some semantic objects. These objects represent state, values, types, constraints and paths.

2.3.1 Semantic objects

Our semantic objects are listed in Figure 5. Just like in regular JavaScript, we keep two pieces of state. One is the heap, which maps locations to objects, the other is an activation record. The activation record maps variables to values.
Values can be either a heap location or a constant. Objects are defined as shown in Figure 5. In the first position, they hold a reference to their prototype object. The remainder is just a mapping from property names to values. The properties "$fun", "$vars" and "$tyvar" are reserved and cannot be used by the programmer. Their use will become clear in the next section.
Furthermore, we also have Φ, sets of paths, where a single path ϕ is defined as one or more literals p.
C contains a list of one or more constraints, where constraints are of the form $\tau \leq \tau'$, meaning τ is a subtype of τ'.
Finally, we define the set "Falsey", a term regularly used by JavaScript programmers to indicate the set of all values that are equivalent to false.

2.3.2 Regular Semantics

In Figure 6 and 7, the semantics of regular execution for our core JavaScript language is listed. The big-step reduction judgments are of the form $H; S; e \longrightarrow H'; S'; v$. Under heap H and activation record S, the expression e reduces to the value v and returns the new heap H' and activation record S'.
Most rules are standard. Variables are looked up directly in the activation record. PCall performs primitive operations, and is assumed to only return non-object values. Variable assignment is performed by updating the activation record. Sequence is performed by

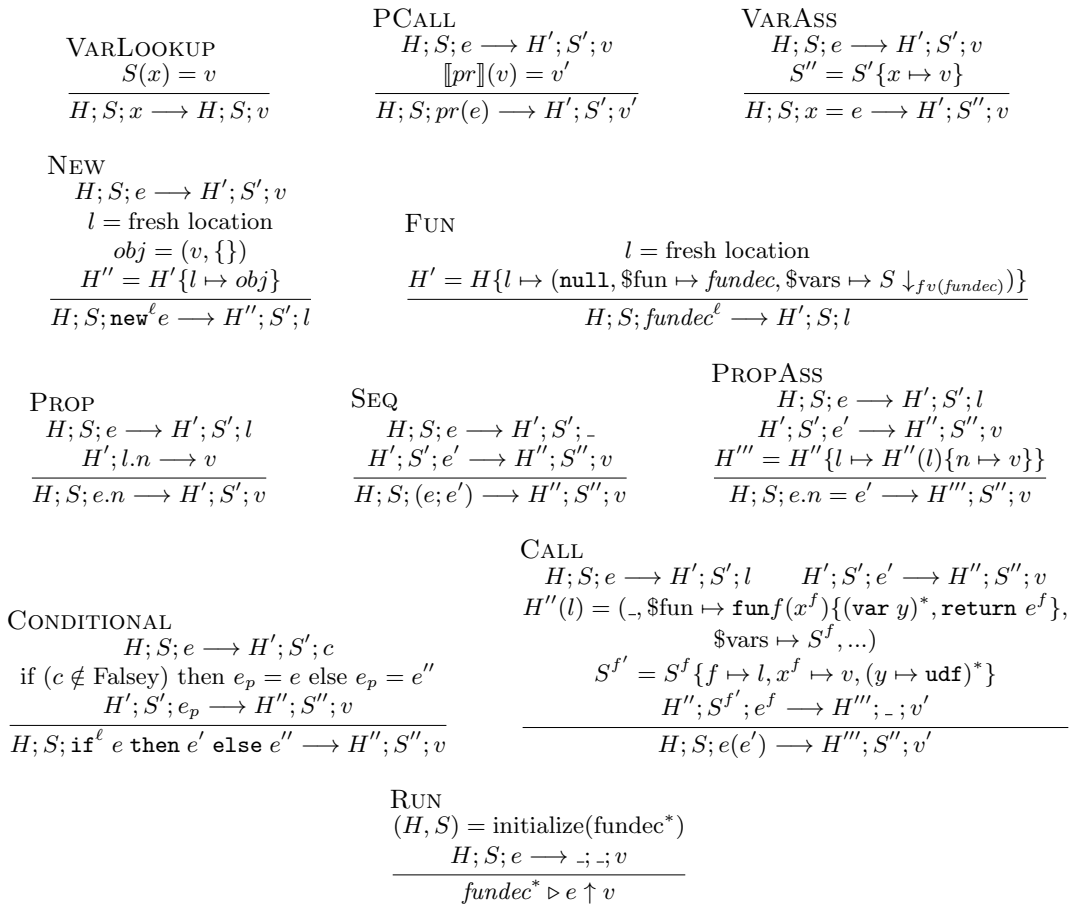

Figure 6: Regular semantics for executing our core JavaScript language

first evaluating the first expression, then the second, and returning the value of the second. Conditional checks if the condition evaluates to false, and acts accordingly, as expected.

Function literals are converted to objects by the Fun-rule. They do not have a prototype, store the actual function in the "$fun" property and the free variables from e are stored in "$vars". This treatment of functions corresponds to the semantics of actual JavaScript. Prototypes are set when a new object is created using the New rule. We explicitly allow creating a new object from either an object or just a regular value. When a regular value is used, for example `null`, the object has no prototype. Prototype look-up is performed by PropLookup and ProtoLookup, when a property of an object is requested in the Prop and MCall rule.

Call also deserves some extra explanation. From the heap, we retrieve the desired function. As mentioned above, this is an object. We construct a new activation record by taking the bound variables in "$vars" and adding references to the function for recursive

$$\textsc{PropLookup}\ \frac{H(l)[n] = v}{H; l.n \longrightarrow v} \qquad \textsc{ProtoLookup}\ \frac{n \notin H(l) \qquad H; H(l)_{proto}.n \longrightarrow v}{H; l.n \longrightarrow v}$$

Figure 7: Judgements for prototype lookup

calls, to the argument and to local variables. We then execute the actual function with this new activation record.

In order to execute programs, we need to do some extra work do deal with the top level functions. Here, at the top of our derivation, we have the RUN-rule. The initialize-function in this rule retrieves all the top level function identifiers and initializes them in the activation record. This allows top level functions to be mutually recursive. Then in the next step, bind uses the FUN-rule to convert the top level functions to objects (under S), stores them in the heap and binds the location to the variable in the activation record.
Consider the following small program.

`fun` $a\{$`var` $x,$ `return` $b($`true`$)\};$
`fun` $b\{$`var` $y,$ `return if` y `then` y `else` $a(y)\};$

The variable b in function a is free in a. Suppose we process the function declarations sequentially. That means that when creating the closure, variable b will be looked up in the current activation record S. But it is empty, since a is the first function we process. Starting with function b gives the same problem, since the functions are mutually recursive. Because we want to allow mutual recursion, we first need to initialize all top level function identifiers with a fresh heap location in the current activation record. Then we create the function closure for each declaration and store it at the previously assigned location.

2.3.3 Training

Now that we have defined the regular semantics for our core JavaScript language, we augment this semantics with type constraint collection. The semantics are listed in Figure 8 and 9. All values are now wrapped with their type or a type variable. Furthermore, we collect constraints and record what paths are taken. These changes do not alter the actual execution of the program, they merely collect information about the execution.

The reduction judgments are now of the form $H; S; e \longrightarrow H'; S'; \omega \mid C; \phi; \Phi$. The C-component contains the constraints collected during evaluation, ϕ contains the path the evaluation is currently on and Φ contains paths collected during evaluation.

$$\textsc{TVarLookup}\ \frac{S(x) = \omega}{H; S; x \longrightarrow H; S; \omega \mid \{\}; \{\}; \{\}} \qquad \textsc{TPCall}\ \frac{H; S; e \longrightarrow H'; S'; v : \_ \mid A \quad [\![pr]\!]v = \omega}{H; S; pr(e) \longrightarrow H'; S'; \omega \mid A}$$

$$\textsc{TNew}\ \frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; v : \bar{\tau} \mid C; \phi; \Phi \quad l = \text{fresh location} \quad \alpha' = \ell \\ \text{if } (\bar{\tau} = \alpha) \text{ then } (C' = \alpha' \leq \alpha) \text{ else } (C' = \{\}) \\ obj = (v : \bar{\tau}, \{\}) \quad H'' = H'\{l \mapsto obj : \alpha'\} \end{array}}{H; S; \texttt{new}^{\ell} e \longrightarrow H''; S'; l : \alpha' \mid C, C'; \phi; \Phi} \qquad \textsc{TVarAss}\ \frac{H; S; e \longrightarrow H'; S'; \omega \mid A \quad S'' = S'\{x \mapsto \omega\}}{H; S; x = e \longrightarrow H'; S''; \omega \mid A}$$

$$\textsc{TFun}\ \frac{l = \text{fresh location} \quad \alpha = \ell \quad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \$\text{vars} \mapsto S \downarrow_{fv(fundec)}) : \alpha\}}{H; S; fundec \longrightarrow H'; S; l : \alpha \mid \{\}; \{\}; \{\}}$$

$$\textsc{TProp}\ \frac{H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \quad C' = \alpha \leq [n : \alpha.n] \quad H'; l.n \longrightarrow \omega}{H; S; e.n \longrightarrow H'; S'; \omega \mid C, C'; \phi; \Phi} \qquad \textsc{TPropAss}\ \frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \quad H'; S'; e' \longrightarrow H''; S''; v : \bar{\tau} \mid C'; \phi'; \Phi' \\ C'' = \alpha \leq [n : \alpha.n], \bar{\tau} \leq \alpha_n \quad H''' = H''\{l \mapsto H''(l)\{n \mapsto v : \alpha.n\}\} \end{array}}{H; S; e.n = e' \longrightarrow H'''; S''; v : \alpha.n \mid C, C', C''; \phi, \phi'; \Phi, \Phi'}$$

$$\textsc{TConditional}\ \frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; c : \tau \mid C; \phi; \Phi \\ \text{if } (c \notin \text{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \quad H'; S'; e_p \longrightarrow H"; S''; \omega \mid C'; \phi'; \Phi \end{array}}{H; S; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \longrightarrow H''; S''; \omega \mid C, C'; \phi, p, \phi'; \Phi, \Phi'}$$

$$\textsc{TSeq}\ \frac{H; S; e \longrightarrow H'; S'; \_ \mid C; \phi; \Phi \quad H'; S'; e' \longrightarrow H''; S''; \omega \mid C'; \phi'; \Phi'}{H; S; (e; e') \longrightarrow H''; S''; \omega \mid C, C'; \phi, \phi'; \Phi, \Phi'}$$

$$\textsc{TCall}\ \frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \quad H'; S'; e' \longrightarrow H''; S''; v : \bar{\tau} \mid C'; \phi'; \Phi' \\ H''(l) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \$\text{vars} \mapsto S^f, ...) : \alpha \\ S^{f'} = S^f\{f \mapsto l : \alpha, x^f \mapsto v : \alpha_{arg}, (y \mapsto \texttt{udf : Udf})^*\} \quad C^{call} = \alpha \leq \alpha_{arg} \longrightarrow \alpha_{ret}, \bar{\tau} \leq \alpha_{arg} \\ H''; S^{f'}; e^f \longrightarrow H'''; \_; v' : \bar{\tau}' \mid C''; \phi''; \Phi'' \quad C^{ret} = \bar{\tau}' \leq \alpha_{ret} \end{array}}{H; S; e(e') \longrightarrow H'''; S''; v' : \alpha_{ret} \mid C, C^{call}, C', C^{ret}; \phi, \phi'; \Phi, \Phi', \phi'', \Phi''}$$

$$\textsc{TRun}\ \frac{(H, S) = \text{initialize}(\text{fundec}^*) \quad H; S; e \longrightarrow \_; \_; \_ \mid C; \_; \Phi}{fundec^* \triangleright e \uparrow (T, \mathcal{P}) = \text{Solve}(\text{C}); \Phi}$$

Figure 8: Training semantics

$$\frac{H(l)[n] = \omega}{H; l.n \longrightarrow \omega}\ \textsc{TPropLookup} \qquad \frac{n \notin H(l) \qquad H; H(l)_{proto}.n \longrightarrow \omega}{H; l.n \longrightarrow \omega}\ \textsc{TProtoLookup}$$

Figure 9: Judgements for prototype lookup in Training semantics

Type constraints are generated in four rules:

TNew When a new object is created, it should have at least the same type as its prototype, but can have more properties.

TProp When we look up a property, we want it to actually be there.

TPropAss When we assign to a property, we not only want the property to exist, but we constrain the type of the new value to be a subtype of the property type.

TCall We constrain the type of the object we obtain, to actually be a function. The argument of a top level call should be of the right type. We therefore constrain the type of the argument to be a subtype of the function argument. Then after the call, we hope to obtain a result of the right type, so we constrain it to have a subtype of the function's type domain.

In order to relate arguments and return values to the function or method, we wrap them. For arguments, this means that when we pass the argument to the function, we replace its type by α_{arg}. Any usage of the argument inside the function body will now be related to the argument type of the function. The same holds for return types. When we return from a function call, we replace the type of the return value with α_{ret}.

The path taken by the function is also recorded. We want to collect the paths of functions separately. This is due to the fact that the path inside a function call might be different, depending on the value of a variable, although it has the same type.
This allows us to record the coverage of the program.

At the top of our derivation we have the TRun rule. Besides the regular initialization, it now also extracts the set of constraints C and uses it to infer the types $\mathcal{T}$ and equivalence set mapping $\mathcal{P}$ for our program. It also returns the set of observed paths Φ. The expression Solve(C) in the TRun rule is to result in a mapping from type variables to types, similar as in An et al.[6], as well as the mapping $\mathcal{P}$.

This equivalence set mapping $\mathcal{P} : \alpha_1 \mapsto \{\alpha\}$ maps a type variable to a set of type variables that are equivalent to it. From the constraints, we can conclude that certain type variables are equivalent. We define two type variables to be equivalent if both type variables are a subtype for the same property, like the example below.

$o.x = test;$	$\alpha_o \leq [x : \alpha_o.x], \alpha_{test} \leq \alpha_o.x$
$o.x = test1;$	$\alpha_o \leq [x : \alpha_o.x], \alpha_{test1} \leq \alpha_o.x$

On the left, two lines of a program are shown. On the right, the constraints generated by this small piece of code are listed.

These constraints show that both the type variable for test, α_{test}, and test1, α_{test1}, are subtypes for the property x. Therefore we consider the two type variables equivalent. For practical purposes, if $\mathcal{P}$ is called on a non-function or non-object, we return the universal set.

Algorithm 1: Solve algorithm

Input : Set of type constraints C
Output: $T, \mathcal{P}$
Algorithm `Solve(`C`)`
  initialize $\mathcal{P}$ to all singleton sets
  $T = \{\}$
  forall the $\alpha_1 \leq \alpha_2 \in C$ **do**
    if $c \equiv \alpha_1 \leq \alpha_2$ && $\mathcal{P}(\alpha_1) \neq \mathcal{P}(\alpha_2)$ **then**
      $\mathcal{P}' = \mathcal{P} + \alpha_1 \sim \alpha_2$
      $T(\mathcal{P}'(\alpha_1)) = T(\mathcal{P}(\alpha_1)) \cup T(\mathcal{P}(\alpha_2))$
      $\mathcal{P} = \mathcal{P}'$
    if $c \equiv \alpha_1 \leq [m : \alpha_2]$ **then**
      if $[m : \_\,] \notin T(\mathcal{P}(\alpha_1))$ **then** add $[m : \mathcal{P}(\alpha_2)]$ to $T(\mathcal{P}(\alpha_1))$;
      if $[m : \alpha_3] \in T(\mathcal{P}(\alpha_1))$ **then**
        $\mathcal{P}' = \mathcal{P} + \alpha_2 \sim \alpha_3$
        $T(\mathcal{P}'(\alpha_2)) = T(\mathcal{P}(\alpha_2)) \cup T(\mathcal{P}(\alpha_3))$
        $\mathcal{P} = \mathcal{P}'$
    if $c \equiv \alpha_1 \leq (\alpha_2 \longrightarrow \alpha_3)$ **then**
      if $\longrightarrow (\_\,, \_\,) \notin T(\mathcal{P}(\alpha_1))$ **then** Add $\longrightarrow (P(\alpha_2), \mathcal{P}(\alpha_3))$ to $T(\mathcal{P}(\alpha_1))$;
      if $\longrightarrow (\alpha_2', \alpha_3') \in T(\mathcal{P}(\alpha_1))$ **then**
        $\mathcal{P}' = \mathcal{P} + \alpha_2 \sim \alpha_2' + \alpha_3 \sim \alpha_3'$
        $T(\mathcal{P}'(\alpha_2)) = T(\mathcal{P}(\alpha_2)) \cup T(\mathcal{P}(\alpha_2'))$
        $T(\mathcal{P}'(\alpha_3)) = T(\mathcal{P}(\alpha_3)) \cup T(\mathcal{P}(\alpha_3'))$
        $\mathcal{P} = \mathcal{P}'$
  end
  return $T, \mathcal{P}$

Solve is listed in pseudocode in Algorithm 1. Solve takes the set of constraints C as

an argument. It returns the equivalence mapping $\mathcal{P}$ and a partial mapping from type variables to type constructors, T. We have omitted the algorithm that converts this mapping T to the mapping from type variables to complete types $\mathcal{T}$, since this algorithm is quite complicated and we do not need the full types. For most uses, except printing types, the mapping T is sufficient. If needed though, T can be converted to $\mathcal{T}$.

The type constructors that are used in T are:

- $[m : \alpha]$, where m is a field name and α its type variable
- $\longrightarrow (\alpha_1, \alpha_2)$, where $\longrightarrow$ indicates a function type, from type α_1 to α_2

Solve steps trough all type constraints in C. If the constraint consists of just two type variables, it checks whether these two variables are already equivalent. If not, this equivalence information is added to $\mathcal{P}$.
For field constraints, the algorithm checks if the type constructor for this field is already present in T. If not, the constructor is added. If it is, then $\mathcal{P}$ is updated with the new equivalence information. Changes in $\mathcal{P}$ also affect T. Therefore, we unify the type constructor information based on the old $\mathcal{P}$ and store it in T under the new $\mathcal{P}$.
For function type constraints, we again check if the function type constructor is present in T. If not, we add it. If it is, then we update $\mathcal{P}$ with the new equivalence information. Then we also update T since it is affected by changes in $\mathcal{P}$, like in the case of field constraints.
After all constraints have been processed, T and $\mathcal{P}$ are returned.

2.3.4 Monitoring

The last semantics we define is the monitoring semantics, listed in Figures 10, 11, 12 and 13. The goal of this semantics is to provide a way of executing a program, given a type mapping $\mathcal{T} : \alpha \mapsto \tau$, equivalence mapping $\mathcal{P} : \alpha_1 \mapsto \{\alpha\}$ and set of paths Φ. Note that this implies that execution must be altered, since it is not possible to traverse paths that are not given. The mapping $\mathcal{T}$, $\mathcal{P}$ and set of paths Φ are obtained from the training run, and are implicit parameters to the monitoring run. Why do we force the monitoring semantics to stay on the observed paths Φ? This is the only part of the program that we have information about.

The reduction judgments are now of the form $H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi'$, where ϕ contains the path that the reduction has to follow, and ϕ' contains the remainder of the path after reduction.

Figure 10 contains the basic monitoring semantics rules. All rules, except for the CONDITIONAL and MCALL rule, follow the regular execution. The only difference is that they pass on the current path in the parameter ϕ. Additionally, the MNEW and MFUN now store the type variable of the object created in this rule, in the reserved field $tyvar.

$$\textsc{MVarLookup}\ \frac{S(x) = v}{H; S; x \mid \phi \longrightarrow H; S; v \mid \phi} \qquad \textsc{MPCall}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \qquad [\![pr]\!]v = v'}{H; S; pr(e) \mid \phi \longrightarrow H'; S'; v' \mid \phi'}$$

$$\textsc{MNew}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \\ l' = \text{fresh label} \qquad \alpha = \ell \\ obj = (v, \$\text{tyvar} \mapsto \alpha) \qquad H' = H\{l' \mapsto obj\} \end{array}}{H; S; \texttt{new}^{\ell}\, e \mid \phi \longrightarrow H'; S; l \mid \phi'} \qquad \textsc{MVarAss}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \qquad S'' = S'\{x \mapsto v\}}{H; S; x = e \mid \phi \longrightarrow H'; S''; v \mid \phi'}$$

$$\textsc{MFun}\ \frac{\begin{array}{c} l = \text{fresh location} \\ \alpha = \ell \qquad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \\ \$\text{vars} \mapsto S \downarrow_{fv(fundec)}, \$\text{tyvar} \mapsto \alpha)\} \end{array}}{H; S; fundec^{\ell} \mid \phi \longrightarrow H; S; o \mid \phi} \qquad \textsc{MProp}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; l.n \longrightarrow v}{H; S; e.n \mid \phi \longrightarrow H'; S'; v \mid \phi'}$$

$$\textsc{MSeq}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; \_ \mid \phi' \\ H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \end{array}}{H; S; (e; e') \mid \phi \longrightarrow H''; S''; v \mid \phi''} \qquad \textsc{MPropAss}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \\ H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \\ H''' = H''\{l \mapsto H''(l)\{n \mapsto v\}\} \end{array}}{H; S; e.n = e' \mid \phi \longrightarrow H'''; S''; v \mid \phi''}$$

$$\textsc{MConditional}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; c \mid p, \phi' \\ \text{if } (c \notin \text{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \qquad H'; S'; e_p \mid \phi' \longrightarrow H"; S''; v \mid \phi'' \end{array}}{H; S; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi \longrightarrow H''; S''; v \mid \phi''}$$

$$\textsc{MCall}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi \longrightarrow H''; S''; v \mid \phi'' \\ H''(l) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \qquad \$\text{vars} \mapsto S^f, \$\text{tyvar} \mapsto \alpha, ...) \\ S^{f'} = S^f\{f \mapsto l, x^f \mapsto v, (y \mapsto \texttt{udf})^*\} \qquad \bar{\phi} \in \Phi \qquad H'; S^{f'}; e^f \mid \bar{\phi} \longrightarrow H''; , v' \mid \_ \end{array}}{H; S; e(e') \mid \phi \longrightarrow H''; S'; v' \mid \phi'}$$

Figure 10: Basic monitoring semantics rules

$$\textsc{MPropLookup}\ \frac{H(l)[n] = v}{H; l.n \longrightarrow v} \qquad \textsc{MProtoLookup}\ \frac{n \notin H(l) \qquad H; H(l)_{proto}.n \longrightarrow v}{H; l.n \longrightarrow v}$$

Figure 11: Judgements for prototype lookup in Monitoring semantics

$$\textsc{MTLPropAss}\ \frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \\ \text{runtype}_{H''}(v) \in \mathcal{P}(\text{runtype}_{H''}(l).n) \qquad H''' = H''\{l \mapsto H''(l)\{n \mapsto v\}\} \end{array}}{H; S; \underline{e.n = e'} \mid \phi \longrightarrow H'''; S''; v \mid \phi''}$$

Figure 12: Monitoring semantics rule for top level property assignment

$$\textsc{Error}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \quad H'; S'; e' \mid \phi \longrightarrow H''; S''; v \mid \phi'' \quad H''(l) = obj \quad \$\text{fun} \notin obj}{H; S; e(e') \mid \phi \longrightarrow \lightning}$$

$$\textsc{MRun}\ \frac{(H, S) = \text{initialize}(\mathit{fundec}^*) \quad H; S; \text{monitor}(e) \mid \{\} \longrightarrow \_; \_; v \mid \{\}}{\mathcal{T}; \mathcal{P}; \Phi \vdash \mathit{fundec}^* \triangleright e \uparrow v}$$

$$\textsc{Monitor}\ \frac{\{\}; \{\}; \text{monitor}(e) \mid \{\} \lightning}{\mathcal{T}; \mathcal{P}; \Phi \vdash \mathit{fundec}^* \triangleright e \lightning}$$

Figure 13: Error, Run and Monitor rules for Monitoring semantics

This is later used by the function runtype_H.

The CONDITIONAL rule does alter the execution. Instead of just checking the condition and taking one of the two branches, it now reads out what path it has to take, and verifies that the value of the condition adheres to this path. If not, we have no information about that part of the program, and the derivation is stuck.

The MCALL rule also alters execution. In order to execute the method dispatch, it selects a path for the method from Φ. These are the paths observed by the training run. That means that if the path we are about to take, has not been observed, the derivation is stuck.

Figure 12 contains a special version of the MPROPASS rule. The expression of this rule is marked with an underline. This rule perform an extra verification at top level, to only accept values of the correct type. MTLPROPASS verifies for the expression $\underline{e.n = e'}$, that the type of the value of e' has a type that corresponds to the type in $\mathcal{T}$. In the next section, it will become more clear why this is necessary.

Lastly, Figure 13 contains the rules ERROR, MRUN and MONITOR. The ERROR rule throws an error when an expression used as a function does not yield the heap location of a function. The rules for the propagation of this error can be found in the appendix A.

The monitor(e) function used by MRUN and MONITOR replaces all property writes expressions by their underlined counterpart. $e.n = e'$ becomes $\underline{e.n = e'}$. Again, this is done in order to treat the top level differently from the rest of the program.

$$\begin{aligned}\text{runtype}_H = \{&l \mapsto H(l)[\$\text{tyvar}], \mathit{num} \mapsto \texttt{Number}, \mathit{str} \mapsto \texttt{String},\\ &\mathit{bool} \mapsto \texttt{Bool}, \texttt{null} \mapsto \texttt{Null}, \texttt{udf} \mapsto \texttt{Udf}\}\end{aligned}$$

In these rules, we use the function runtype_H to convert runtype values and location to types. Values directly result in a concrete type. For locations, a type variable is retrieved from the heap.

2.4 Proof of Soundness

We now have all the ingredients that make up our formal dynamic type inference system. We have a core language that resembles JavaScript, we have a semantics that collects constraints during run-time and infers types after execution.

In this section, we will show that the types inferred by this system are sound. Formally, we want to prove that the following soundness theorem holds:

Theorem 1 (soundness)**.** *Suppose that we have some program* $fundec^*$ *and some expression* e_1*. We evaluate* e_1 *under* $fundec^*$*, notated as such:* $fundec^* \triangleright e_1 \uparrow \mathcal{T}; \mathcal{P}; \Phi$*, with* $\mathcal{T}$ *the types and* $\mathcal{P}$ *the equivalence mapping resulting from constraint solving and* Φ *the set of traversed paths.*
Then there cannot be a different expression e_2 *such that evaluation results in* $\lightning$ *under the inferred types and traversed paths, notated as* $\mathcal{T}; \mathcal{P}; \Phi \vdash fundec^* \triangleright e_2 \lightning$*.*

In words, if the training run has inferred a set of types for a certain program, then there can be no expression that results in a not-a-function error, given those types, equivalence information and the coverage of the training run. As mentioned before, we do not care about the execution getting stuck.

Before we begin with the proof, we define the simulation relation on the training and monitoring runs. The training run is marked $_t$, the monitoring run with $_m$. We need this definition in order to relate the training run to the monitoring run.

Definition 1 (Simulation)**.** *The simulation relation on* $H_t; S_t$ *and* $H_m; S_m$ *under types* $\mathcal{T}$ *and equivalence mapping* $\mathcal{P}$*, denoted by* $H_t; S_t \sim_{\mathcal{T}} H_m; S_m$*, holds iff the following holds.*

- *For all* $x \in dom(S_t)$*,* $S_t(x) = l_t : \alpha_t \leftrightarrow S_m(x) = l_m$ *and* $runtype_{H_m}(l_m) \in \mathcal{P}(\alpha_t)$
- *For all* $l_t \in dom(H_t)$*, whenever* $H_t(l_t) = obj_t : \alpha$ *such that* $obj_t.p = v_t : \alpha'$*, we have* $\mathcal{P}(\alpha') = \mathcal{P}(\alpha.p)$*.*
- *For all* $l_m \in dom(H_m)$*, whenever* $H_m(l_m) = obj_m$ *such that* $obj_m.p = v_m$*, we have* $runtype_{H_m}(v_m) \in \mathcal{P}(runtype_{H_m}(obj_m).p)$*.*

Definition 2 (Training heap stability)**.** H_t *is training-stable under equivalence mapping* $\mathcal{P}$ *iff, for all* $l_t \in dom(H_t)$*, whenever* $H_t(l_t) = obj : \alpha$ *such that* $obj.p = v_t : \alpha'$*, we have* $\mathcal{P}(\alpha') = \mathcal{P}(\alpha.p)$*.*

Definition 3 (Monitoring heap stability)**.** H_m *is monitoring-stable under equivalence mapping* $\mathcal{P}$ *iff, for all* $l_m \in dom(H_m)$*, whenever* $H_m(l_m) = obj$ *such that* $obj.p = v_m$*, we have* $runtype_{H_m}(v_m) \in \mathcal{P}(runtype_{H_m}(obj).p)$*.*

Lemma 1 (Simulation Splitting)**.** *Suppose that* $H_t; S_t \sim_{\mathcal{T}} H_m; S_m$*. Then by definition it holds that* H_t *is training stable and* H_m *is monitoring stable.*

Lemma 2 (Every training heap is stable)**.** *For all heaps in the training run it holds that the heap is training-stable.*

Proof. Let us assume that we have a heap that is not training-stable. This means that there is at least one heap location l_0 resulting in $H_t(l_0) = obj : \alpha$ with at least one property such that $obj.p = v_t : \alpha'$, we have $\alpha' \neq \alpha.p$.

There is only one semantic rule that assigns to properties, and changes the type variables of object properties.

$$\frac{\begin{array}{c}\textsc{TPropAss} \\ H_t; S_t; e \longrightarrow H_t'; S_t'; l_t : \alpha \mid C; \phi_t; \Phi \qquad H_t'; S_t'; e' \longrightarrow H_t''; S_t''; v_t : \bar{\tau} \mid C'; \phi_t'; \Phi' \\ C'' = \alpha \leq [n : \alpha.n], \bar{\tau} \leq \alpha_n \qquad H_t''' = H_t''\{l_t \mapsto H_t''(l_t)\{n \mapsto v_t : \alpha.n\}\}\end{array}}{H_t; S_t; e.n = e' \longrightarrow H_t'''; S_t''; v_t : \alpha.n \mid C, C', C''; \phi_t, \phi_t'; \Phi, \Phi'}$$

From this rule, it is trivial to see that it cannot be the case that $\alpha' \neq \alpha.p$.

Therefore there cannot exist an object with a property such that $H_t(l_0) = obj : \alpha$ with at least one property such that $obj.p = v_t : \alpha'$, and thus every heap that is part of the training run must be training-stable. □

Lemma 3 (Simulation from stability)**.** *Suppose that H_t is training stable, that H_m is monitoring stable, and $S_t \sim_{\mathcal{T}} S_m$ under H_M. Then $H_t; S_t \sim_{\mathcal{T}} H_m; S_m$.*

Lemma 3 follows directly from Definitions 1,2 and 3.

With the simulation definition, we can formulate a preservation lemma.

Lemma 4 (Preservation)**.** *Suppose that we have a program execution $fundec^* \triangleright e_0 \uparrow \mathcal{T}, \mathcal{P}, \Phi$, and somewhere inside the execution we derive $H_t; S_t; e_1 \longrightarrow H_t'; S_t'; \_ : \bar{\tau}_t \mid \_ ; \phi_t$. Let H_m and S_m be such that $H_t; S_t \sim_{\mathcal{T}} H_m; S_m$ and $\phi_t = \phi_m$.*
Then if $H_m; S_m; e_1 | \phi_m \longrightarrow R$:

- *We have $R = H_m'; S_m'; v_m | \phi_m'$, $runtype_{H_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$ and $H_t'; S_t' \sim_{\mathcal{T}} H_m'; S_m'$*

In words, Lemma 4 requires that *if* there is a derivation, it cannot result in a $\lightning$ and the runtype of the resulting value must be a subtype of the type inferred in the training run.

Proof. We prove this by induction on the monitoring semantics of $H_m; S_m; e_1 | \phi_m \longrightarrow R$ as follows.

Case $e_1 \equiv x$. The following rules apply:

$$\frac{S_t(x) = v_t : \bar{\tau}_t}{H_t; S_t; x \longrightarrow H_t; S_t; v_t : \bar{\tau}_t \mid \{\}; \{\}; \{\}}\ \textsc{TVarLookup} \qquad \frac{S_m(x) = v_m}{H_m; S_m; x \mid \phi_m \longrightarrow H_m; S_m; v_m \mid_m \phi}\ \textsc{MVarLookup}$$

Since we assumed $H_t; S_t \sim_{\mathcal{T}} H_m; S_m$, it follows that $\text{runtype}_{H_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$. We automatically obtain $H'_t; S'_t \sim_{\mathcal{T}} H'_m; S'_m$, since $H' = H$ and $S' = S$. Furthermore, a variable lookup cannot result in $\lightning$.

Case $e_1 \equiv pr(e)$. The following rules apply:

$$\textsc{TPCall}\ \frac{H_t; S_t; e \longrightarrow H'_t; S'_t; v_t : \bar{\tau}_t \mid C; \phi_t; \Phi \qquad [\![pr]\!] v_t = v'_t : \tau'_t}{H_t; S_t; pr(e) \longrightarrow H'_t; S'_t; v'_t : \tau'_t \mid C; \phi_t; \Phi}$$

$$\textsc{MPCall}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; v_m \mid \phi'_m \qquad [\![pr]\!] v_m = v'_m}{H_m; S_m; pr(e) \mid \phi_m \longrightarrow H'_m; S'_m; v'_m \mid \phi'_m}$$

By induction we have that if $H_m; S_m; e \mid \phi \longrightarrow R$, then $R = H'_m; S'_m; v_m \mid \phi'$. Therefore, we also have that if $H_m; S_m; pr(e) \mid \phi \longrightarrow R$, then $R = H'_m; S'_m; v'_m \mid \phi'$.
From the IH, we also get $H'_t; S'_t \sim_{\mathcal{T}} H'_m; S'_m$ and $\text{runtype}_{H'_m}(v_m) \in \mathcal{P}(\tau_t)$, thus we also have $\text{runtype}_{H'_m}(v'_m) \in \mathcal{T}(\tau'_t)$.

Case $e_1 \equiv \texttt{new}^{\ell}\ e$. The following rules apply:

$$\textsc{TNew}\ \frac{\begin{array}{c} H_t; S_t; e \longrightarrow H'_t; S'_t; v_t : \bar{\tau}_t \mid C; \phi_t; \Phi \\ l_t = \text{fresh location} \qquad \alpha'_t = \ell \\ \text{if } (\bar{\tau} = \alpha_t) \text{ then } (C' = \alpha'_t \leq \alpha_t) \text{ else } (C' = \{\}) \\ obj_t = (v_t, \{\}) \\ H''_t = H'_t\{l_t \mapsto obj_t : \alpha'_t\} \end{array}}{H_t; S_t; \texttt{new}^{\ell} e \longrightarrow H''_t; S'_t; l_t : \alpha'_t \mid C, C'; \phi_t; \Phi}$$

$$\textsc{MNew}\ \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; v_m \mid \phi'_m \\ l_m = \text{fresh label} \qquad \alpha_m = \ell \\ obj_m = (v_m, \{\text{tyvar} \mapsto \alpha_m\}) \\ H'_m = H_m\{l_m \mapsto obj_m\} \end{array}}{H_m; S_m; \texttt{new}^{\ell} e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m}$$

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H'_m; S'_m; v_m \mid \phi'_m$. Therefore, we also have that if $H_m; S_m; \texttt{new}^{\ell} e \mid \phi_m \longrightarrow R$, then $R = H'_m; S'_m; l_m \mid \phi'_m$.

By IH we also obtain $H'_t; S'_t \sim_{\mathcal{T}} H'_m; S'_m$. Since we know that both statements have the same label attached to them, we know that $\text{runtype}_{H'_m}(l_m) \in \mathcal{P}(\alpha'_t)$. These two facts give us $H''_t; S'_t \sim_{\mathcal{T}} H''_m; S'_m$.

Case $e_1 \equiv x = e$. The following rules apply:

$$\textsc{TVarAss}\ \frac{H_t; S_t; e \longrightarrow H'_t; S'_t; v_t : \bar{\tau}_t \mid C; \phi_t; \Phi \qquad S''_t = S'_t\{x \mapsto v_t : \bar{\tau}_t\}}{H_t; S_t; x = e \longrightarrow H'_t; S''_t; v_t : \bar{\tau}_t \mid C; \phi_t; \Phi}$$

$$\textsc{MVarAss}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; v_m \mid \phi'_m \qquad S''_m = S'_m\{x \mapsto v_m\}}{H_m; S_m; x = e \mid \phi_m \longrightarrow H'_m; S''_m; v_m \mid \phi'_m}$$

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H'_m; S'_m; v_m \mid \phi'_m$. Therefore, we also have that if $H_m; S_m; x = e \mid \phi_m \longrightarrow R$, then $R = H'_m; S'_m; l_m \mid \phi'_m$, since an activation record update can never result in $\lightning$.

We also get from IH that $runtype_{H'_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$ and $H'_t; S'_t \sim_{\mathcal{T}} H'_m; S'_m$. Since $S''_t \sim_{\mathcal{T}} S''_m$ due to the fact that $\text{runtype}_{H'_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$, we also get $H'_t; S''_t \sim_{\mathcal{T}} H'_m; S''_m$.

Case $e_1 \equiv \text{fun}^{\ell}\ f(x)\{\text{var } y^*, \text{return } e\}$. The following rules apply:

$$\textsc{TFun}\ \frac{\begin{array}{c} l_t = \text{fresh location} \qquad \alpha_t = \ell \\ H_t' = H_t\{l_t \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \\ \$\text{vars} \mapsto S\downarrow_{fv(fundec)})\} \end{array}}{H_t; S_t; fundec^{\ell} \longrightarrow H_t'; S_t; l_t : \alpha_t \mid \{\}; \{\}; \{\}}$$

$$\textsc{MFun}\ \frac{\begin{array}{c} l_m = \text{fresh location} \qquad \alpha_m = \ell \\ H_m' = H_m\{l_m \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \\ \$\text{vars} \mapsto S\downarrow_{fv(fundec)}, \$\text{tyvar} \mapsto \alpha_m)\} \end{array}}{H_m; S_m; fundec^{\ell} \mid \phi_m \longrightarrow H_m'; S_m; l_m \mid \phi_m}$$

By definition, we have $\text{runtype}_{H_m}(l_m) \in \mathcal{P}(\alpha_t)$
We automatically obtain $H_t'; S_t' \sim_{\mathcal{T}} H_m'; S_m'$, since $H' = H$ and $S' = S$.

Case $e_1 \equiv e.n$. The following rules apply:

$$\textsc{TProp}\ \frac{\begin{array}{c} H_t; S_t; e \longrightarrow H_t'; S_t'; l_t : \alpha_t \mid C; \phi_t; \Phi \\ C' = \alpha_t \leq [n : \alpha] \qquad H_t'; l_t.n \longrightarrow v_t : \bar{\tau}_t \end{array}}{H_t; S_t; e.n \longrightarrow H_t'; S_t'; v_t : \bar{\tau}_t \mid C, C'; \phi_t; \Phi}$$

$$\textsc{MProp}\ \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H_m'; S_m'; l_m \mid \phi_m' \\ H_m'; l_m.n \longrightarrow v_m \end{array}}{H_m; S_m; e.n \mid \phi_m \longrightarrow H_m'; S_m'; v_m \mid \phi_m'}$$

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; v_m \mid \phi_m'$. Therefore, we also have that if $H_m; S_m; e.n \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; l_m \mid \phi_m'$, since a property look-up can never result in $\lightning$.

We also get from IH that $runtype_{H_m'}(l_m) \in \mathcal{P}(\alpha_t)$ and $H_t'; S_t' \sim_{\mathcal{T}} H_m'; S_m'$.
Therefore we know that $runtype_{H_m'}(v_m) \in \mathcal{P}(\bar{\tau}_t)$.

Case $e_1 \equiv e; e'$. The following rules apply:

$$\textsc{TSeq}\ \frac{H_t; S_t; e \longrightarrow H_t'; S_t'; \_ \mid C; \phi_t; \Phi_t \qquad H_t'; S_t'; e' \longrightarrow H_t''; S_t''; v_t : \bar{\tau}_t \mid C'; \phi_t'; \Phi_t'}{H_t; S_t; (e; e') \longrightarrow H_t''; S_t''; v_t : \bar{\tau}_t \mid C, C'; \phi_t, \phi_t'; \Phi_t, \Phi_t'}$$

$$\textsc{MSeq}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H_m'; S_m'; \_ \mid \phi_m' \qquad H_m'; S_m'; e' \mid \phi_m' \longrightarrow H_m''; S_m''; v_m \mid \phi_m''}{H_m; S_m; (e; e') \mid \phi_m \longrightarrow H_m''; S_m''; v_m \mid \phi_m''}$$

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; v_m \mid \phi_m'$. Applying the IH a second time gives us that if $H_m'; S_m'; e' \mid \phi_m' \longrightarrow R$, then $R = H_m''; S_m''; v_m' \mid \phi_m''$. Therefore, we also have that if $H_m; S_m; (e; e') \mid \phi_m \longrightarrow R$, then $R = H_m''; S_m''; v_m \mid \phi_m''$.
These induction steps also give us $runtype_{H_m''}(v_m) \in \mathcal{P}(\bar{\tau}_t)$ and $H_t''; S_t'' \sim_{\mathcal{T}} H_m''; S_m''$.

Case $e_1 \equiv e.n = e'$. The following rules apply:

$$\textsc{TPropAss}\quad \frac{\begin{array}{c} H_t; S_t; e \longrightarrow H_t'; S_t'; l_t : \alpha_t \mid C; \phi_t; \Phi_t \qquad H_t'; S_t'; e' \longrightarrow H_t''; S_t''; v_t : \bar{\tau}_t \mid C'; \phi_t'; \Phi_t' \\ C_t'' = \alpha_t \leq [n : \alpha_{tn}], \bar{\tau}_t \leq \alpha_{tn} \qquad H_t''' = H_t''\{l_t \mapsto H_t''(l_t)\{n \mapsto v_t : \bar{\tau}_t\}\} \end{array}}{H_t; S_t; e.n = e' \longrightarrow H_t'''; S_t''; v_t : \bar{\tau}_t \mid C, C', C''; \phi, \phi'; \Phi_t, \Phi_t'}$$

$$\textsc{MPropAss}\quad \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H_m'; S_m'; l_m \mid \phi_m' \\ H_m'; S_m'; e' \mid \phi_m' \longrightarrow H_m''; S_m''; v_m \mid \phi_m'' \qquad H_m''' = H_m''\{l_m \mapsto H_m''(l_m)\{n \mapsto v_m\}\} \end{array}}{H_m; S_m; e.n = e' \mid \phi_m \longrightarrow H_m'''; S_m''; v_m \mid \phi_m''}$$

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; l_m \mid \phi_m'$. Applying the IH a second time gives us that if $H_m'; S_m'; e' \mid \phi_m' \longrightarrow R$, then $R = H_m''; S_m''; v_m' \mid \phi_m''$. Therefore, we also have that if $H_m; S_m; e.n = e' \mid \phi_m \longrightarrow R$, then $R = H_m''; S_m''; v_m \mid \phi_m''$ since a heap update cannot result in $\lightning$.

From the same induction steps we obtain that $\text{runtype}_{H_m''}(v_m) \in \mathcal{P}(\bar{\tau}_t)$ and $H_t''; S_t'' \sim_{\mathcal{T}} H_m''; S_m''$. Since we only change property n of the object located at l_t and we know that $\text{runtype}_{H_m''}(v_m) \in \mathcal{P}(\bar{\tau}_t)$, it follows that $H_t''' \sim_{\mathcal{T}} H_m'''$ and thus $H_t'''; S_t'' \sim_{\mathcal{T}} H_m'''; S_m''$

Case $e_1 \equiv \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''$. The following rules apply:

$$\textsc{TConditional}\quad \frac{\begin{array}{c} H_t; S_t; e \longrightarrow H_t'; S_t'; c_t : \tau_t \mid C; \phi_t; \Phi \\ \text{if } (c_t \notin \text{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \\ H_t'; S_t'; e_p \longrightarrow H_t"; S_t''; v_t : \bar{\tau}_t \mid C'; \phi_t'; \Phi' \end{array}}{H_t; S_t; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \longrightarrow H_t''; S_t''; v_t : \bar{\tau}_t \mid \{C, C'\}; \phi_t, p, \phi_t'; \Phi, \Phi'}$$

$$\textsc{MConditional}\quad \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H_m'; S_m'; c_m \mid p, \phi_m' \\ \text{if } (c_m \notin \text{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \\ H_m'; S_m'; e_p \mid \phi_m' \longrightarrow H_m"; S_m''; v_m \mid \phi_m'' \end{array}}{H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \longrightarrow H_m''; S_m''; v_m \mid \phi_m''}$$

The proof of this case is symmetrical for the two possible paths.

By induction we have that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; c_m \mid p, \phi_m'$.

Now there are two possibilities. If the monitoring semantics deviates from the path that the training semantics is on, we don't have $H_m'; S_m'; e_p \mid \phi_m \longrightarrow R$ and therefore also not $H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \longrightarrow R$. If we are on the same path, then we can apply the IH a second time, which gives us that if $H_m'; S_m'; e' \mid \phi_m' \longrightarrow R$, then $R = H_m''; S_m''; v_m' \mid \phi_m''$. Therefore, we also have that if $H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \longrightarrow R$, then $R = H_m''; S_m''; v_m \mid \phi_m''$.

Assuming that we do have $H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \longrightarrow R$, we know that it must hold that $\text{runtype}_{H_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$, since our semantics forces both executions to take the same path at this point and we know that we have $H_t''; S_t'' \sim_{\mathcal{T}} H_m''; S_m''$ from the second application of the IH.

Case $e_1 \equiv e(e')$. The following rules apply:

$$\textsc{TCall}\quad \frac{\begin{array}{c} H_t; S_t; e \longrightarrow H_t'; S_t'; l_t : \alpha_t \mid C; \phi_t; \Phi_t \qquad H_t'; S_t'; e' \longrightarrow H_t''; S_t''; v_t : \bar{\tau}_t \mid C; \phi_t'; \Phi_t' \\ H''(l_t) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f_t(x_t^f)\{(\texttt{var}\ y_t)^*, \texttt{return}\ e_t^f\}, \$\text{vars} \mapsto S_t^f, ...) \\ S_t^{f'} = S_t^f\{f \mapsto l_t : \alpha_t, x^f \mapsto v : \alpha_{1arg}, (y \mapsto \texttt{udf} : \texttt{Udf})^*\} \\ C^{call} = \alpha_t \leq \alpha_{targ} \longrightarrow \alpha_{tret}, \bar{\tau}_t \leq \alpha_{targ}, \\ H_t''; S_t^{f'}; e^f \longrightarrow H'''; \_; v_t : \bar{\tau}_t' \mid C'; \phi_t''; \Phi_t^f \qquad C^{ret} = \bar{\tau}_t' \leq \alpha_{tret} \end{array}}{H_t; S_t; e(e') \longrightarrow H_t'''; S_t''; v_t' : \alpha_{tret} \mid \{C, C^{call}, C^{ret}, C'\}; \phi_t, \phi_t'; \Phi_t, \Phi_t', \phi_t'', \Phi_t^f}$$

$$\textsc{MCall}\quad \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H_m'; S_m'; l_m \mid \phi_m' \qquad H_m'; S_m'; e' \mid \phi_m' \longrightarrow H_m''; S_m''; v_m \mid \phi_m'' \\ H_m''(l_m) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f_m(x_m^f)\{(\texttt{var}\ y_m)^*, \texttt{return}\ e_m^f\}, \$\text{vars} \mapsto S_m^f, \$\text{tyvar} \mapsto \alpha_m, ...) \\ S_m^{f'} = S_m^f\{f \mapsto l_m, x^f \mapsto v_m, (y \mapsto \texttt{udf})^*\} \qquad \bar{\phi} \in \Phi \qquad H_m''; S_m^{f'}; e^f \mid \bar{\phi} \longrightarrow H_m'''; \_; v_m' \mid \_ \end{array}}{H_m; S_m; e(e') \mid \phi_m \longrightarrow H_m'''; S_m''; v_m' \mid \phi_m''}$$

By applying the induction hypothesis, we get that if $H_m; S_m; e \mid \phi_m \longrightarrow R$, then $R = H_m'; S_m'; l_m \mid \phi_m'$, $H_t'; S_t' \sim_{\mathcal{T}} H_m'; S_t'$ and $\text{runtype}_{H_m''}(l_m) \in \mathcal{P}(\alpha_t)$. By applying it a second time, we get that if $H_m'; S_m'; e' \mid \phi_m' \longrightarrow R$, then $R = H_m''; S_m''; v_m \mid \phi_m''$ and $\text{runtype}_{H_m''}(v_m) \in \mathcal{P}(\bar{\tau}_t)$. We also have $H_t''; S_t'' \sim_{\mathcal{T}} H_m''; S_t''$.

Since we have $\text{runtype}_{H_m''}(l_m) \in \mathcal{P}(\alpha_t)$, we know that at some point, the training semantics has taken the path that the monitoring run is about to take. Therefore, we know that there must be a derivation in the training semantics for $H_0''; S_m^f\{f \mapsto l_0 : \alpha_0, x^f \mapsto v_0 : \alpha_{arg}, (y \mapsto \texttt{udf} : \texttt{Udf})^*\}; e_m^f \longrightarrow H_0'''; \_; v_0' : \bar{\tau}_t' \mid C'; \phi_0'; \Phi_0^f$.

$$\textsc{TCall}\quad \frac{\begin{array}{c} H_0; S_0; e \longrightarrow H_0'; S_0'; l_0 : \alpha_0 \mid C; \phi_0; \Phi_0 \qquad H_0'; S_0'; e' \longrightarrow H_0''; S_0''; v_0 : \bar{\tau}_0 \mid C; \phi_0'; \Phi_0' \\ H''(l_0) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f_m(x_m^f)\{(\texttt{var}\ y_m)^*, \texttt{return}\ e_m^f\}, \$\text{vars} \mapsto S_m^f, ...) \\ S_0^{f'} = S_t^f\{f \mapsto l_0 : \alpha_0, x^f \mapsto v_0 : \alpha_{1arg}, (y \mapsto \texttt{udf} : \texttt{Udf})^*\} \\ C^{call} = \alpha_0 \leq \alpha_{0arg} \longrightarrow \alpha_{0ret}, \bar{\tau}_t \leq \alpha_{0arg}, \\ H_0''; S_0^{f'}; e_m^f \longrightarrow H_0'''; \_; v_0' : \bar{\tau}_0' \mid C'; \phi_0''; \Phi_0^f \qquad C^{ret} = \bar{\tau}_0' \leq \alpha_{0ret} \end{array}}{H_0; S_0; e(e') \longrightarrow H_0'''; S_0''; v_0' : \alpha_{0ret} \mid \{C, C^{call}, C^{ret}, C'\}; \phi_0, \phi_0'; \Phi_0, \Phi_0', \phi_0'', \Phi_0^f}$$

Since we have $H_0''; S_0'' \sim_{\mathcal{T}} H_m''; S_m''$, and we know that there must be a derivation for $H_0''; S_m^f\{f \mapsto l_0 : \alpha_0, x^f \mapsto v_0 : \alpha_{arg}, (y \mapsto \texttt{udf} : \texttt{Udf})^*\}; e_m^f$ in the training semantics, we can apply the induction hypothesis on the monitoring semantics. We get that if $H_0''; S_0^{f'}; e_m^f \longrightarrow R$, then $R = H_m'''; \_; v_m' \mid \_$ and $H_0'''; \_ \sim_{\mathcal{T}} H_m'''; \_$.

By lemma 1, we have that H'''_m is monitoring stable. From lemma 2 we have that H'''_t is training stable.

We already have that $H''_t; S''_t \sim_{\mathcal{T}} H''_m; S''_t$. We know that $S''_t \sim_{\mathcal{T}} S''_m$ must also hold under H'''_m since the changes to H''_m that result in H'''_m are caused by a path that is observed in the training run. This, together with monitoring and training stability give us that $H'''_t; S''_t \sim_{\mathcal{T}} H'''_m; S''_t$. □

This preservation proof only holds within the execution, and explicitly not for top level calls. For the top level, we need to do some extra work. We want to prove that the following holds.

Lemma 5 (Top Level Preservation)**.** *Suppose that we have a program execution* $fundec^{*\triangleright} e_0 \uparrow \mathcal{T}, \mathcal{P}, \Phi$. *Let* H_m *be such that monitoring heap simulation holds.*
Then if $H_m; S_m; monitor(e_1)|\phi \longrightarrow R$*:*

- *We have* $R = H'_m; S'_m; v_m|\phi'$
- H'_m *is monitoring stable.*

Proof. By induction on expression:

Case monitor$(e_1) \equiv x \mid \text{fun} f(x)\{\text{var } y^*, \text{return } e\} \mid pr(e) \mid \text{new } e \mid x = e \mid e.n \mid e; e \mid$ if e then e' else $e'' \mid e(e')$
In these cases, one can immediately apply the induction hypothesis.

Case monitor$(e_t) \equiv e.n = e'$

$$\frac{\begin{array}{l}\textsc{MTLPropAss} \\ H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \qquad H'_m; S'_m; e' \mid \phi'_m \longrightarrow H''_m; S''_m; v_m \mid \phi''_m \\ \text{runtype}_{H''_m}(v_m) \in \mathcal{P}(\text{runtype}_{H''_m}(l_m).n) \qquad H'''_m = H''_m\{l \mapsto H''(l_m)\{n \mapsto v_m\}\}\end{array}}{H_m; S_m; \underline{e.n = e'} \mid \phi_m \longrightarrow H'''_m; S''_m; v_m \mid \phi''_m}$$

We can apply induction twice here to obtain if $H_m; S_m; e \mid \phi \longrightarrow R$, we have $R = H'_m; S'_m; l_m \mid \phi'$ and H'_m is monitoring stable, and if $H'_m; S'_m; e' \mid \phi' \longrightarrow R$, we have $R = H''_m; S''_m; v_m \mid \phi''$ and H''_m is monitoring stable.

We now need to show that H'''_m is monitoring stable. This heap has one updated field. For this field, it must hold that $\text{runtype}_{H''_m}(v_m) \in \mathcal{P}(\text{runtype}_{H''_m}(l_m).n)$. Since MTLPropAss requires this to be true in order for the rule to apply, we know that this must hold.
□

From lemma 4 and 5, we can conclude that theorem 1 holds for our formal system.

3 Practical Implementation

In this chapter, we will describe our practical implementation of a dynamic type inference system for JavaScript. This implementation is based on the same ideas as the formal system. We will see the the advantages of doing a dynamic analysis. It is simple and can be developed quickly.

3.1 Approach

During execution, we want to observe what types occur. There are three different kind of items that we observe types for. Objects, functions and frames. These three kinds all have properties that have types.
As in the formal system, we generate constraints for each observation we do. Additionally, the programmer can annotate his code with types. These annotations are converted into trusted constraints. After execution, the collected constraints are processed. If the annotated types conflict with the inferred types, we issue a type error. If we detect multiple types for a single property, we warn the programmer that there is an inconsistency. Finally, we also report the types that we inferred for the program.

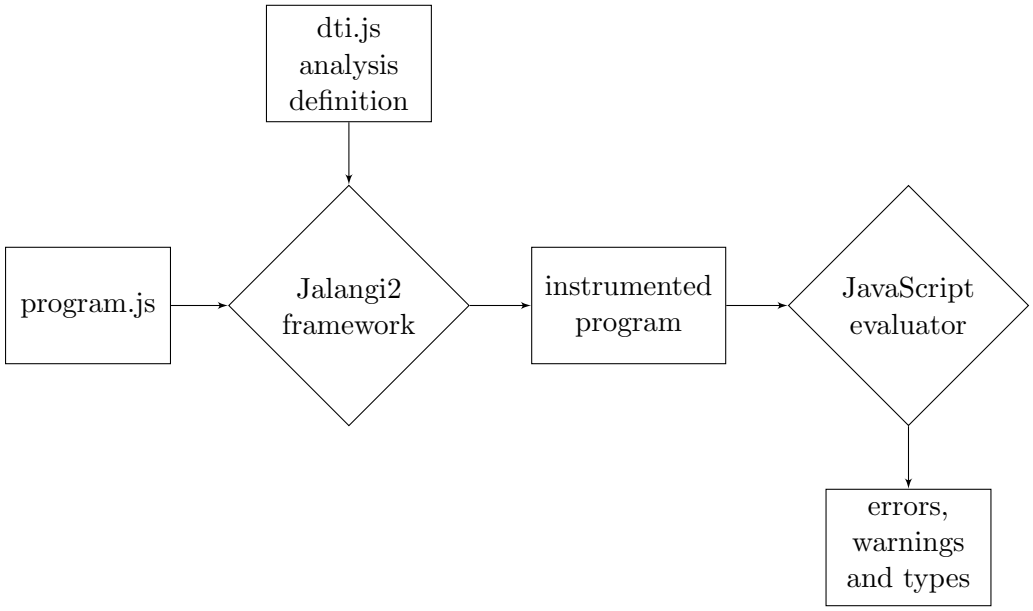

Figure 14: Analysis pipeline diagram

3.2 Relation to Formal System

Before we dive into the implementation itself, we want to make clear how this practical implementation relates to the formal system. As mentioned, the implementation is based on the same principles as the formal system. During execution of the program

we collect constraints from observations we do. Our practical implementation works on full JavaScript, and this is where it deviates from the formal system. This also means that the properties that hold for our formal system do not immediately hold for the implementation. The goal of our formal system is to show that our idea makes sense, which we have done by proving soundness. Now, we want to provide an implementation for full JavaScript based on this idea that we have proven to make sense.

3.3 Implementation

We have implemented our system on top of Jalangi2, a dynamic analysis framework for JavaScript [18, 2]. Our approach is similar to Pradel et al. [16]. The difference between their solution and ours is that we allow the user to supply type annotations. This allows us to provide the user with errors when his annotated types are violated. Furthermore, we report the inferred types, whereas Pradel et al. only report on inconsistencies. Our types are also a bit more precise, since we use singleton types.

As mentioned earlier, doing dynamic type inference has the advantage of being easy to implement. The Jalangi2 framework makes our life even easier. It provides api's for intercepting function calls, property read and writes and literals (to extract annotations). Our implementation consists of a single JavaScript file, with roughly 750 lines of code.

The constraints we collect are of the form "(base, property, type)" where base is the object, frame or function that has the property. In case of a frame, this is a local variable and in case of a function, the properties are the arguments and return. The type is the type of the property. Types are defined as the tuple "(type, value, [location])", where type is a primitive JavaScript type (number, boolean, string, undefined, object, function), value a primitive value, and [location] is a list of locations where the type was observed.
An example of a constraint can be: (frame global, day, (string, "monday", [line 1]))

3.3.1 Type Annotations

It might seem trivial at first, but providing programmers with a solid system for type annotations can be a bit tricky.
Figure 15 lists the syntax for our types. The syntax is pretty straight forward. For functions, the programmer uses the keyword `function`, followed by the name of the function, followed by a colon, followed by the type in Haskell style. For frames, the programmer uses the keyword `frame`, followed by a colon, followed by the object type. Listing 1 shows an example program with type annotations.
Annotations can be placed anywhere in the source code of the program as literal strings. The frame that they are positioned in, implies what frame or function they belong to. Looking again at Listing 1, the annotation on line 1 belongs to the function foo,

```
1  "function foo:{number->number->number->Array}"
2   function foo(x,y,z){
3
4    "function bar:{number->number->Array}"
5     function bar(x,y){return Array(x,y);}
6
7     "frame:[result:Array]"
8     var result = bar(x,bar(y,z));
9
10    return result;
11 }
12 "function foobar:{[x1:Array,x2:Array]->[xyz:Array->Array->Array]->Array}"
13  function foobar(obj,fun){...}
```

Listing 1: Example of type annotations

$$
\begin{aligned}
\mathit{annotation} &::= \texttt{function}\ \mathit{id} : \{\tau_{fun}\} \mid \texttt{frame} : [\tau_{obj}] \\
\mathit{id}, \mathit{prop} &\in \text{set of names} \\
\tau_{fun} &::= \mathit{type} \mid \ \mathit{type}\ \texttt{->}\ \tau_{fun} \mid \mathit{type} \nmid \mathit{type} \\
\mathit{type} &::= [\tau_{obj}] \mid \{\tau_{fun}\} \mid \texttt{number} \mid \texttt{boolean} \mid \texttt{string} \mid \texttt{Array} \mid \texttt{self} \mid \texttt{undefined} \mid \texttt{null} \\
\tau_{obj} &::= \mathit{prop} : \mathit{type} \mid \mathit{prop} : \mathit{type}\ , \tau_{obj}
\end{aligned}
$$

Figure 15: Syntax for type annotation

contained in the global frame. The annotation for function bar belongs to the function bar contained in the frame of foo.

3.3.2 Instrumentation

As mentioned before, we want to collect types for objects. In order to properly do this, we need to annotate every object we come across during execution with a shadow value. This unique value added as a property of the object. In order to prevent this value from influencing the execution of the origional program, we set its enumerable and writable properties to false. This prevents it from being enumerated by an iterator such as "for ... in ...".
In order to collect constrains for frames, we need to keep track of what frame we are currently in. We do this by simply defining a global variable to our analysis, that holds an array of frame names. The last entry is our current frame. When we enter a function, we use the Jalangi2 API "functionEnter" to push the name of the function we are about to enter in this array. When leaving a function, the "functionExit" API allows us to intercept and pop the current frame, and thus returning to one frame above.

These two shadow accounts are then used when generating constraints. Constraints

are generated by the following six different Jalangi2 API calls.

invokeFun is called when a function is invoked. For each argument passed to the function, we generate a constraint of the form (function name, arg$n$, type), where function name is the name of the function invoked, n the index of the argument, and type the type of the argument. We also generate a constraint for the return value. Additionally, we also check if the function being called is used as a constructor. If so, we also constrain the new object.

getField is called whenever a property is being read. We go up in the prototype chain, to find who is actually supplying the field. Then we constrain the supplier to actually have that property, and the type that we found for the value the property has.

putField is called whenever a property is begin set. Since we ware setting a property, we don't have to to a prototype lookup.

read is called whenever we read a local variable. This variable belongs to the current frame, so we look up what frame we are in, using are frame array, and constrain it to have the variable we try to read as a property, and the type we get from reading.

write is called when we write to a local property. Same as with read, we constrain the current frame to have the property we write to, and have the type of the value we try to write.

literal is called whenever we run into a literal. When we detect that the literal is a string, and starts with "function", we assume it is a type annotation, and collect it as a special constraint.

Although it might not be completely obvious at first, we actually collect constraints in the same way as the formal system. The constraints collected in the TCall rule correspond to those collected in the invokeFun API call. TProp corresponds to getField and TPropAss corresponds to putField. We extend upon the formal system by also collecting information about local variables, inferring types for frames, and of course by collecting type annotations.

It will be obvious to the reader that this approach generates an enormous amount of constraints, most of which will be duplicates. This results in memory problems when performing the analysis. To overcome this problem, we have constructed a lattice-like data structure in the form of a sparse array. This allows us to efficiently check if a constraint is duplicate. It it is, we just carry on, if it is not, we add it do our data structure. Additionally, we only keep one value around for a type. If we see the same type but a different value again for the same property, we just mark it as $\top$. This is similar to singleton types as described by Thiemann [19].

3.3.3 Processing

When the instrumented program code is executed and terminates, a final API call is made to "endExecution". Here, we preform post processing on the constraints we have collected. The first step is condensing the type observations. Although we know that we don't have duplicate constraints, we likely have duplicate object types. With duplicate object types, we mean two different type variables, both pointing to two object types that are equal. We only need to keep one of those duplicate object types, since we can just update the type variables to point to the one we keep. This heavily condenses the set of observations and makes it easier to generate the warnings, errors and inferred types.

After condensing, we check for errors. The only way to get type errors is if we observe types that are different from the annotation supplied by the programmer. This process is fairly straight-forward. For each annotation, we check if it is in the set of observed types and equal to the type of the observation. If it's not, we return an error.

After checking for errors, we look for inconsistencies. But before we can do that, we need to establish what an inconsistent type actually is. For now, we will just assume that a property has an inconsistent type when it has more then one type. The condensing of the type observations has made this step easier for us, since we don't have any redundant type variables. Therefore, if a property has more then one type, we know that these are truly different. Note that they could still be subtypes. For every property that has more than one type, we generate an inconsistency warning.

3.3.4 Reporting

There are three pieces of information to report to the programmer. Type errors, type inconsistencies (warnings) and the inferred types. The first and the last one are very straight forward. We just print the errors, combined with line numbers where the error came from. The inferred types can just be printed as they are, with some nice formatting. The only problematic information to report, is the type inconsistencies. During development and testing, we found out that you get a lot of false warnings with the aforementioned definition of type inconsistency. High polymorphic code can result in hundreds of type inconsistency warnings for medium sized programs. We have to come up with a better definition of type inconsistency than the one given in the previous section.

As shown by Pradel et al. [16], coming up with a good definition is not as straightforward as it seems. In their paper, the authors start out with the same definition mentioned earlier, a property has an inconsistent type when it has more than one type. Then a number of metrics is used to prune warnings that are almost certainly false positives.
The metrics suggested require complex systems and fine tuning to prune just enough so

that the analysis is still useful. Since type inconsistency detection is beyond the scope of this thesis, we will not implement nor extend on their complete set of metrics. We have however implemented three very basic metrics, taken from the aforementioned work.

Null-related warning pruning The value `null`, unlike `undefined`, does not occur in JavaScipt, unless the programmer explicitly assigns it. Therefore, the type `Null` only occurs intentionally. Therefore we can prune `null`-related warnings.

Degree of inconsistency pruning Polymorphic code generates a lot of inconsistency warnings. These are most likely false positives. We therefore define a maximum number of types that we consider to be inconsistent. If we for example set this number to 2, we only report properties with exactly two types in our inconsistency warnings.

Max difference pruning Another symptom of polymorphic code is that widely different objects may be passed to a function, that, for example, performs some generic operation on it. Therefore we allow the programmer to set the maximum difference between the types. If they have more differences than this number, the warning is pruned and considered to be a false positive due to intentionally polymorphic code.

All these metrics can be set by the user of the analysis if and as desired. We do not investigate different settings for these methods, this is beyond the scope of this thesis.

3.4 Complete example

In this section, we will demonstrate how the implementation works, and how a programmer can use it by showing and discussing a complete example. Listing 2 shows the source code for the program "access-nsieve" from the SunSpider benchmark. The program calculates three large prime numbers.
The program has been augmented with three type annotations, on lines 5, 14 and 30. These annotations are very straight forward. We came up with these types by just inspecting the source code by hand.

```
1  // The Great Computer Language Shootout
2  // http://shootout.alioth.debian.org/
3  //
4  // modified by Isaac Gouy
5  "function pad:{number->number->string}"
6  function pad(number,width){
7     var s = number.toString();
8     var prefixWidth = width - s.length;
9     if (prefixWidth>0){
10       for (var i=1; i<=prefixWidth; i++) s = " " + s;
11    }
12    return s;
13 }
```

```
14 "function nsieve:{number->Array->number}"
15 function nsieve(m, isPrime){
16   var i, k, count;
17
18    for (i=2; i<=m; i++) { isPrime[i] = true; }
19    count = 0;
20
21    for (i=2; i<=m; i++){
22       if (isPrime[i]) {
23          for (k=i+i; k<=m; k+=i) isPrime[k] = false;
24          count++;
25       }
26    }
27     console.log(count);
28    return count;
29 }
30 "function sieve:{undefined}"
31 function sieve() {
32     for (var i = 1; i <= 3; i++ ) {
33         var m = (1<<i)*10000;
34         var flags = Array(m+1);
35         nsieve(m, flags);
36     }
37 }
38
39 sieve();
```

Listing 2: access-nsieve.js from SunSpider benchmark

These type annotations will be extracted during the execution of the instrumented version of the program.
The types constraints that are collected during execution of this program are shown in Figure 16. These constraints are then condensed and checked against the trusted type annotations. The output of this process is shown in Figure 17.

```
'frame global':                i:        number            k:        number
  sieve:    'function sieve'    m:        number          'function nsieve':
  Array:    'function Array'    flags:    Array             arg0:     number
  nsieve:   'function nsieve' 'frame nsieve':               arg1:     Array
  result:   number              i:        number            return:   number
  expected: number              m:        number          'function sieve':
'frame sieve':                  isPrime:  Array             return:   number
  sum:      number              count:    number
```

Figure 16: Constraints of Listing 2

The program returns two type errors for this program, namely "pad not observed in frame global" and "function pad not observed". This is because the function "pad" is never called, and therefore the function was never observed and no constraints were generated. As we will see in the next section, this is a very common error. We observed

```
We detected 2 type error(s)

pad not observed in frame global
function pad not observed

We inferred the following types:

frame global has the following properties:
    sieve with type: function sieve
    Array with type: function Array
    nsieve with type: function nsieve
    result with type: number(14302)
    expected with type: number(14302)
frame sieve has the following properties:
    sum with type: number(T)
    i with type: number(T)
    m with type: number(T)
    flags with type: Array
frame nsieve has the following properties:
    i with type: number(T)
    m with type: number(T)
    isPrime with type: Array
    count with type: number(T)
    k with type: number(T)
function nsieve has the following type:
    arg0 number(T) -> arg1 Array -> return number(T)
function sieve has the following type:
    return number(14302)
```

Figure 17: Output for access-nsieve.js

no warnings for this program.

3.5 Evaluation

In this section, we evaluate our implementation by applying it to the SunSpider benchmark. All tests are performed on an Apple iMac with a 2.7GHz quad-core Intel Core i5 processor with 8 GB memory.

3.5.1 Benchmark Programs

We make use of the SunSpider 1.0.2 benchmark [3] for our evaluation. This benchmark was designed to simulate real world use of JavaScript. The benchmark uses basic JavaScript, in the sense that it does not use the DOM or other browser specific APIs. Some of the real world uses it includes are cryptography, raytracing, code decompression and date formatting, among many others. For a complete list, see Table 18. To give the reader an idea of the size of these programs, we have included the number of lines of code (LOC). This count excludes blank lines and comments.

In order to really put our implementation to the test, we need to run it on programs with type annotations. Unfortunately, these programs are not readily available. We have looked at programs written in TypeScript, but these do not allow for easy translation to JavaScript and our particular type annotation syntax.
Instead, we have written type annotations for the full SunSpider benchmark. Our type annotations can be found in appendix B. While writing these type annotations, we found out that our approach in implementing type annotations is too limited for real world applications. As we will see later, this caused problems for two programs. For the other programs, we were able to provide appropriate type annotations.

Now that we have our benchmark programs ready, we are able to evaluate our implementation. In order to measure the running time, we make use of the timing API of Node.js, implemented by extending the console object [1]. This allows us to start and stop timers directly in the JavaScript source code.

3.5.2 Results

Table 18 lists the evaluation results. Every program is run five times, the times listed are averages. It is clear that the execution time of the instrumented programs is much longer, about 1000 times. This is expected since every read, write and function call is instrumented with additional code. Pradel et al. [16] did not list their running time, An et al. have a similar increase in execution time for their Ruby system [6]. These execution times are however still acceptable, since the instrumented code is only run by the developer of a program to infer types and find errors. He can then fix these errors and will ship the fast, uninstrumented code.

All three warning pruning methods were turned on, the maximum degree of inconsistency and max difference were both set at 2. This setting is suggested by Pradel et al. [16]. As mentioned before, type inconsistency is beyond the scope of this thesis, and therefore we have not experimented with different settings.

The program "string-tagcloud" did not work with out implementation. Upon running the instrumented code, the program crashes. Therefore, we do not report any results for this program.

Appendix C lists the evaluation time, broken down per step. On average, the additional steps after execution only takes a few milliseconds, and is not significant on the total execution time.

Figure 19 shows the errors broken down by cause and the warnings classified by true and false positives. To further evaluate our implementation, we will look into the cause of these numbers. The next two paragraph discuss the errors and warnings in detail.

3.5.2.1 Errors

We have observed type errors in several programs from the benchmark. Remember that the type annotations are not provided in the benchmark, and written up especially for this evaluation.
To better understand where these errors come from, we have inspected every error and the code that caused it. A summary of the results from this inspection is listed below.

Unused functions 114 of the 139 errors are caused by unused functions. A lot of the benchmark programs define functions that are called used. These functions are

Program	LOC	Baseline	Dynamic Type Inference		
		time	time	#errors	#inconsistencies
3d-cube	301	16ms	20.491s	3	1
3d-morph	26	15ms	8.810s	0	0
3d-raytrace	348	15ms	11.685s	11	6
access-binary-trees	41	3ms	4.447s	0	0
access-fannkuch	54	8ms	49.606s	0	0
access-nbody	145	4ms	16.329s	6	0
access-nsieve	33	4ms	11.457s	2	0
bitops-3bit-bits-in-byte	19	2ms	10.092s	0	0
bitops-bits-in-byte	20	4ms	17.217s	0	0
bitops-bitwise-and	7	3ms	7.920s	0	0
bitops-nsieve-bits	29	4ms	13.504s	2	0
controlflow-recursive	22	3ms	6.513s	0	0
crypto-aes	291	10ms	13.914s	8	1
crypto-md5	215	6ms	7.996s	24	12
crypto-sha1	150	6ms	7.135s	19	2
date-format-tofte	211	17ms	12.969	47	5
date-format-xparb	378	15ms	3.381s	0	2
math-cordic	59	4ms	23.389s	2	0
math-partial-sums	31	12ms	6.338s	11	0
math-spectral-norm	45	4ms	8.275s	0	0
regexp-dna	1702	9ms	0.193s	1	0
string-base64	69	9ms	4.850s	0	0
string-fasta	73	12ms	8.224s	0	0
string-tagcloud	181	32ms	ERR		
string-unpack-code	14	27ms	3.264s	0	0
string-validate-input	74	13ms	4.340s	3	1
total	4538	257ms	282.240s	139	30

Figure 18: Summary of results

	Total	3d-cube	3d-raytrace	access-nbody	access-nsieve	bitops-nsieve-bits	crypto-aes	crypto-md5	crypto-sha1	date-format-tofte	date-format-xparb	math-cordic	math-partial-sums	regexp-dna	string-validate-input
Unused Functions	114	3	2	0	2	2	8	18	18	47	0	2	11	1	0
Type Limits	15	0	9	6	0	0	0	0	0	0	0	0	0	0	0
Native Functions	3	0	0	0	0	0	0	0	0	0	0	0	0	0	3
Errors	7	0	0	0	0	0	0	6	1	0	0	0	0	0	0
False Errors	13	0	6	0	0	0	1	0	0	5	0	0	0	0	1
True Errors	17	1	0	0	0	0	0	12	2	0	2	0	0	0	0

Figure 19: Breakdown of errors and warnings

annotated, and therefore generate trusted constraints. When our implementation tries to verify them, it generates one or more error since it is not present in the inferred types.

Type Limits 15 of the 139 errors are related to the limitations of our type annotation system. As mentioned before, for some programs we were not able to write appropriate type annotations. It turned out that our implementation of annotations was too limited for the use in real-world programs. To be more specific, programmers are unable to write recursive type annotations. The keyword self allows recursive types to refer to themselves, but there is no deeper support for recursive type annotations.

native functions 3 of the 139 errors are related to native functions. Although our system has support for native (built-in) functions, during evaluation it turned out that the support is not complete yet. The program "string-validate-input" uses regular expressions. First, a string containing the regular expression is defined, then the function "test" is called, such that the string now becomes an object with the RegExp prototype.

Actual error 7 of the 139 errors are actual programming errors. The programs "crypto-md5" and "crypto-sha1" both contained problematic code. In both cases, padding with "undefined" is observed. These problems were also discovered by Pradel et al. [16].

3.5.2.2 Inconsistencies

We have detected 32 inconsistencies in eight SunSpider programs. All inconsistencies have different causes. Therefore, we will discuss all the eight programs separately.

3d-cube This program contains a function that returns "undefined" if it is called with a certain value of the arguments. In all other cases it returns an Array. This causes problems when the program tries to access the array.

3d-raytrace All six warnings in this program are due to the fact that there is one function that is called with a variable number of arguments. The function is defined as having four arguments. When it is called with less, the remaining arguments get the value "undefined". These arguments now have two types, namely undefined and their actual type. This causes further warnings, because the frame of the function now also has properties with inconsistent types. These warnings are false though, because this use was intended.

crypto-aes In this program, one property in a frame has the inconsistent type string and Array. As long as the array contains characters, this is fine.

crypto-md5 This program caused twelve warnings that are related to an actual problem with the program. This is the same problem as we found while discussing the errors.

crypto-sha1 This problem caused two warnings that are related to an actual problem with the program. This is the same problem as we found while discussing the errors.

date-format-tofte This program caused five warnings. All warnings are related to the fact that four functions have both string and number as a return type. These functions return a value, with padding of zeros preceding the value. When there are preceding zeros, the return value is typed as string, otherwise as a number.

date-format-xparb This program caused one warning, that is related to an actual problem. The function "leftPad" returns a string or an object, depending on the length of its argument. Therefore the return property of this function has an inconsten type. This problem is also found and discussed by Pradel et al. [16]. While inspecting this function, we also discovered that in this program, "String" is used as a constructor. While not wrong, this is certainly bad practice.

string-validate-input This program yields one warning. This is a false positive. Our analysis cannot deal properly with the regular expression used by the program. This was also the case for the errors generated by this program.

As shown by the discussion above, our analysis yields useful errors and warnings that can be used by programmers to increase the quality of their programs.

4 Conclusion

With this master thesis, we have shown that dynamic type inference for JavaScript is indeed feasible. In order to demonstrate that the general idea is useful, we have developed a sound formal system for a core JavaScript language. To demonstrate that the concept of dynamic type inference for JavaScript is also useful in practice, we have developed a practical implementation based on the same principles as the formal system.

4.1 Formal system

We have implemented a complete formal dynamic type inference system for JavaScript. Our system consists of a core JavaScript language, a type language and semantics. We have shown that our formal implementation is sound.

4.2 Implementation

We have implemented a prototype dynamic type inference system for JavaScript. We have evaluated our system on benchmark programs. From this evaluation we got useful errors and warnings that allow developers to improve the quality of their JavaScript code.

4.3 Future work

Future work on this subject could be extending both the formal system and the implementation as mentioned below.

4.3.1 Formal system

We could extend the formal system in several different ways. First of all, we could extend it to cover a larger subset of Javascript, possibly even the whole language. The core language used for this thesis excludes some concepts like while, arrays, let-expressions, deletion of properties, etc. It would be very interesting to see how the formal system has to be adapted to deal with these language concepts. An et al. [5] have shown how to deal with loops for their formal system developed for Ruby. Basic concepts used here could also be used to develop the extension of our implementation.

Another direction the formal system could be extended in, is in the types. Currently, our system only collects very basic types. This could for example be extended to singleton types as defined by Thiemann [19]. In this work, his types also hold a value. If you only observe one value, you store that value in the type. If you also observe different values, you record $\top$. We have implemented these singleton types in our practical system.

Besides extending the type information we collect, we could also extend the way we handle types. The technical report of An et al. [5] has some very interesting suggestions

that could also be usefull for our system. They report on a formal implementation of intersection types for methods and support for trusted and untrusted type annotations.

4.3.2 Implementation

Our practical implementation has several opportunities for future work. First of all, our implementation of type annotations should be extended to support recursive types better.

An interesting topic of future work could also be improving the execution time of our algorithm. Our implementation could perhaps be made a bit more efficient, although results by other authors are not very encouraging.

An et al. [6] include a coverage measure in their implementation of dynamic type inference for Ruby. This measure is an indication of how trustworthy the inferred types are. They measure both line coverage and method coverage.

Additionally, we could work on more advanced pruning methods for warnings. As mentioned, these pruning methods are beyond the scope of this thesis, but that does not make them less interesting. Pradel et al. [16] presents eight methods of pruning, three of which we have implemented. But he presents these methods without proof for their correctness, either theoretical or practical. Some of these methods require fine tuning. Investigating the correctness and fine tuning of these pruning methods could also be subject of future work.

References

[1] Node.js v0.12.4 Manual & Documentation. `https://nodejs.org/api`, 2014. [Online; accessed 12-June-2015].

[2] Samsung/jalangi2 GitHub. `https://github.com/Samsung/jalangi2`, 2015. [Online; accessed 9-July-2015].

[3] SunSpider 1.0.2 JavaScript Benchmark. `https://www.webkit.org/perf/sunspider/sunspider.html`, 2015. [Online; accessed 22-June-2015].

[4] Ole Agesen. The cartesian product algorithm. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 2–26. Springer, 1995.

[5] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. Technical Report CS-TR-4965, Computer Science Department, University of Maryland, 2010.

[6] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. *SIGPLAN Not.*, 46(1):459–472, January 2011.

[7] Christopher Anderson and Paola Giannini. Type checking for javascript. *Electron. Notes Theor. Comput. Sci.*, 138(2):37–58, November 2005.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Robert Cartwright and Mike Fagan. Soft typing. *SIGPLAN Not.*, 39(4):412–428, April 2004.

[10] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. *SIGPLAN Not.*, 47(6):239–250, June 2012.

[13] Phillip Heidegger. Increasing software quality of javascript programs. 2012.

[14] Robert Jakob and Peter Thiemann. A falsification view of success typing. *CoRR*, abs/1502.01278, 2015.

[15] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. 2014.

[17] Michael Salib. Faster than c: Static type inference with starkiller. *PyCon Proceedings, Washington DC*, 3, 2004.

[18] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 615–618, New York, NY, USA, 2013. ACM.

[19] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, January 1997.

Appendices

A Error propagation rules

$$\textsc{MPCallError}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; p(e) \mid \phi_m \lightning} \qquad \textsc{MNewError}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \text{new } e \mid \phi_m \lightning} \qquad \textsc{MVarAssError}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; x = e \mid \phi_m \lightning} \qquad \textsc{MPropError}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; e.n \mid \phi_m \lightning}$$

$$\textsc{MSeqError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; (e; e') \mid \phi_m \lightning} \qquad \textsc{MSeqError2}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; \_ \mid \phi'_m \quad H'_m; S'_m; e' \mid \phi'_m \lightning}{H_m; S_m; (e; e') \mid \phi_m \lightning} \qquad \textsc{MPropAssError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; e.n = e' \mid \phi_m \lightning}$$

$$\textsc{MPropAssError2}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l \mid \phi'_m \quad H'_m; S'_m; e' \mid \phi'_m \lightning}{H_m; S_m; e.n = e' \mid \phi_m \lightning} \qquad \textsc{MTLPropError}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \underline{e.n} \mid \phi_m \lightning} \qquad \textsc{MConditionalError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \lightning}$$

$$\textsc{MConditionalError2}\ \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; v \mid p, \phi'_m \\ \text{if } (c \notin \text{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \\ H'_m; S'_m; e_p \mid \phi'_m \lightning \end{array}}{H_m; S_m; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi_m \lightning} \qquad \textsc{MCallError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; x(e) \mid \phi_m \lightning}$$

$$\textsc{MCallError2}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \quad H'_m; S'_m; e' \mid \phi'_m \lightning}{H_m; S_m; e(e') \mid \phi_m \lightning}$$

$$\textsc{MCallError3}\ \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \\ H'_m; S'_m; e' \mid \phi'_m \longrightarrow H''_m; S''_m; v_m \mid \phi''_m \\ H''_m(l_m) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \\ \$\text{vars} \mapsto S^f, \$\text{tyvar} \mapsto \alpha_m) \\ S^{f'} = S^f\{f \mapsto n, x^f \mapsto v_m, y^f \mapsto \texttt{udf}\} \\ \bar{\phi_m} \in \phi_m \qquad H''_m; S^{f'}; e^f \mid \bar{\phi_m} \lightning \end{array}}{H_m; S_m; e(e') \mid \phi_m \lightning}$$

$$\textsc{MTLPropAssError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \underline{e.n = e'} \mid \phi_m \lightning} \qquad \textsc{MTLPropAssError2}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \quad H'_m; S'_m; e' \mid \phi'_m \lightning}{H_m; S_m; \underline{e.n = e'} \mid \phi_m \lightning} \qquad \textsc{MTLCallError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \underline{e(e')} \mid \phi_m \lightning}$$

$$\textsc{MTLCallError1}\ \frac{H_m; S_m; e \mid \phi_m \lightning}{H_m; S_m; \underline{e(e')} \mid \phi_m \lightning} \qquad \textsc{MTLCallError2}\ \frac{H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \qquad H'_m; S'_m; e' \mid \phi'_m \lightning}{H_m; S_m; \underline{e(e')} \mid \phi_m \lightning}$$

$$\textsc{MTLCallError3}\ \frac{\begin{array}{c} H_m; S_m; e \mid \phi_m \longrightarrow H'_m; S'_m; l_m \mid \phi'_m \\ H'_m; S'_m; e' \mid \phi'_m \longrightarrow H''_m; S''_m; v_m \mid \phi''_m \qquad H''_m(l_m) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \\ \$\text{vars} \mapsto S^f, \$\text{tyvar} \mapsto \alpha_m, \ldots) \qquad S^{f'} = S^f\{f \mapsto n, x^f \mapsto v_m, (y \mapsto \texttt{udf})^*\} \\ \text{runtype}_{H'}(v_m) \leq \mathcal{T}(H'(l_m)[n]_{arg}) \qquad \bar{\phi_m} \in \phi_m \qquad H'_m; S^{f'}; e^f \mid \bar{\phi_m} \lightning \end{array}}{H_m; S_m; \underline{e(e')} \mid \phi_m \lightning}$$

B SunSpider program annotations

```
"frame:[Q:Array|null,MTrans:Array|null,MQube:Array|null,I:Array|null,Origin:null|[
    V:Array],LoopTimer:number|null]"
"frame:[Testing:null|[LoopCount:number,LoopMax:number,TimeMax:number,TimeAvg:
    number,TimeMin:number,TimeTemp:number,TimeTotal:number,Init:boolean]]"
"function DrawLine:{[V:Array]->[V:Array]->undefined}"
function DrawLine(From, To) {
  "frame:[x1:number,x2:number,y1:number,y2:number,dx:number,dy:number,x:number,y:
      number,IncX1:number,IncX2:number,IncY1:number,IncY2:number,Den:number,Num:
      number,NumAdd:number,NumPix:number,i:number]"
  ...
}
"function CalcCross:{Array->Array->Array}"
"function CalcNormal:{Array->Array->Array->Array}"
"function CreateP:{number->number->number->undefined|[V:Array]}"
"function MMulti:{Array->Array->Array}"
"function VMulti:{Array->Array->Array}"
"function VMulti2:{Array->Array->Array}"
"function MAdd:{Array->Array->Array}"
"function Translate:{Array->number->number->number->Array}"
"function RotateX:{Array->number->Array}"
"function RotateY:{Array->number->Array}"
"function RotateZ:{Array->number->Array}"
"function DrawQube:{undefined}"
"function Loop:{undefined}"
"function Init:{number->undefined}"
```

Listing 3: Type annotations for 3d-cube.js

```
"frame:[loops:number,nx:number,nz:number,a:Array|null,epsilon:number]"
"function morph:{Array->number->undefined}"
```

Listing 4: Type annotations for 3d-morph.js

```
"function createVector:{number->number->number->Array}"
"function sqrLengthVector:{Array->number}"
"function lengthVector:{Array->number}"
"function addVector:{Array->Array->Array}"
"function subVector:{Array->Array->Array}"
"function scaleVector:{Array->number->Array}"
"function normaliseVector:{Array->Array}"
"function add:{Array->Array->Array}"
"function sub:{Array->Array->Array}"
"function scalev:{Array->Array->Array}"
"function dot:{Array->Array->number}"
"function scale:{Array->number->Array}"
"function cross:{Array->Array->Array}"
"function normalise:{Array->Array}"
"function transformMatrix:{Array->Array->Array}"
"function invertMatrix:{Array->Array}"
"function Triangle:{Array->Array->Array->[axis:number,normal:Array,nu:number,nv:
    number,nd:number,eu:number,ev:number,nu1:number,nv1:number,nu2:number,nv2:
    number,material:Array]}"
function Triangle(p1, p2, p3) {
    "frame:[edge1:Array,edge2:Array,normal:Array,u:number,v:number,u1:number,v1:
        number,u2:number,v2:number,det:number]"
    ...
}
Triangle.prototype.intersect = function(orig, dir, near, far) {
```

```
    "frame:[u:number,v:number,d:number,t:number,Pu:number,Pv:number,a2:number,a3:
        number]"
    ...
}
"function Scene:{Array->[triangles:Array,lights:Array,ambient:Array,background:
    Array]}"
"function Camera:{Array->Array->Array->[origin:Array,directions:Array,
    generateRayPair:{number->Array},render:{[intersect:{Array->Array->number->
    number->Array},blocked:{Array->Array->number->boolean},triangles:Array,lights:
    Array,ambient:Array,background:Array]->Array->number->number->undefined}]}"
"function renderRows:{[origin:Array,directions:Array]->[intersect:{Array->Array->
    Array},blocked:{Array->Array->number->boolean},triangles:Array,lights:Array,
    ambient:Array,background:Array]->Array->number->number->number->[origin:Array,
    directions:Array]->undefined}"
Camera.prototype.render = function(scene, pixels, width, height) {
    "frame:[cam:[origin:Array,directions:Array],row:number]"
    ...
}
"function raytraceScene:{Array}"
"function arrayToCanvasCommands:{Array->string}"
```

Listing 5: Type annotations for 3d-raytrace .js

```
"frame:[minDepth:number,maxDepth:number,stretchDepth:number,ret:number,check:
    number,longLivedTree:[left:self|null,right:self|null,item:number,itemCheck:{
    number}]]"
"function TreeNode:{null|[left:self|null,right:self|null,item:number,itemCheck:{
    number}]->null|[left:self|null,right:self|null,item:number,itemCheck:{number
    }]->number->[left:self|null,right:self|null,item:number,itemCheck:{number}]}"
"function bottomUpTree:{number->number->[left:self|null,right:self|null,item:
    number,itemCheck:{number}]}"
```

Listing 6: Type annotations for access-binary-trees.js

```
"function fannkuch:{number->number}"
function fannkuch(n) {
   "frame:[check:number,perm:Array,perm1:Array,count:Array,maxPerm:Array,
       maxFlipsCount:number,m:number,r:number,flipsCount:number,s:string]"
  ...
}
```

Listing 7: Type annotations for access-fannkuch.js

```
"function Body:{number->number->number->number->number->number->[x:number,
    y:number,z:number,vx:number,vy:number,vz:number,mass:number]}"
"function Jupiter:{[x:number,y:number,z:number,vx:number,vy:number,vz:number,mass:
    number]}"
"function Saturn:{[x:number,y:number,z:number,vx:number,vy:number,vz:number,mass:
    number]}"
"function Uranus:{[x:number,y:number,z:number,vx:number,vy:number,vz:number,mass:
    number]}"
"function Neptune:{[x:number,y:number,z:number,vx:number,vy:number,vz:number,mass:
    number]}"
"function Sun:{[x:number,y:number,z:number,vx:number,vy:number,vz:number,mass:
    number]}"
"function NBodySystem:{Array->[advance:{number->undefined},energy:{number},bodies:
    Array]}"
NBodySystem.prototype.advance = function(dt){
   "frame:[dx:number,dy:number,dz:number,distance:number,mag:number,size:number]"
    ...
```

```
}
NBodySystem.prototype.energy = function(){
    "frame:[dx:number,dy:number,dz:number,distance:number,e:number,size:number]"
    ...
}
```

Listing 8: Type annotations for access-nbody.js

```
"function pad:{number->number->string}"
"function nsieve:{number->Array->number}"
"function sieve:{number}"
```

Listing 9: Type annotations for access-nsieve.js

```
"function fast3bitlookup:{number->number}"
"function TimeFunc:{{number->number}->number}"
```

Listing 10: Type annotations for bitops-3bit-bits-in-byte.js

```
"function bitsinbyte:{number->number}"
"function TimeFunc:{{number->number}->number}"
```

Listing 11: Type annotations for bitops-bits-in-byte.js

```
"frame:[bitwiseAndValue:number,i:number,result:number,expected:number]"
```

Listing 12: Type annotations for bitops-bitwise-and.js

```
"function pad:{number->number->string}"
"function primes:{Array->number->undefined}"
"function sieve:{Array}"
```

Listing 13: Type annotations for bitops-nsieve-bits.js

```
"function ack:{number->number->number}"
"function fib:{number->number}"
"function tak:{number->number->number->number}"
```

Listing 14: Type annotations for controlflow-recursive.js

```
"function Cipher:{Array->Array->Array}"
"function SubBytes:{Array->number->Array}"
"function ShiftRows:{Array->number->Array}"
"function MixColumns:{Array->number->Array}"
"function AddRoundKey:{Array->Array->number->number->Array}"
"function KeyExpansion:{Array->Array}"
"function SubWord:{Array->Array}"
"function RotWord:{Array->Array}"
"function AESEncryptCtr:{string->string->number->string}"
"function AESDecryptCtr:{string->string->number->string}"
"function escCtrlChars:{string->string}"
"function unescCtrlChars:{string->string}"
"function encodeBase64:{string->string}"
"function decodeBase64:{string->string}"
"function encodeBase64:{string->string}"
"function decodeUTF8:{string->string}"
"function byteArrayToHexStr:{Array->string}"
```

Listing 15: Type annotations for crypto-aes.js

```
"function hex_md5:{string->string}"
"function b64_md5:{string->string}"
"function str_md5:{string->string}"
"function hex_hmac_md5:{string->string->string}"
"function b64_hmac_md5:{string->string->string}"
"function str_hmac_md5:{string->string->string}"
"function md5_vm_test:{undefined->boolean}"
"function core_md5:{Array->number->Array}"
"function md5_cmn:{number->number->number->number->number->number->number}"
"function md5_ff:{number->number->number->number->number->number->number}"
"function md5_gg:{number->number->number->number->number->number->number}"
"function md5_hh:{number->number->number->number->number->number->number}"
"function md5_ii:{number->number->number->number->number->number->number}"
"function core_hmac_md5:{string->string->string}"
"function safe_add:{number->number->number}"
"function bit_rol:{number->number->number}"
"function str2binl:{string->Array}"
"function binl2str:{string->string}"
"function binl2hex:{Arrat->string}"
"function binl2b64:{string->string}"
```

Listing 16: Type annotations for crypto-md5.js

```
"function hex_sha1:{string->string}"
"function b64_sha1:{string->string}"
"function str_sha1:{string->string}"
"function hex_hmac_sha1:{number->string->string}"
"function b64_hmac_sha1:{number->string->string}"
"function str_hmac_sha1:{number->string->string}"
"function sha1_vm_test:{undefined->boolean}"
"function core_sha1:{Array->number->Array}"
"function sha1_ft:{number->number->number->number->number}"
"function sha1_kt:{number->number}"
"function core_hmac_sha1:{number->string->string}"
"function safe_add:{number->number->number}"
"function rol:{number->number->number}"
"function str2binb:{string->Array}"
"function binb2str:{string->string}"
"function binb2hex:{Array->string}"
"function binb2b64:{Array->string}"
```

Listing 17: Type annotations for crypto-sha1.js

```
"function arrayExists:{Array->string->boolean}"
Date.prototype.formatDate = function (input,time) {
    "frame:[switches:Array,daysLong:Array,daysShort:Array,monthsShort:Array,
        monthsLong:Array,daysSuffix:Array]"
    "function a:{string}"
    "function A:{string}"
    "function B:{string}"
    "function d:{string|number}"
    "function D:{string}"
    "function F:{string}"
    "function g:{number}"
    "function G:{string}"
    "function h:{string}"
    "function H:{string}"
    "function i:{string|number}"
    "function j:{number}"
    "function l:{string}"
    "function L:{number}"
```

```
    "function m:{string|number}"
    "function M:{string}"
    "function n:{number}"
    "function O:{number}"
    "function r:{string}"
    "function S:{number}"
    "function s:{number|string}"
    "function t:{number}"
    "function U:{number}"
    "function W:{number}"
    "function w:{number}"
    "function Y:{number}"
    "function y:{number}"
    "function z:{number}"
    ...}
```

Listing 18: Type annotations for date-format-tofte.js

```
Date.createNewFormat = function(format) {
    "frame:[funcName:string,code:string,special:boolean,ch:string]"
...}
Date.createParser = function(format) {
    "frame:[funcName:string,regexNum:number,currentGroup:number,code:String,regex:
        string,special:boolean,ch:string]"
...}
Date.prototype.getDayOfYear = function() {
    "frame:[num:number]"
...}
Date.prototype.getWeekOfYear = function() {
    "frame:[now:string,jan1:string,then:string]"
...}
Date.prototype.isLeapYear = function() {
    "frame:[year:string]"
...}
```

Listing 19: Type annotations for date-format-xparb.js

```
"function FIXED:{number->number}"
"function FLOAT:{number->number}"
"function DEG2RAD:{number->number}"
"function cordicsincos:{number->number}"
"function cordic:{number->number}"
```

Listing 20: Type annotations for math-cordic.js

```
"function partial:{number->number}"
function partial(n){
    "frame:[a1:number,a2:number,a3:number,a4:number,a5:number,a6:number,a7:number,
        a8:number,a9:number,twothirds:number,alt:number,k2:number,k3:number,sk:
        number,ck:number]"
...}
```

Listing 21: Type annotations for math-partial-sums.js

```
"function A:{number->number->number}"
"function Au:{Array->Array->undefined}"
"function Atu:{Array->Array->undefined}"
"function AtAu:{Array->Array->Array->undefined}"
"function spectralnorm:{number->number}"
```

Listing 22: Type annotations for math-spectral-norm.js

```
"frame:[l:string,dnaInput:string,seqs:Array,subs:[B:string,D:string,H:string,K:
    string,M:string,N:string,R:string,S:string,V:string,W:string,Y:string],ilen:
    number,clen:number,dnaOutputString:string,i:string,k:string,
    expectedDNAOutputString:string,expectedDNAInput:string]"
```

Listing 23: Type annotations for regexp-dna.js

```
"function toBase64:{string->string}"
"function base64ToString:{string->string}"
```

Listing 24: Type annotations for string-base64.js

```
"function rand:{number->number}"
"function makeCumulative:{[c:number,a:number,g:number,t:number,B:number,D:number,H
    :number,K:number,M:number,N:number,R:number,S:number,V:number,W:number,Y:
    number]->undefined}"
"function fastaRandom:{number->[c:number,a:number,g:number,t:number]->undefined}"
```

Listing 25: Type annotations for string-fasta.js

```
"frame:[result:number,decompressedMochiKit:string,decompressedJQuery:string,
    decompressedDojo:string,decompressedPrototype:string]"
```

Listing 26: Type annotations for string-unpack-code.js

```
"function doTest:{undefined}"
function doTest(){
   "frame:[pattern:[test:?],zipGood:boolean,zip:string,i:number,ch:string]"
...}
"function makeName:{number->string}"
"function makeNumber:{number->number}"
"function addResult:{string->undefined}"
```

Listing 27: Type annotations for string-validate-input.js

Analyzing string-tagcloud.js results in an error, therefore we don't provide annotations.

C Breakdown of Execution Time

Program	LOC	Baseline	Dynamic Type Inference			
		time	total time	condensing type graph	generating errors	generating warnings
3d-cube	301	16ms	20.491s	2ms	1ms	1ms
3d-morph	26	15ms	8.810s	1ms	1ms	1ms
3d-raytrace	348	15ms	11.685s	2ms	1ms	2ms
access-binary-trees	41	3ms	4.447s	1ms	1ms	1ms
access-fannkuch	54	8ms	49.606s	1ms	0ms	1ms
access-nbody	145	4ms	16.329s	1ms	1ms	1ms
access-nsieve	33	4ms	11.457s	1ms	1ms	1ms
bitops-3bit-bits-in-byte	19	2ms	10.092s	1ms	0ms	1ms
bitops-bits-in-byte	20	4ms	17.217s	1ms	0ms	1ms
bitops-bitwise-and	7	3ms	7.920s	0ms	0ms	1ms
bitops-nsieve-bits	29	4ms	13.504s	1ms	1ms	1ms
controlflow-recursive	22	3ms	6.513s	0ms	0ms	1ms
crypto-aes	291	10ms	13.914s	2ms	1ms	1ms
crypto-md5	215	6ms	7.996s	1ms	1ms	1ms
crypto-sha1	150	6ms	7.135s	1ms	0ms	1ms
date-format-tofte	211	17ms	12.969	5ms	0ms	1ms
date-format-xparb	378	15ms	3.381s	109ms	0ms	1ms
math-cordic	59	4ms	23.389s	1ms	0ms	1ms
math-partial-sums	31	12ms	6.338s	0ms	0ms	1ms
math-spectral-norm	45	4ms	8.275s	1ms	0ms	1ms
regexp-dna	1702	9ms	0.193s	1ms	0ms	1ms
string-base64	69	9ms	4.850s	1ms	1ms	1ms
string-fasta	73	12ms	8.224s	1ms	0ms	1ms
string-tagcloud	181	32ms	ERR	ERR	ERR	ERR
string-unpack-code	14	27ms	3.264s	26ms	0ms	18ms
string-validate-input	74	13ms	4.340s	1ms	1ms	1ms

Figure 20: Breakdown of execution time