A Typed Intermediate Language for Specifying the ECMAScript Standard

André Filipe Ferreira do Nascimento

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. Nuno Miguel Carvalho dos Santos
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. João Costa Seco

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor José Fragoso dos Santos, for his constant guidance and exceptional support, as without him this work would have never come to fruition. In look forward to continue our work throghout the following years.

Um agradecimento muito especial à minha família, particularmente aos meus pais e à minha irmã. Agradeço por terem-me sempre dado a possibilidade de fazer as minhas escolhas e seguir os meus interesses, e por me apoiarem sempre, mesmo nos momentos em que uma parte significativa do meu tempo foi dedicada ao trabalho, muitas vezes deixando-vos para terceiro e quarto plano. O vosso apoio incansável ao longo destes anos tem sido inestimável e eu não estaria onde estou sem vocês.

A toda a CPLEIC, por me terem providenciado os melhores momentos que passei em Lisboa. Em particular, gostava de agradecer aos meus meninos do meu ano: à minha companheira de whisky, Mia, por ter sido uma amiga incondicional, por me ter aturado nos bons e maus momentos, e por me ter ”obrigado” a voltar a esta casa após dois longos anos; ao Gundas, pelas boleias no Micra, pelos múltiplos coktails, e por todas as parvoíces e momentos hilariantes; à Sancha e à Chinopa por terem sido amigas incríveis e por me terem acompanhado e apoiado sempre. Um obrigado também a todos os que vieram antes, por todos os ensinamentos e por nunca terem desistido de mim. Aos que cá ficam, foi um gosto acompanhar o vosso crescimento, e não podia estar mais orgulhoso pelas pessoas em que se estão a tornar. Espero que continuem a lutar por esta casa e que ela vos traga tanta felicidade quanto me trouxe a mim.

A todos os elementos do Bar-Wings, Iúri, Catarina, e Diogo, por todos os bons momentos, por todos os bons treinos, e por me terem ajudado a manter o equilíbrio na minha vida. É fundamental lembrar que, embora o curso seja importante, há outros aspectos que não devem ser esquecidos, nomeadamente o nosso bem estar físico. Aguardo ansiosamente pelos próximos anos, por tudo aquilo que iremos construir no clube, e pelo impacto que iremos ter na Calistenia a nível nacional.

Aos meus companheiros do CDP e arredores, Luís, Barbas, Chico Manel, António, Reboleira, e Tiririca, que têm sido pilares desde que tenho memória. Obrigado por todas as noites incríveis no Central e por permanecerem ao meu lado, ano após ano, independentemente do rumo que as nossas vidas tomem. Tenho a mais absoluta certeza de que cá vão estar para sempre.

Aos meus colegas de projeto, Bento e Malveiro, que me aturaram a ser insuportável com detalhes insignificantes, que não me matavam sempre que eu refatorizava 80% de um projeto, e que me ajudaram a manter a sanidade quando as deadlines estavam a apertar. Qualquer projeto do Técnico se torna fácil quando se tem um grupo de trabalho invejável, detentor do melhor chat do Messenger.

A todas as outras pessoas que de uma maneira ou outra ajudaram a melhorar a minha vida. À Chumbo por ser uma amiga incrível e por me arrastar para Coimbra e à minha afilhada linda por me dar teto em Lisboa. À Carorlina, Margarida, Vanessa, Lopes, Capinha, Jeni, Sofia, Nascimento$_2$, Afonso, Sr. Nuno, às Tragédias, e a todos os outros não mencionados mas que aqui pertencem.

... a todos vós, o mais sentido obrigado.

Abstract

ECMA-SL is a new platform for the specification and analysis of the ECMAScript standard. At the core of the ECMA-SL platform is ECMA-SL, a new intermediate language designed to be as close as possible to the meta-language of the ECMAScript standard. Using this intermediate language, the ECMA-SL team has developed two reference interpreters for the 5th and 6th versions of the ECMAScript standard, ECMARef5 and ECMARef6, respectively. Both interpreters were thoroughly tested against Test262, the official conformance test suite for JavaScript, and are currently the most complete academic implementations of the standard for their respective versions. An important shortcoming of the ECMA-SL language is that it is untyped, making ECMA-SL programs extremely difficult to debug and refactor. To address this issue, we introduce Typed ECMA-SL, an extension of ECMA-SL that adds static typing with optional type annotations to the language. Typed ECMA-SL comes with a static type system for verifying the adherence of programs to the supplied type declarations and provides user-friendly feedback when type errors occur. We formalized a subset of Typed ECMA-SL, proving the soundness of our type system for the formalized fragment. Additionally, we implemented and typed a simplified JavaScript interpreter with a similar structure to the reference interpreters. This suggests that we will be able to fully type the reference interpreters, enabling their extension to newer versions of the ECMAScript standard.

Keywords

ECMAScript; ECMA-SL; ECMARef6; Typed ECMA-SL; Type Declarations; Type System.

Resumo

ECMA-SL é uma nova plataforma para a especificação e análise do ECMAScript standard. No núcleo desta plataforma está a ECMA-SL, uma nova linguagem intermédia desenhada para ser o mais semelhante possível à metalinguagem do ECMAScript standard. Com esta linguagem intermédia, a equipa do projeto ECMA-SL desenvolveu dois interpretadores de referência para a quinta e sexta versão do standard, ECMARef5 e ECMARef6, respetivamente. Ambos os interpretadores foram testados exaustivamente contra o Test262, o conjunto oficial de testes de conformidade para JavaScript, e constituem as implementações académicas mais completas do standard para as suas respetivas versões. Uma limitação importante da linguagem ECMA-SL é que não é tipificada, tornando os programas escritos em ECMA-SL extremamente difíceis de corrigir e refatorizar. Para atenuar este problema, introduzimos Typed ECMA-SL, uma extensão de ECMA-SL que adiciona tipificação estática com anotações de tipos opcionais. Typed ECMA-SL inclui um sistema de tipos estáticos para verificar a conformidade dos programas com as declarações de tipos e gera feedback intuitivo quando deteta erros. Formalizámos um subconjunto de Typed ECMA-SL, provando a consistência do sistema de tipos para o fragmento formalizado. Além disso, implementámos e tipificámos um interpretador simplificado de JavaScript com uma estrutura semelhante aos interpretadores de referência. Isto sugere que seremos capazes de tipificar completamente os interpretadores de referência, possibilitando a sua extensão para novas versões do ECMAScript standard.

Palavras Chave

ECMAScript; ECMA-SL; ECMARef6; Typed ECMA-SL; Declarações de Tipos; Sistema de Tipos.

Contents

List of Figures

List of Tables

List of Listings

List of Symbols

$FuncID$	The set of function identifiers of a Typed ECMA-SL program.
Var	The set of variables and parameters of a Typed ECMA-SL program.
$TVar$	The set of type variable identifiers of a Typed ECMA-SL program.
Loc	The set of object locations of a Typed ECMA-SL program.
Fld	The set of field names of a Typed ECMA-SL program.
$Stmt$	The set of Typed ECMA-SL statements.
$Expr$	The set of Typed ECMA-SL expressions.
Pat	The set of Typed ECMA-SL patterns.
$PatB$	The set of Typed ECMA-SL pattern bindings.
Val	The set of Typed ECMA-SL values.
Int	The set of Typed ECMA-SL integers.
Flt	The set of Typed ECMA-SL floats.
Str	The set of Typed ECMA-SL strings.
$Bool$	The set of Typed ECMA-SL booleans.
Obj	The set of Typed ECMA-SL object values.
T	The set of Typed ECMA-SL types.
$TPrim$	The set of Typed ECMA-SL primitive types.
$TObj$	The set of Typed ECMA-SL object types.
$TLit$	The set of Typed ECMA-SL literal types.
$TFunc$	The set of Typed ECMA-SL function types.
g	A function identifier in a Typed ECMA-SL program — $g \in Func$.
x	A variable or parameter in a Typed ECMA-SL program — $x \in Var$.
α	A type variable identifier in a Typed ECMA-SL program (or a bound variable) — $\alpha \in TVar$.
l	An object location in a Typed ECMA-SL program — $l \in Loc$.
f	A field name in a Typed ECMA-SL program — $f \in Fld$.

$\boldsymbol{s}$	A Typed ECMA-SL statement — $s \in Stmt$.
$\boldsymbol{e}$	A Typed ECMA-SL expression — $e \in Expr$.
$\boldsymbol{\psi}$	A Typed ECMA-SL pattern — $\psi \in Pat$.
$\boldsymbol{\xi}$	A Typed ECMA-SL pattern bindings — $\xi \in PatB$.
$\boldsymbol{v}$	A Typed ECMA-SL value — $v \in Val$.
$\boldsymbol{o}$	An object value (or heap region) from a Typed ECMA-SL program.
$\boldsymbol{\tau}$	A Typed ECMA-SL type — $\tau \in \mathcal{T}$.
$\boldsymbol{t}$	A Typed ECMA-SL primitive type — $t \in TPrim$.
$\boldsymbol{\sigma}$	A Typed ECMA-SL object type — $\sigma \in TObj$.
$\boldsymbol{\nu}$	A Typed ECMA-SL literal type — $\hat{v} \in TLit$.
$\ominus$	A Typed ECMA-SL unary operator.
$\oplus$	A Typed ECMA-SL binary operator.
$*$	A summary field.
$\boldsymbol{\tau}^*$	The type of a summary field.
$\cup$	A union type.
$\boldsymbol{\Sigma}[\boldsymbol{\alpha}]$	A sigma type.
$\boldsymbol{\beta}$	The conditional type refinement operator.
$\boldsymbol{\Psi}$	The pattern type binding operator.
$\boldsymbol{\Omega}$	A program context for Typed ECMA-SL.
$\boldsymbol{\Delta}$	A global typing context for Typed ECMA-SL.
$\boldsymbol{\Gamma}$	A store typing environment for Typed ECMA-SL.
$\boldsymbol{\mathcal{H}}$	A heap typing environment for Typed ECMA-SL.
$\boldsymbol{\Phi}$	The set of all possible statement outcomes.
$\boldsymbol{\gamma}$	A store from a Typed ECMA-SL program.
$\boldsymbol{\eta}$	A heap from a Typed ECMA-SL program.
$\boldsymbol{\phi}$	The outcome of a Typed ECMA-SL statement.

Acronyms

AST	Abstract Syntactic Tree
AOT	Ahead-of-Time
AO	Array Object
CESL	Core ECMA-SL
CTR	Conditional Type Refinement
DSL	Domain-Specific Language
ECMA	European Computer Manufacturers Association
ES	ECMAScript
ESL	ECMA-SL
FTS	Featherweight TypeScript
FO	Function Object
GO	Global Object
IR	Intermediate Representation
IML	Interpreter Main Loop
JS	JavaScript
TPB	Type Pattern Binding
TC39	Technical Committee 39
TESL	Typed ECMA-SL
TS	TypeScript

1

Introduction

Contents

ECMAScript, commonly known as JavaScript, currently stands as one of the most popular and widely used programming languages. It is the *de facto* language for client-side web applications, given that approximately 98.8% of all websites use it for their frontend development,[1] and it is supported by all major browsers. The popularity of JavaScript has also been increasing in the development of server-side applications (through Node.js[2]), and the language runs even on small embedded devices.[3]

To ensure a consistent behaviour of JavaScript programs across browsers, the ECMA International association develops and maintains the ECMAScript standard, a highly complex document written in English that describes both the syntax and semantics of JavaScript. This document is written as if it was the pseudocode of an ECMAScript interpreter, detailing the steps for the evaluation of every construct of the language. JavaScript is in a state of constant evolution, with a new (and often bigger) version of its specification being released annually. In fact, the language specification grew to over four times its original size over the past two decades. Figure 1.1 illustrates the evolution of the standard's size from its first widely addopted release in 1999 (ES3) to its latest version in 2023 (ES14).

[1] Usage statistics of JavaScript as a client-side programming language on websites, October 2023, W3Techs.com - https://w3techs.com/technologies/details/cp-javascript

[2] NodeJS is an open-source cross-platform server environment that runs off the V8 JavaScript engine and executes JavaScript code - https://nodejs.org/en/

[3] JerryScript, a JavaScript engine for the Internet of Things - https://jerryscript.net/

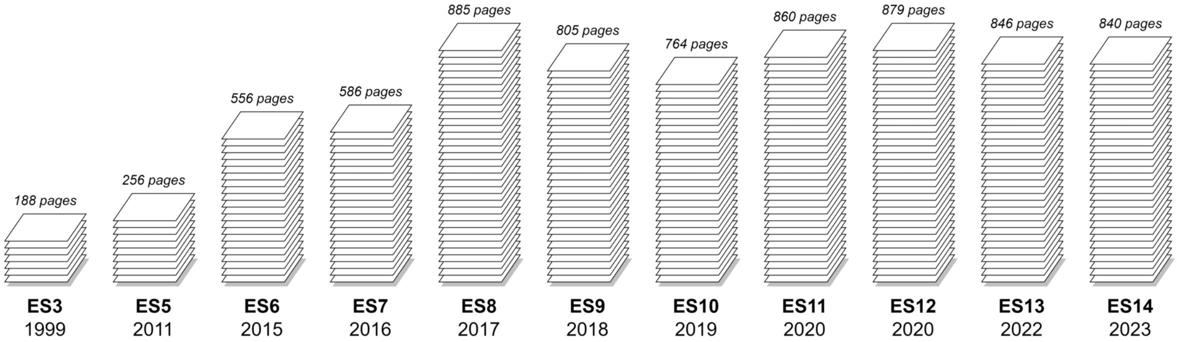


Figure 1.1: Evolution of the size of the ECMAScript standard (number of pages) from 1999 to 2013.

As the complexity of the language grows, the ECMAScript standard becomes increasingly difficult to manage and extend. When adding new features to the standard, it is extremely easy to introduce errors that compromise the internal invariants maintained by the specification and break the behaviour of other existing features. Furthermore, developing tests for these new features is particularly challenging, given the convoluted nature of the semantics of the language.

To ameliorate this situation, a team of researchers at IST created the ECMA-SL project[4] [1], which aims to provide a set of tools for specifying and analysing the JavaScript language. This project is centred around an intermediate language called ECMA-SL, designed to be as similar as possible to the meta-language used in the standard's pseudocode. As a result, ECMA-SL is significantly less complex than JavaScript. To put it into perspective, while the latest version of the JavaScript specification spans over 800 pages, the ECMA-SL specification can be described in a single one.

Using ECMA-SL, it is possible to create a reference interpreters for JavaScript that adhere to the standard's pseudocode line-by-line. Currently, the project includes two interpreters, ECMARef5 [1] and ECMARef6 [2], for ES5 [3] and ES6 [4], respectively. Both interpreters were thoroughly tested against Test262 [5], the official JavaScript test suite, being the most complete academic implementations of the standard for their respective versions. Besides being able to evaluate JavaScript programs according to the exact semantics of the standard, the reference interpreters offer a number of applications, including:

- *Proving invariants of the standard:* We can use formal methods on the implementation of the reference interpreters to prove certain properties and invariants of the standard, ensuring that it is free of errors and inconsistencies [6, 7, 8].
- *Generating tests suits for implementations of the language:* The interpreters can be used to automatically generate test cases for new features introduced in the language, particularly by making use of dynamic test generation techniques [9, 10, 11].

[4] Currently, the ECMA-SL project is under the responsibility and supervision of the authors of this thesis.

- *Generating documentation for the language:* We can generate from the reference interpreters an HTML version of the standard without any major differences from the official ECMAScript document in terms of structure and content [1].

Despite the aforementioned applications, it is still not possible to replace the official JavaScript standard with its ECMA-SL implementation. This is mainly due to the standard's rapid growth in both size and complexity, making it difficult to extend the interpreters to newer versions. Even though the standard is currently at the 14th version, released in 2023, the ECMA-SL project only supports up to the 6th. The challenges associated with extending the reference interpreters are related with the time-consuming nature of development in ECMA-SL. This is a direct result of ECMA-SL being an untyped language, leading to numerous bugs and hindering the refactoring process. The addition of types to the language will help to streamline code changes, providing greater assurance to developers that their code is bug-free.

Contributions Essentially, this project aims to address one of the biggest limitations of the ECMA-SL language, namely its lack of a static type system. We consider this thesis to have three main contributions to the overall ECMA-SL project, including:

- **Implementation of Typed ECMA-SL:** The central contribution of this thesis is the design and implementation of Typed ECMA-SL, an extension of the ECMA-SL language with type declarations and a static type system for type checking programs. Throughout the development process, we consistently prioritized the aspect of usability. In particular, we were very meticulous in our approach for reporting type errors, ensuring that developers can easily identify the source of the problems. Furthermore, we also provide mechanisms (e.g., syntax highlighting) to integrate the Typed ECMA-SL language with development environments like VSCode.

- **Formalization of Typed ECMA-SL:** The second contribution of this work is the formalization of a subset of Typed ECMA-SL and its type system. Additionally, we prove type safety for the formalized fragment, particularly the soundness of the type system with respect to the operational semantics of the language. Type safety helps in building confidence that, under Typed ECMA-SL, well-typed programs cannot go wrong.[5]

- **Implementation of a Simplified JavaScript Interpreter:** The third contribution is the implementation of a simplified interpreter for JavaScript, developed in parallel with Typed ECMA-SL. Besides being used to evaluate the type system of the language, this interpreter has many applications across the entire ECMA-SL project. For instance, it can be used as the foundation for certain types of static analysis for JavaScript that do not benefit from a precise language model.

[5] In reality, well-typed programs in Typed ECMA-SL can still cause runtime errors, as our formalization does not precisely model the type system implemented in the context of this thesis.

Evaluation The main goal of Typed ECMA-SL is to be used to fully type the ECMARef5 and ECMARef6 interpreters. However, this is a very complex and time-consuming task that extends beyond the scope of this thesis. Therefore, to evaluate the success of this project, we used Typed ECMA-SL to implement and fully type a simplified interpreter for the JavaScript language. This interpreter covers the most fundamental JavaScript constructs and is sufficiently complex so that, by typing it, we employ the majority of the type system features required to type the complete reference interpreters. As a result, typing the simplified interpreter gives us confidence that it will be possible to type the reference interpreters. Finally, we also attempted to identify syntactical issues within the reference interpreters by applying our type system to them without introducing type annotations.

1.1 Thesis Outline

This document is organized as follows: In Chapter 2, we described the core aspects of the ECMAScript standard, followed by an in-depth description of the ECMA-SL project, including its applications and limitations. In Chapter 3, we examine operational semantics and type systems that have been studied for the JavaScript language. In Chapter 4, we discuss the challenges and design choices associated with Typed ECMA-SL, and we introduce the formal model of the language, including its syntax, typing rules, operational semantics, and proof of soundness. In Chapter 5, we discuss the implementation of Typed ECMA-SL, including its architecture, additional features not included in the formal model of the language, and its error reporting mechanism. In Chapter 6, we evaluate the main outcomes of this thesis and assess the quality of the implemented type system. Finally, in Chapter 7, we draw some conclusions about our work and reflect on some future work directions.

2

Background

Contents

In this chapter, we provide the fundamental background required to understand the context and significance of our research. We begin with a brief analysis of the ECMAScript standard (Section 2.1). This standard constitutes the main subject of the ECMA-SL project (Section 2.2) within which our research is situated. We end this section by enumerating the main problems and limitations of the ECMA-SL project (Section 2.3), with a particular focus on the ones we are attempting to mitigate.

2.1 ECMAScript

The ECMAScript standard [12] is the official document that defines the ECMAScript (ES) scripting language, the official name for the programming language commonly known as JavaScript (JS). The standard is developed and maintained by the Technical Committee 39 (TC39), a committee formed by

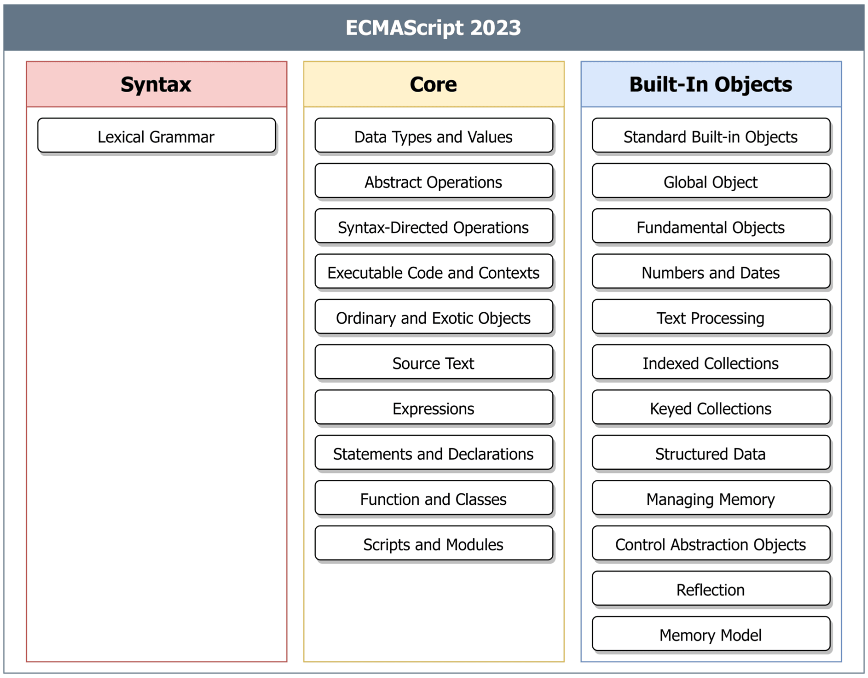


Figure 2.1: The internal division of the 14th version of the ECMAScript standard.

developers and academics under the European Computer Manufacturers Association (ECMA) organization, and it is currently on its 14th version (ES14), published in June 2023.

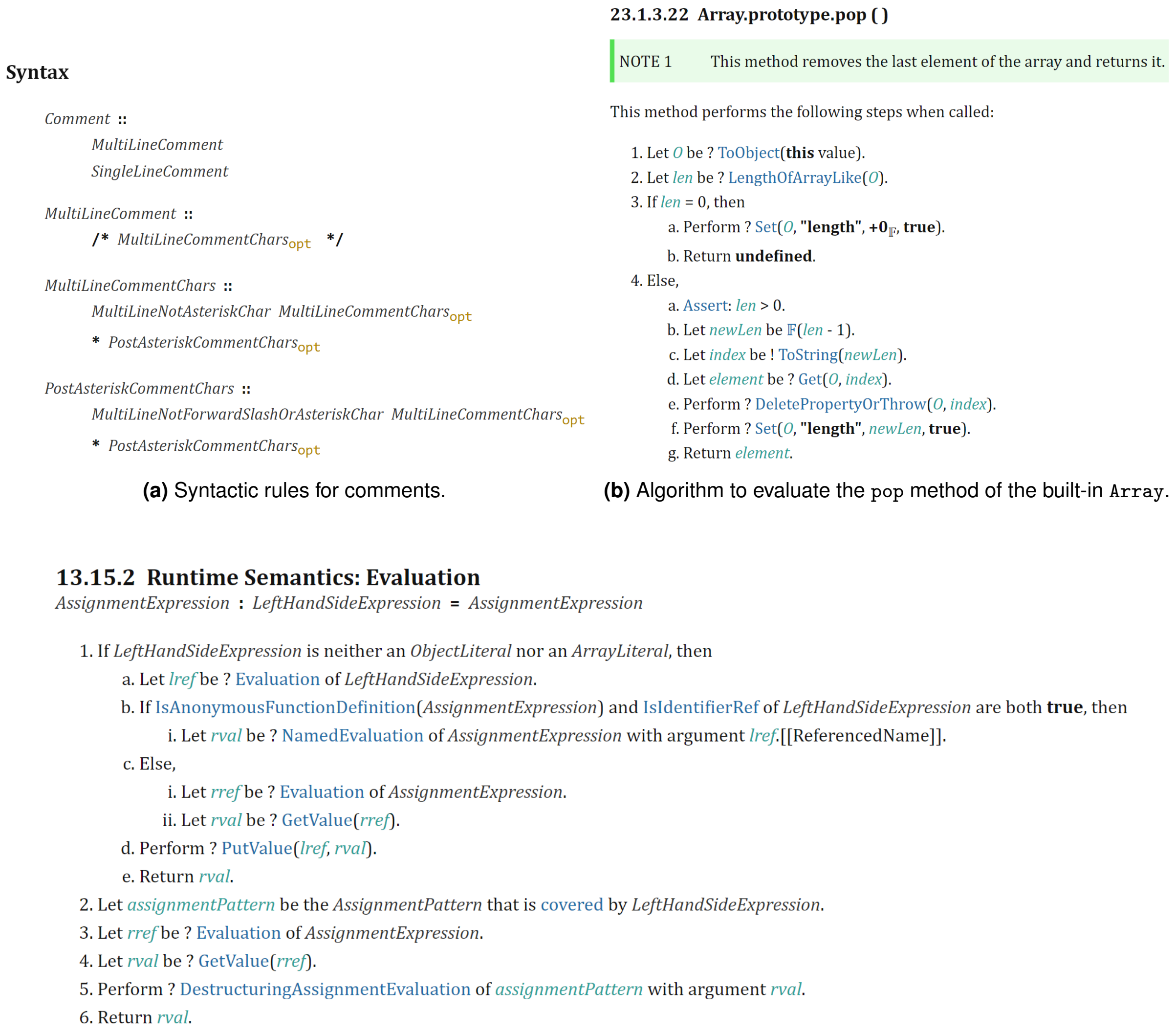
This ES specification fully outlines the syntax, semantics, and behaviour of the JavaScript language, including its data types, values, expressions, statements, functions, built-ins, and others. Furthermore, it is written as if it was the pseudocode of a JS interpreter, with every operation being described as a sequence of steps that must be executed to evaluate it. In the remaining section, we provide a high-level overview of the standard, with a particular focus on its data types (our primary research focus).

The specification of the language is currently divided into 23 sections. As described in Figure 2.1, these sections can be grouped into three different categories, namely:

- **Syntax:** Defines the grammatical rules for the language, including comments, tokens, names, keywords, and other building blocks. These rules are described in BNF notation. Figure 2.2a depicts a subset of the grammatical governing comments in JavaScript.

- **Core:** Defines a set of algorithms that are used to evaluate expressions, statements, and other building blocks, according to the semantics of the language. Figure 2.2c illustrates the evaluation rules for the assignment expression.

- **Built-In Objects:** Defines the interface and behaviour of the various built-in libraries available to developers, such as the `String`, `Array`, and `Date` built-ins. Figure 2.2b contains the algorithm for popping an element from an array (i.e., the `Array.prototype.pop()` function).



Syntax

Comment ::
MultiLineComment
SingleLineComment

MultiLineComment ::
`/*` *MultiLineCommentChars*$_{opt}$ `*/`

MultiLineCommentChars ::
MultiLineNotAsteriskChar *MultiLineCommentChars*$_{opt}$
`*` *PostAsteriskCommentChars*$_{opt}$

PostAsteriskCommentChars ::
MultiLineNotForwardSlashOrAsteriskChar *MultiLineCommentChars*$_{opt}$
`*` *PostAsteriskCommentChars*$_{opt}$

(a) Syntactic rules for comments.

23.1.3.22 Array.prototype.pop ( )

NOTE 1 This method removes the last element of the array and returns it.

This method performs the following steps when called:

```
1. Let O be ? ToObject(this value).
2. Let len be ? LengthOfArrayLike(O).
3. If len = 0, then
    a. Perform ? Set(O, "length", $+0_{\mathbb{F}}$, true).
    b. Return undefined.
4. Else,
    a. Assert: len > 0.
    b. Let newLen be $\mathbb{F}$(len - 1).
    c. Let index be ! ToString(newLen).
    d. Let element be ? Get(O, index).
    e. Perform ? DeletePropertyOrThrow(O, index).
    f. Perform ? Set(O, "length", newLen, true).
    g. Return element.
```

(b) Algorithm to evaluate the `pop` method of the built-in `Array`.

13.15.2 Runtime Semantics: Evaluation

AssignmentExpression : *LeftHandSideExpression* `=` *AssignmentExpression*

```
1. If LeftHandSideExpression is neither an ObjectLiteral nor an ArrayLiteral, then
    a. Let lref be ? Evaluation of LeftHandSideExpression.
    b. If IsAnonymousFunctionDefinition(AssignmentExpression) and IsIdentifierRef of LeftHandSideExpression are both true, then
        i. Let rval be ? NamedEvaluation of AssignmentExpression with argument lref.[[ReferencedName]].
    c. Else,
        i. Let rref be ? Evaluation of AssignmentExpression.
        ii. Let rval be ? GetValue(rref).
    d. Perform ? PutValue(lref, rval).
    e. Return rval.
2. Let assignmentPattern be the AssignmentPattern that is covered by LeftHandSideExpression.
3. Let rref be ? Evaluation of AssignmentExpression.
4. Let rval be ? GetValue(rref).
5. Perform ? DestructuringAssignmentEvaluation of assignmentPattern with argument rval.
6. Return rval.
```

(c) Evaluation rules for the assignment expression.

Figure 2.2: Excerpts of different sections of the ECMAScript standard.

The Data Types and Values section (Core) describes the language datatypes and values. The standard defines 7 built-in value types for the language: `number`, `string`, `boolean`, `symbol`, `null`, `undefined`, `bigint`,[1] and `object`. Arguably, objects are the most interesting datatype as they act as the foundation for most of the standard. For example, in JavaScript, functions are modelled as objects (referred to as function objects) that include a special method `call()` to allow them to be called.

Objects are simple collections of key-value pairs. Keys, also referred to as properties, are of type `string` or `symbol`. They are used to access the object's members, via the property accessor operator `obj.prop`. There are other ways of accessing object properties, such as the index operator `obj["prop"]` or object deconstructing `const { prop } = obj`.

[1] `bigint` - introduced in the 11th version of the standard (ES11), this type represents numeric values that are too large to be stored by the `number` primitive.

Evidently, the ES standard is an extremely complex specification. We will refrain from describing it extensively, as it is not required to have a complete grasp of it to understand our research project. Instead, we will describe further aspects of the standard as the need arises.

2.2 ECMA-SL Project

The ECMA-SL project was established with the goal of building an executable specification for the ES standard. This specification takes the form of a reference interpreter called ECMARef, written to adhere the standard's pseudocode line-by-line. Consequently, the interpreter can execute a JS program exactly as it is described by the standard, producing the expected output and side effects.

In this section, we provide an in-depth description of the ECMA-SL project. In Section 2.2.1, we outline the rationale behind the project, enumerating the purposes and advantages of having an executable specification over the official textual standard. In Section 2.2.2, we analyse the intermediate language employed in the development of the ECMARef interpreter. Lastly, in Section 2.2.3, we explain how we can use the reference interpreters to execute a JS program according to the rules of the standard.

2.2.1 ECMA-SL Rationale

In addition to being easier to maintain, there are numerous advantages in having an executable specification of JavaScript instead of a textual description of its syntax and semantics. Figure 2.3 illustrates the multiple ways in which the reference interpreters can be employed.

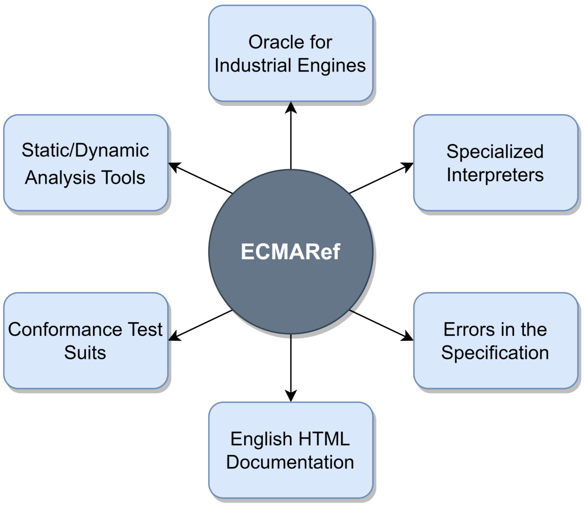

Oracle for Industrial Engines
Static/Dynamic Analysis Tools
Specialized Interpreters
ECMARef
Conformance Test Suits
Errors in the Specification
English HTML Documentation

Figure 2.3: Applications of the ECMARef interpreter.

- **Oracle for Industrial Engines:** The interpreters can function as reference points for industrial JavaScript engines [8, 13]. Real JS engines often incorporate optimizations that could lead to behaviours that deviate from the language specification, potentially resulting in bugs and vulnerabilities. By assessing these engines against the ECMARef interpreter, developers can achieve a higher level of confidence in the quality of their engine.

- **Static/Dynamic Analysis Tools:** The interpreters can be employed in the creation of static and dynamic analysis tools for JS, such as multi-level debuggers [14, 15] and symbolic execution tools [16, 17]. Such tools can be used, for example, to improve code quality, enhance security, optimize performance, and reason about code behaviour in complex and dynamic environments.
- **Specialized Interpreters:** The interpreters may be tweaked to derive a slightly different specification for JavaScript tailored to specific requirements. For instance, one could extract an interpreter for the language without implicit type coercions.
- **Conformance Test Suits:** The interpreters can facilitate the generation of tests for new features of the language and assess the coverage of existing test suits [6, 18]. This may be achieved with dynamic test generation techniques [9, 10, 11], that work by generating symbolic inputs to explore different execution paths of the interpreter.
- **Errors in the Specification:** The interpreters aids in detecting inconsistencies within the ES standard [6, 7] and proving its meta-properties [8]. For instance, we can use formal methods to prove that it is never possible for a JS variable to reference a declarative environment record; that is a JS variable can never point to the internal object used to represent function scopes.
- **English HTML Documentation:** The interpreters can also be used to generate the textual version of the standard [1]. This version can be obtained with a tool that generates faithful HTML code, with no significant differences from the text of the real standard. Additionally, since this version is automatically generated, it has the benefit of being more consistent than the standard, as similar concepts are always described in the same way.

2.2.2 ECMA-SL Language

The ECMARef interpreter is implemented using an intermediate language for JavaScript known as the ECMA Specification Language (ECMA-SL) [1]. The language is designed to be as close as possible to the meta-language of the standard, exclusively containing the meta-constructs required to model its pseudocode. This is essential to guarantee that the interpreter is written in the most faithful way possible.

ECMA-SL (ESL) features extensible objects and dynamic behaviour, including: (1) dynamic function calls, (2) dynamic creation and deletion of object properties, and (3) dynamic code evaluation. However, the language lacks implicit JS behaviours such as prototype-based inheritance and implicit type coercions. Regarding its structure, ECMA-SL contains three main syntactic categories, namely: Values, Expressions, and Statements. In the following, we describe each of these categories.

```
⟨expr⟩ ::= ⟨simple-expr⟩ | ⟨operator-expr⟩ | ⟨object-expr⟩ | ⟨call-expr⟩

⟨simple-expr⟩ ::= ⟨value⟩
  | ⟨var⟩
  | ’|’ ⟨var⟩ ’|’

⟨operator-expr⟩ ::= ⊖ ⟨expr⟩
  | ⟨expr⟩ ⊕ ⟨expr⟩
  | ⊗ (⟨expr⟩, ..., ⟨expr⟩)

⟨object-expr⟩ ::= ’{’ ⟨prop⟩ ’:’ ⟨expr⟩, ..., ⟨prop⟩ ’:’ ⟨expr⟩ ’}’
  | ⟨expr⟩ . ⟨prop⟩
  | ⟨expr⟩ ’in’ ⟨expr⟩

⟨call-expr⟩ ::= ⟨expr⟩ ’(’ ⟨expr⟩, ..., ⟨expr⟩ ’)’
  | ⟨expr⟩ ’(’ ⟨expr⟩, ..., ⟨expr⟩ ’) catch ’ ⟨fname⟩
```

Figure 2.4: ECMA-SL Grammar - Expressions. The non-terminals ⟨value⟩, ⟨var⟩, ⟨prop⟩, and ⟨fname⟩ range over values, variable names, property names, and function names, respectively.

Values Values in ECMA-SL closely resemble those found in JavaScript. They include numbers (divided into integers and floats), booleans, strings, symbols, object locations, the special `null` and `void` values, as well as value lists and tuples. Lists differ from tuples in that they can be dynamically extended and shrunk during execution, while tuples maintain a constant size.

Expressions Figure 2.4 describes ECMA-SL expressions. They can be divided into four categories:

- **Simple expressions:** Includes value expressions and local/global variables accesses.
- **Operator expressions:** Includes unary, binary, and n-ary built-in operators.
- **Object expressions:** Includes object literals, property lookups, and the `in` operator used to check if a property name exists within an object. Lookups can be categorized as either static, where names are determined during compile time, or dynamic, which delay their evaluation until runtime.
- **Call expressions:** Includes function calls with and without a `catch` clause. This clause is used to specify an error handler that executes when the function throws an error.

Statements Figure 2.5 describes the grammar of the ECMA-SL statements. In the language, statements can be grouped into five different categories:

- **Simple statements:** Includes blocks, prints, and local/global variable assignments.
- **Object statements:** Includes static and dynamic property assignments, and property deletion. In static property assignments, property names are known at static time. Contrarily, dynamic property assignments evaluate property names at runtime.

```
⟨stmt⟩ ::= ⟨simple-stmt⟩ | ⟨object-stmt⟩ | ⟨conditional-stmt⟩ | ⟨loop-stmt⟩ | ⟨ret-err-stmt⟩ |

⟨simple-stmt⟩ ::= '{' ⟨stmt⟩ ';' ... ';' ⟨stmt⟩ '}'
  | 'print' ⟨expr⟩
  | ⟨var⟩ ':=' ⟨expr⟩
  | '|' ⟨var⟩ '| :=' ⟨expr⟩

⟨object-stmt⟩ ::= ⟨expr⟩ '.' ⟨var⟩ ':=' ⟨expr⟩
  | ⟨expr⟩ '[' ⟨expr⟩ ']' ':=' ⟨expr⟩
  | 'delete' ⟨expr⟩ '[' ⟨expr⟩ ']'

⟨conditional-stmt⟩ ::= 'if (' ⟨expr⟩ ') {' ⟨stmt⟩ '} elif (' ⟨expr⟩ ') '{' ⟨stmt⟩ '}' ... 'else {' ⟨stmt⟩ '}'
  | 'switch (' ⟨expr⟩ ') { case' ⟨expr⟩ ':' ⟨stmt⟩ ... 'default:' ⟨stmt⟩ '}'
  | 'match' ⟨expr⟩ 'with |' ⟨pattern⟩ '->' ⟨stmt⟩ '|' ... '| default ->' ⟨stmt⟩

⟨loop-stmt⟩ ::= 'while (' ⟨expr⟩ ') {' ⟨stmt⟩ '}'
  | 'repeat {' ⟨stmt⟩ '} until' ⟨expr⟩
  | 'foreach (' ⟨var⟩ ':' ⟨expr⟩ ') {' ⟨stmt⟩ '}'

⟨ret-error-stmt⟩ ::= 'return' ⟨expr⟩
  | 'throw' ⟨expr⟩
  | 'fail' ⟨expr⟩

⟨pattern⟩ ::= '{' ⟨pattern-pair⟩, ..., ⟨pattern-pair⟩ '}'

⟨pattern-pair⟩ ::= ⟨prop⟩ ':' ⟨value⟩
  | ⟨prop⟩ ':' ⟨var⟩
  | ⟨prop⟩ ': None'
```

Figure 2.5: ECMA-SL Grammar - Statements. The non-terminals ⟨value⟩, ⟨var⟩ and ⟨prop⟩ range over values, variable names and property names, respectively.

- **Conditional statements:** Includes the `if-then-else`, `switch`, and the `match-with` statements. The latter is similar to a `switch` statement, but it is used to match an object against a pattern. For example, to successfully match an object against the `{ foo: x, bar: "literal", baz: None }` pattern, the object must have an arbitrary `foo` property (that will be assigned to variable `x`), a `bar` property with the value `"literal"`, and it must not contain the `baz` property. Furthermore, note that the object may contain other properties not explicitly declared within the pattern.

- **Loop statements:** Includes the `while`, `repeat` and `foreach` statements. The `repeat` statement behaves similarly to a `while`, but with its conditional test being done at the end of the looped block. On the other hand, `foreach` statements are used to iterate through a list of elements.

- **Return and error statements:** Includes the `return`, `throw`, and `fail` statements. The difference between a `throw` and a `fail` is that, while the former returns the execution to the closest error handler, the latter simply terminates the program with the error message specified.

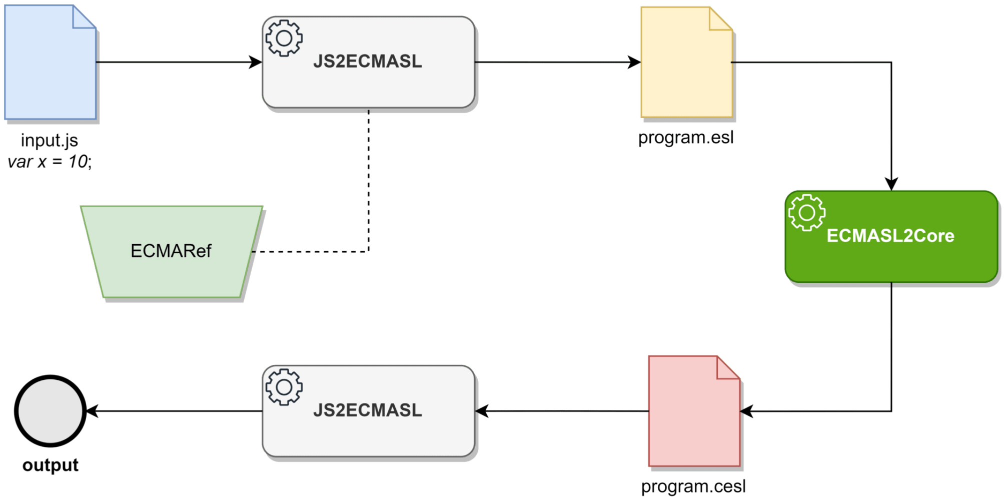


Figure 2.6: Execution pipeline of the ECMA-SL project.

2.2.3 ECMA-SL Execution Pipeline

Figure 2.6 illustrates the pipeline for executing a JavaScript program using the ECMARef interpreter. Besides the code of the ECMARef interpreter itself, the pipeline it composed of three main components:

1. JS2ECMASL: Models the JavaScript program in the ECMA-SL language.
2. ECMASL2Core: Compiles the ECMA-SL program to Core ECMA-SL. Core ECMA-SL (CESL) consists of a simplified version of the ECMA-SL language, and it is described below.
3. ECMA-SL Interpreter: Interprets a Core ECMA-SL program.

Consider a `input.js` file, containing the program with a single variable initialization `var x = 10`. In the following, we analyse each of the components by the order in which they appear within the execution pipeline. We refer to this example to illustrate how they work.

JS2ECMASL The JS2ECMASL tool is responsible for modelling a JavaScript program in ECMA-SL. Consider the running example described above. Listing 2.1 depicts the process of modelling this JS program, which consists of generating an ECMA-SL representation of the program (line 18) and then evaluating it using the standard-compliant JS interpreter (line 19).

Representing the JavaScript input in ECMA-SL is accomplished in two steps. First, JS2ECMASL resorts to Esprima [19], a standard-compliant JavaScript parser written in JavaScript, to generate the Abstract Syntactic Tree (AST) of the input program. Then, JS2ECMASL creates the ESL function `Build_AST()` that reconstructs the program's AST in the memory of ECMA-SL.

```
function Build_AST() {
  __n0 := {  };
  __n0["type"] := "Program";
  ...
  __n2 := {  };
  __n2["type"] := "VariableDeclaration";
  ...
  __n1 := [ __n2 ];
  __n0["body"] := __n1;
  return __n0
}
```

```
function JS_Interpreter_Program(ast) {
  ...
}

function main() {
  ast := Build_AST();
  retval := JS_Interpreter_Program(ast);
  return retval
}
```

Listing 2.1: Compilation of a JavaScript program into ECMA-SL.

ECMASL2Core After generating the ECMA-SL program with the JS2ECMASL tool, we need to execute it. However, programs written in ECMA-SL are not directly interpreted as they still contain some constructs that can be further simplified. Instead, they are compiled to a simpler language, referred to as Core ECMA-SL. For example, the `repeat` and `foreach` loops can both be expressed as `while` statements, while the `match-with` construct can be compiled to a sequence of `if-then-else` statements.

```
1 match s with
2 | { type: "Program", b: body } -> {
3
4 }
```

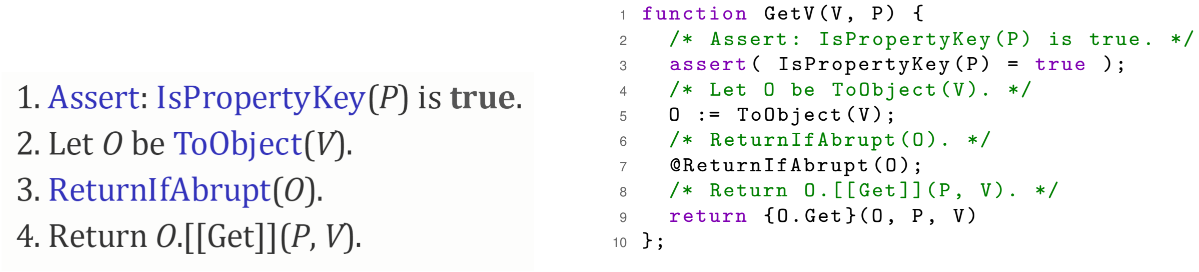
```
1 _v1 := (s["type"] = "Program");
2 _v2 := ("body" in s);
3 if (true && _v1 && _v2) {
4     b := s["body"];
5 }
```

Listing 2.2: Compilation of a `match-with` statement into an `if-then-else` statement.

Listing 2.2 illustrates the compilation of a `match-with` statement (left snippet), written in ECMA-SL, into its equivalent `if-then-else` counterpart (right snippet), in Core ECMA-SL. In the left snippet, we match variable `s` against a pattern with two properties: `type`, expected to have the value `"Program"`, and `body`, which must exist so that it can be assigned to variable `b`. Similarly, on the right code snippet, there are two preconditions to enter the `if` statement. First, `s["type"]` needs to evaluate to `"Program"` (variable `_v1`, in line 1). Second the `body` property must exist (variable `_v2`, in line 2), so that it can be assigned to variable `b` within the body of the `if` statement (line 4).

Core ECMA-SL differs from ECMA-SL in several aspects. Firstly, expressions can only interact with the variable store. Additionally, the language does not support global variables, and only features a single conditional statement (`if-then-else`) and a single loop statement (`while`). Finally, Core ECMA-SL does not incorporate any error handling mechanisms.

The ECMA-SL compiler is also responsible for resolving all imports within ESL files. This means that the resulting program (`"program.cesl"`, according to Figure 2.6) is self-contained, including both the code that will be interpreted as well as the code of the interpreter itself. This makes it easier to execute the program, as all necessary code exists in the same file.



```
1. Assert: IsPropertyKey(P) is true.
2. Let O be ToObject(V).
3. ReturnIfAbrupt(O).
4. Return O.[[Get]](P, V).
```

(a) The ECMAScript pseudocode.

```
1  function GetV(V, P) {
2    /* Assert: IsPropertyKey(P) is true. */
3    assert( IsPropertyKey(P) = true );
4    /* Let O be ToObject(V). */
5    O := ToObject(V);
6    /* ReturnIfAbrupt(O). */
7    @ReturnIfAbrupt(O);
8    /* Return O.[[Get]](P, V). */
9    return {O.Get}(O, P, V)
10 };
```

(b) The ECMARef implementation.

Figure 2.7: Comparison between the ECMAScript's pseudocode for the `GetV(V, P)` function and its implementation in the ECMARef6 interpreter.

ECMA-SL Interpreter After constructing the final Core ECMA-SL file, the ECMA-SL interpreter is used to execute the program. The interpreter is written in OCaml [20], an industrial programming language focused on expressiveness and safety. It features two modes of operation: (1) silent mode, which only shows the final output of the program; and (2) verbose mode, which displays all trace information generated during the program's execution.

ECMARef As previously mentioned, the ECMARef interpreter is specifically designed to adhere, as much as possible, to the text of the ES standard. Moreover, the interpreter supports all built-in objects of the standard, and it is tested against Test262 [5], the official JavaScript conformance test suite.

To show the similarities between the reference interpreter and the pseudocode of the ES specification, consider the example of Figure 2.7. This example presents a side-by-side comparison between the pseudocode of the `GetV(V, P)` function,[2] responsible for retrieving the value of a specific property within a JS object, and its corresponding implementation in the reference interpreter. By analysing the figure, it becomes clear that the interpreter meticulously adheres to the standard, following it line-by-line. Each instruction in the interpreter is preceded by a comment that references the corresponding step in the standard. Take, for instance, line 2 of the standard (`Let O be ToObject(V)`), which is identical to line 5 of the interpreter (`O := ToObject(V)`), albeit expressed in ECMA-SL instead of plain text.

2.3 ECMA-SL Limitations

In Section 2.2.1, we discussed the main advantages of the ECMARef interpreter over the official ES standard. Given these advantages, one might question if it is possible to seamlessly replace the official standard with its ESL implementation. Unfortunately, the rapid growth in size and complexity of the standard makes it difficult to keep the implementation up to date with the newer versions of the JS specification. In fact, the last ES version supported by the ECMA-SL project is the 6th (ES6), released in 2015.

[2] `GetV(V, P)` function - `https://262.ecma-international.org/6.0/#sec-getv`

There are many reasons why extending the ECMARef interpreters is such a laborious task. Firstly, the standard itself is a complex specification. The main problem, however, lies in the difficulty of specifying the standard using ECMA-SL, due to several inherent limitations of the language, such as:

- **No Separate Compilation:** Separate compilation is a technique that allows developers to divide large projects into smaller, more manageable pieces called modules. Modules are compiled independently and subsequently linked together to create the final program. Alas, the ECMA-SL compiler lacks these encapsulation mechanisms. One immediate consequence is that ECMA-SL developers must manually guarantee that function names are unique across all program files.

- **No Easy Syntactic Checks:** There are several easy syntactic checks that the ECMA-SL compiler does not perform. These checks are valuable in helping developers identify bugs that might otherwise be hidden until execution, where their detection becomes more challenging. They include but are not limited to: (1) detecting naming conflicts, including duplicated function names, duplicated function parameters, and duplicated object properties; (2) detecting calls to undefined functions or accesses to unknown variables; and (3) detecting control paths that do not terminate with a `return` or `throw` instruction, which is a mandatory requirement in the language.

- **No Static Typing:** Similarly to JS, ECMA-SL is an untyped language. Untyped languages have several disadvantages compared to statically typed languages: (1) they pose a greater challenge in the detection of type-related bugs, as these issues only become apparent during runtime; and (2) maintaining an untyped program is significantly more difficult, mainly due to readability issues stemming from the lack of explicit type annotations. Conversely, typed languages promote a design-by-contract approach, where function signatures serve as a clear interface for their usage.

With our research, we aim to address the last two limitations. However, there are some challenges associated with this endeavour. While extending the ECMA-SL compiler with syntactic checks is a relatively straightforward task, the ECMA-SL language is a hard target for standard type systems, due to being a highly dynamic language. Multiple research projects have attempted to solve the issues associated with statically typing dynamic languages. In the following, we will be analysing some of these systems for JavaScript-like languages and the specific problems they can effectively address.

3

Related Work

Contents

The research literature covers a large variety of analysis techniques for the JavaScript language. They include, among others, type systems [21,22,23], abstract interpreters [24,25], point-to analysis [26, 27], program logics [28, 29], operational semantics [8, 13, 30, 31], and information flow [32, 33, 34, 35].

In this chapter, we focus on the most important operational semantics suggested for JavaScript (Section 3.1) as well as the key type systems proposed for the language (Section 3.2). Even though we plan to type ECMA-SL, both languages are very similar in that ECMA-SL was designed to be as close as possible to the meta-language used in the JavaScript specification. Consequently, the features and challenges associated with typing ECMA-SL are identical to those of JavaScript.

3.1 Operational Semantics for JavaScript

The complexity of JavaScript has led to the development of multiple academic reference implementations for the language. While these implementations may differ in their development methodologies, supported versions, and evaluation methods, they have been gradually converging to a better alignment with the

ES standard. They serve multiple purposes, such as acting as oracles for assessing the behaviour of industrial JS engines [8, 13], facilitating the design of static analysis tools [14, 15, 16, 17], and detecting problems within the standard [6, 7], as well as proving its meta-properties. Section 2.2.1 provides a comprehensive description of the rationale behind the ECMA-SL project by elaborating on the general purposes behind reference implementations and introducing others that are specific to the project itself.

Reference interpreters thus play an increasingly significant role in the management and engineering of the ES standard. One notable example is the JISET interpreter [6], which served as the foundation for tools such as JEST [18] for concurrent testing and JSTAR [7] for detecting type errors within the standard. In particular, JSTAR has recently been integrated into the tooling infrastructure of the official ECMAScript repository. However, all proposed reference implementations have significant limitations, including: (1) limited coverage of the standard; (2) lack of modifiability, which makes it challenging to extend them to newer versions of the standard; and (3) lack of efficiency, which can render it difficult to use these implementations as the foundation for program analysis.

Reference Interpreter	ES Version	Implementation Language	EXE	LBL	#Passed Tests	#Total Tests	Success Rate
S5 [36]	5	S5 Core Language	✓		8157	12074	67.56%
JSRef [8]	5	Coq + OCaml	✓	✓	3749	12074	31.05%
KJS [13]	5	K Framework	✓		2782	12074	23.04%
JSExplain [14]	5	OCaml (subset)	✓	✓	>5000	12074	41.41%
ECMARef5 [1]	5	ECMA-SL	✓	✓	12026	12074	99.60%
ECMARef6 [2]	6	ECMA-SL	✓	✓	19009	21662	87.75%
JISET [6]	10	IR_{ES}	✓	✓	16355	35990	45.44%

Table 3.1: The most significant reference implementations for JavaScript. The EXE column identifies executable implementations, and the LBL column identifies that an implementation followed the standard line-by-line. These implementations were tested against the Test262 conformance test suite [5].

Table 3.1 summarizes the key reference implementations of JavaScript. As the table shows, recent implementations tend to closely adhere to the standard line-by-line. This approach has proven to be effective and widely accepted in establishing trust in reference implementations. Additionally, they often define their own Domain-Specific Languages (DSLs) specifically tailored to the standard's specification, instead of relying on general-purpose programming specification languages. Lastly, the table highlights that the ESL reference interpreters, namely ECMARef5 and ECMARef6, are currently the most complete academic reference implementations of the standard. They respectively pass 99.60% and 87.75% of the tests from the Test262 conformance test suite [5], which is a considerably higher success rate than that achieved by all other implementations. In the following, we briefly describe these implementations as well as some other important JS formalizations. They are presented in chronological order.

Maffeis et al. (2008) [30] were the first to develop an operational semantics for JavaScript, specifically targeting the third version of its standard (ES3). These semantics accurately modelled the behaviour of JavaScript according to the standard, and served as a foundational tool for analysing security aspects of the language in web applications and mashups [37, 38]. Despite the complexity of their work, the authors presented their semantics in a non-mechanized form, conveyed through an extensive textual document with a large number of semantic rules written in their own custom-made language.

Guha et al. (2010) [31] introduced λJS, a core lambda calculus that captures essential features of ES3, such as extensible objects, prototype-based inheritance, and dynamic function calls, while excluding the `eval` and certain built-in libraries. The system works by translating ES3 programs into simple λJS expressions, which are subsequently evaluated by an interpreter written in Racket. Furthermore, it incorporates a type system for checking a simple confinement property of λJS programs. The authors verified the correctness of their semantics using the Mozilla JS test suite, ensuring consistency with mainstream implementations like SpiderMonkey,[1] V8,[2] and Rhino.[3]

Politz et al. (2012) [36] extended λJS from the third version of the standard (ES3) to the fifth (ES5). This reference implementation, referred to as S5, introduced the semantics for accessors (getters and setters) and the eval operator. The project included a de-sugaring translation from ES5 programs to the S5 core language and a reference interpreter for S5 written in Racket. S5 supports about 60% of the ES5 standard library objects and was tested against Test262, passing about 70% of its tests. Many of the unsuccessful tests derived from challenges in implementing certain built-in objects and issues with non-strict code, suggesting some inconsistencies with the ES standard.

Bodin et al. (2014) [8] introduced JSCert, the first mechanized specification of ES semantics for the fifth version of the standard (ES5). This formalization was achieved using the Coq proof assistant, and aimed to closely align with the ES standard by mapping its every line into JSCert rules. Additionally, the authors developed JSRef, a reference interpreter for JavaScript also written in Coq and subsequently transformed into executable OCaml code for testing against Test262. JSCert proved effective in detecting bugs within the JavaScript standard, in its official conformance test suite and browser implementations. To assess the level of trust in the JSCert specification, its formal rules were placed side-by-side with the text of the standard, and their closeness ”eyeballed”. The ECMA-SL project follows a refined version of this approach, where we instead quantify the similarities between the official standard and an HTML version of our reference interpreter.

Park et al. (2015) [13] introduced KJS, a robust formalization of ECMAScript5 using the K framework.[4] KJS aimed to provide a comprehensive definition of the language’s syntax and semantics, generating parsers, interpreters, and formal analysis tools. It was able to pass all core language tests from

[1] SpiderMonkey - https://spidermonkey.dev/

[2] V8 - https://v8.dev/

[3] Rhino - https://mozilla.github.io/rhino/

[4] K Framework - https://kframework.org/

Test262, which is challenging even for industrial JS engines. While the authors fully defined the core semantics of the language, they considered its built-in libraries to be outside the project's scope.

Charguéraud et al. (2018) [14] proposed a reference interpreter for JS named JSExplain, designed to closely adhere to the language specification. The interpreter was developed in a purely functional style using a subset of OCaml specifically designed for that purpose, and it successfully passed over 5000 tests from the Test262 test suite. The main purpose of JSExplain was to assist developers in debugging JS code. This was done by enabling step-by-step execution of JS programs, whilst allowing developers to simultaneously analyse the state of the program and the internal state of the interpreter.

Park et al. (2020) [6] introduced JISET, an Intermediate Representation (IR)-based toolchain for extracting JavaScript semantics from its HTML specification. This toolchain automatically generates parsers and AST-IR translators from the JS specification, allowing for a partial implementation of the standard. Note that the extracted interpreter does not work out-of-the-box. The authors reported that they needed to manually fix approximately 5% of the code generated. The JISET execution engine was tested against Test262, passing 18,064 core language tests out of 35,990 applicable tests. The low success rate derives from the fact that the interpreter does not support most of the language built-in libraries, such as the RegExp, JSON, and String, mainly because of their inherent implementation complexity. One of the main advantages of the ECMA-SL project over JISET is the comprehensive coverage of JS built-in functions, which results in a significantly higher test success rate.

3.2 Type Systems for JavaScript

Thiemann (2005) [21] proposed the first type system for a subset of the JavaScript language. While this system considered some dynamic aspects of JS, such as extensible objects and dynamic function calls, it failed to account for other important features like JavaScript's prototype-based inheritance mechanism and implicit type coercions. The soundness of the system is achieved by tracking all type conversions and flagging those that could result in dangerous or unexpected behaviour. Type convertibility can be adjusted to be more or less conservative, depending on the needs of the analysis.

Anderson et al. (2005) [22] proposed JS_0, a flow-sensitive type system for a subset of JavaScript focused on runtime mutability for objects and methods. The system allows objects to evolve in a controlled manner by labelling their fields based on a notion of their definite presence or potential absence. Another improvement over Thiemann's work is the support for recursive types. Additionally, the authors defined a type inference algorithm capable of using well-formed constraints to annotate an untyped JS_0 program, and they also included a sound proof for their algorithm.

Jensen et al. (2009) [39] presented the first type system capable of inferring detailed and sound type information for a JavaScript program. The system uses abstract interpretation and it is designed to support the entire language defined in the ES standard, including its implicit behaviours and all built-in libraries. Furthermore, the authors describe how to reason about unary and binary operations with implicit type coercions by utilizing a complex lattice of safety types.

TypeScript (TS) (2012) [40] is a strict syntactical superset of JavaScript that adds optional static typing to the language. It is designed to be compiled down to plain JavaScript, so it can run in any JS environment. To minimize adoption costs, the language uses gradual typing, which allows parts of the system to be statically checked while others are left unchecked using the special type `any`. This is one of the many reasons that make TypeScript unsound by design.

Biermann et al. (2014) [41] attempted to capture the essence of TypeScript by providing a precise definition of the system on a core subset of the language. Their mathematical formalization helped to clarify ambiguities in the language documentation, leading to the discovery of inconsistencies and errors in both the language specification and its compiler. Furthermore, this definition allowed for a distinction between the sound and unsound aspects of TypeScript.

Choi et al. (2015) [42] proposed SJS, a static type system for a significant subset of JavaScript. Their goal was to allow efficient Ahead-of-Time (AOT) compilation of JS programs by ensuring that objects have a known layout at allocation time. One of the advantages of a fixed object layout is that it allows the compiler to translate attribute accesses into direct memory accesses. However, ensuring a fixed object layout while supporting prototype-based inheritance, structural subtyping, and method updates has significant challenges, primarily due to the JS semantics regarding attribute updates.

Chandra et al. (2016) [43] expanded the previous work to contemplate abstract and recursive types, as well as first-class methods. Furthermore, the authors developed a sound type inference algorithm for the formulated problem. The algorithm employs a combination of lower and upper bound propagation to infer types and discover type errors in the entire program. The system supports additional features, such as polymorphic arrays, operator overloading, and intersection types.

Language Features	TS1	TS2	TS3	TS4	TS5	TS6
Dynamic Function Calls	✓	✓	✓	✓	✓	✓
Extensible Objects	✓	✓	✓	✓		
Implicit Type Coercions		✓	✓	✓	✓	✓
Prototype-Based Inheritance			✓	✓	✓	✓

Table 3.2: Language features supported by type systems for JavaScript. TS1: Thiemann [21]; TS2: Anderson et al. [22]; TS3: Jensen et al. [39]; TS4: Bierman et al. [41]; TS5: Choi et al. [42]; TS6: Chandra et al. [43]

Table 3.2 identifies the key language features of JavaScript and shows which of the previously proposed systems can handle them. As we can see, most of the systems support the most important JS features. In particular, dynamic function calls, a fundamental concept in JavaScript, are supported by all of them. Additionally, both Thiemann [21] and Anderson et at. [22] deliberately avoided prototype-based inheritance, which is a complex mechanism to model. Finally, the proposals of Choi et al. [42] and Chandra et al. [43] revolve around a fixed object layout, thus not offering support for extensible objects.

Type System Features	TS1	TS2	TS3	TS4	TS5	TS6
First Class Methods	✓	✓	✓	✓		✓
Subtyping	✓	✓	✓	✓	✓	✓
Type Inference		✓	✓	✓	✓	✓
Type Recursion		✓		✓		✓
Flow Sensitivity		✓	✓	✓		
Union Types				✓		
Parametric Polymorphism				✓		

Table 3.3: Features of the type systems for JavaScript. TS1: Thiemann [21]; TS2: Anderson et al. [22]; TS3: Jensen et al. [39]; TS4: Bierman et al. [41]; TS5: Choi et al. [42]; TS6: Chandra et al. [43]

Table 3.3 enumerates the type system features that are particularly relevant to our project and shows which of the previous type systems include them. First-class methods and subtyping are incorporated by most of the systems, reflecting their significance within the JavaScript language. Additionally, except for Thiemann's proposal [21], all type systems contain some level of type inference, with the system proposed bsssy Anderson et al. [22] being entirely focused on it. Features like type recursion and flow sensitivity start to introduce problems, making it difficult to ensure soundness and thus being supported in fewer type systems. Bierman et al. [41] adopted a different approach by intentionally developing an unsound variant of the system to include all the features enumerated above.

Evaluation	TS1	TS2	TS3	TS4	TS5	TS6
Implemented?		✓	✓	✓	✓	✓
Benchmarked?			✓	✓	✓	✓
Sound?	✓	✓	✓		✓	✓

Table 3.4: Evaluation of the type systems for JavaScript. TS1: Thiemann [21]; TS2: Anderson et al. [22]; TS3: Jensen et al. [39]; TS4: Bierman et al. [41]; TS5: Choi et al. [42]; TS6: Chandra et al. [43]

Table 3.4 describes the evaluation of the previous systems. Except from Bierman et al. [41], all systems were designed with soundness as a requirement. Additionally, most of these systems were also implemented and benchmarked. Jensen et al. [39] used Google V8 benchmark suite,[5] as well as the four most complete SunSpider benchmarks.[6] Choi et al. [42] and Chandra et al. [43] made use of the Octane benchmark suite.[7] There have also been studies to compare the performance of TS and JS.[8]

[5] V8 benchmark suite - https://pages.cpsc.ucalgary.ca/~crwth/js/v8/benchmarks/run.html

[6] SunSpider - http://proofcafe.org/jsx-bench/js/sunspider.html

[7] Octane benchmark suite - https://developers.google.com/octane/

[8] Typescript VS Javascript (2023) - https://programming-language-benchmarks.vercel.app/typescript-vs-javascript

3.2.1 Most Relevant Type Systems

In this section, we analyse the three type systems that are most relevant to our work. To illustrate them, we will refer to the following example: consider real numbers in a two-dimensional plane ($\mathbb{R}_2$) and the method `addR2(n)` that receives the $\mathbb{R}_2$ number `n` and adds it to the $\mathbb{R}_2$ caller object. Listing 3.1 encodes this description in the JavaScript language.

```javascript
function R2(x, y) {
  this.x = x;
  this.y = y;
  this.add = addR2;
  return this;
}
```

```javascript
function addR2(n) {
  this.x = this.x + n.x;
  this.y = this.y + n.y;
  return this;
}
```

```javascript
let n1 = new R2(1, 2);
let n2 = new R2(2, 3);
n1.add(n2);
```

Listing 3.1: Real numbers in a two-dimensional plane ($\mathbb{R}_2$) defined in JavaScript.

The type systems detailed in this section are the proposals of Thiemann [21], Anderson et al. [22], and Bierman et al. [41]. For each system, we begin with a brief overview, followed by a more comprehensive description of its syntax. After that, we encode the previous example according to the rules of the system, and we enumerate some of its limitations.

Towards a Type System for Analyzing JavaScript Programs Thiemann [21] was the first to propose a type system for a subset of JavaScript, offering support for first-class methods, extensible objects, dynamic function calls, and subtyping. The system is designed for a subset of JavaScript, which includes the main expressions of the language, such as literals, variables, property references, and function expressions, as well as the most traditional conditional and loop statements.

The type system supports numbers, strings, and booleans, as well as the `undefined` and `null` primitive types. Object types are written as:

$$\texttt{Obj}(\tau)(p_1 : \tau_1 \;\; \ldots \;\; p_n : \tau_n)(\tau'),$$

where τ describes the wrapper's type, $\tau_1 \;\ldots\; \tau_n$ specify the types of properties $p_1 \;\ldots\; p_n$, and τ' represents the default property type. The wrapper's type can be used to indicate that an object serves as a wrapper for a primitive type, allowing it to be used as if it was that primitive type. Additionally, default property types specify the type of all properties not explicitly declared in the object's type. For example, the object `{ x: 2, y: "foo" }` has type `Obj(undefined)(x: number, y: string)(undefined)`.

On the other hand, function types are written as:

$$\texttt{Func}(\texttt{this} : \tau \;\; ; \;\; (x_1 : \tau_1 \;\; \ldots \;\; x_n : \tau_n) \;\; \rightarrow \tau'),$$

where τ describes the type of the caller referenced by the `this` property, $\tau_1 \;\ldots\; \tau_n$ specify the types of parameters $x_1 \;\ldots\; x_n$, and τ' represents the function's return type. As an example, the type of a function that receives a number `n` and returns a boolean (e.g., a function that checks if a number is positive) is written as `Func(this: undefined ; (n: number)` → `boolean)`.

```
1  function R2(x, y) {
2    this.x = x;
3    this.y = y;
4    this.add = addR2;
5    return this
6  }

8  function addR2(n) {
9    this.x = this.x + n.x;
10   this.y = this.y + n.y;
11   return this
12 }

14 n1 = new R2(1, 2);
15 n2 = new R2(2, 3);
16 n1.add(n2)
```

Listing 3.2: Real numbers in a two-dimensional plane ($\mathbb{R}_2$) defined in Thiemann's type system [21].

Consider the running example of Listing 3.1 described at the beginning of this section. Listing 3.2 encodes this example in Thiemann's type system. The constructor function `R2(x, y)` for numbers in $\mathbb{R}_2$ receives two numbers and returns an object of type τ_1, defined as:

Type of $\mathbb{R}_2$ objects (τ_1): `Obj(undefined)(x: number, y: number, addR2:` τ_2`)(undefined)`
Type of the `addR2(n)` function (τ_2): `Func(this:` τ_3`, (n:` τ_3`)` → τ_3`)`
Type of simpler $\mathbb{R}_2$ objects (τ_3): `Obj(undefined)(x: number, y: number)(undefined)`

The object returned by `R2(x, y)` contains two fields (lines 2 and 3) and the method `addR2(n)` (line 4). Regarding the method's type (τ_2), we would expect it to be defined as `Func(this:` τ_1`, (n:` τ_1`)` → τ_1`)`, since `addR2(n)` is used as a method of $\mathbb{R}_2$ objects (represented by the type τ_1), and it also receives and returns another $\mathbb{R}_2$ object. However, because this system does not support recursive type declarations, we need to define τ_2 with respect to a simpler object type τ_3, such that τ_3 does not contain the types τ_1 and τ_2 in its definition. Despite this, we can still provide an object of type τ_1 to the `addR2(n)` function because τ_1 is a subtype of τ_3. This is the case because the system supports horizontal subtyping. This subtyping relation determines that an object type τ' is a subtype of τ if it contains all the fields of τ (with their respective types), meaning that it can be used wherever a τ object is required.

This type system has several limitations. One of them, as demonstrated by the previous example, is the lack of support for recursive type declarations. While they were not required in this particular instance, if the `addR2(n)` function was defined recursively, the system would not be able to type it. Moreover, the system does neither support prototype-base inheritance nor implicit type coercions, both of which are fundamental features of the JavaScript language.

Towards Type Inference for JavaScript The authors of [22] proposed a type system for a small fragment of the JavaScript language, referred to as JS_0. In addition to first-class methods and subtyping, the system supports recursive type declarations, and it is also flow-sensitive, meaning that object types can evolve throughout the execution of the program. Flow sensitivity is achieved by classifying object properties as potential or definite, a concept that is explained below. Furthermore, the authors presented a type inference algorithm and proved its soundness.

In this type system, object types are written as a map between properties and their respective types:

$$\{p_1 : (\tau_1, \psi_1)\ \ \ldots\ \ p_n : (\tau_n, \psi_n)\} \quad \text{or} \quad \mu\alpha.\{p_1 : (\tau_1, \psi_1)\ \ \ldots\ \ p_n : (\tau_n, \psi_n)\},$$

where (τ_i, ψ_i) specifies the type and potential/definite classification of property p_i, respectively. When objects are instantiated, all their fields are marked as potential ($\circ$), meaning that their type can change but they cannot be accessed. Once initialized, fields become definite ($\bullet$), meaning that their are accessible but their type is permanently fixed throughout the rest of the program's execution. This constitutes one possible approach for handling mutable objects in a sound type system.

The system allows the definition of recursive types through the use of the μ-binder. Consider the type for a node of a circular linked list[9] of numbers. This object requires a numeric `value` field and reference to the `next` element in the list. We can write this type as $\mu\alpha.\{\ \texttt{value: (number, }\bullet\texttt{), next: (}\alpha\texttt{, }\bullet\texttt{)}\ \}$, where α represents the type of the node itself.

A function type may have the following two forms:

$$\texttt{O} \times \tau \rightarrow \tau' \quad \text{or} \quad \mu\alpha.(\texttt{O} \times \tau \rightarrow \tau'),$$

where `O` specifies the type of the receiver, τ the type of the parameters, and τ' the function's return type. In JS$_0$, all functions must be associated with an object type; hence, the receiver's type τ refers to the type of the object bound by the `this` keyword. Furthermore, similarly to object types, recursive function are declared by employing the μ-binder.

```
function R2(x, y): (τ1 × (number, number) → τ2) {
  this.x = x;
  this.y = y;
  this.add = addR2;
  this
}

function addR2(n): (τ2 × (τ2) → τ2) {
  this.x = this.x + n.x;
  this.y = this.y + n.y;
  this
}
```

```
n1 = new R2(1, 2);
n2 = new R2(2, 3);
n1.add(n2)
```

Listing 3.3: Real numbers in a two-dimensional plane ($\mathbb{R}_2$) defined in JS$_0$ [22].

Listing 3.3 encodes the running example of Listing 3.1 in JS$_0$. The function `R2(x, y)` is the constructor for $\mathbb{R}_2$ numbers. Because the `this` keyword refers to the object being created, the type of the function's receiver τ_1 is the type that represents $\mathbb{R}_2$ objects with all fields marked as potential, indicating that they have not yet been initialized. Therefore, we can express τ_1 as:

$$\tau_1\ \texttt{=}\ \{\ \texttt{x: (number, }\circ\texttt{), y: (number, }\circ\texttt{), addR2: ((}\tau_2 \times (\tau_2) \rightarrow \tau_2\texttt{), }\circ\texttt{)}\ \}.$$

[9] Circular linked list - a type of linked list in which the last node points back to the first node, forming a loop.

By the end of the constructor (line 5), all object fields have been initialized. Consequently, the return type of the constructor function (τ_2) is identical to τ_1, except that all fields are now classified as definite:

$$\tau_2 = \mu\alpha.\{\ \texttt{x: (number, }\bullet\texttt{), y: (number, }\bullet\texttt{), addR2: ((}\alpha \times (\alpha) \rightarrow \alpha\texttt{), }\bullet\texttt{) \}}.$$

Note that the definition of the type for the `addR2(n)` function always references τ_2. This is the case because the function requires two fully constructed $\mathbb{R}_2$ objects to sum. Consequently, the type τ_2 shows up in its own definition, as τ_2 objects have a method that operates on τ_2 objects. In order to define the recursive type τ_2, we need to use the μ-binder operator.

JS$_0$ consists of an improvement over Thiemann's work [21]. Nonetheless, the system is still not perfect, as it only supports a very restricted subset of JavaScript. Furthermore, the approach for handling mutable object properties remains somewhat limited, as their types become permanently fixed after being initialized. In particular, this limitation prevents updates to property types in uniquely referenced objects. Such operations could be allowed without compromising the soundness of the system.

Understanding TypeScript The authors of [41] aimed to capture the essence of TypeScript by giving a precise definition of its type system on a core subset of the language, denoted Featherweight TypeScript (FTS). This subset was used to define two separate calculi: (1) a "safe" FTS fragment denoted as `safeFTS`, which was proven to be sound; and (2) an unsound extension of `safeFTS`, called `prodFTS`, that better resembles the production version of TypeScript.

In FTS, types fall into one of three categories: (1) primitive types, such as numbers, strings and booleans; (2) the distinguished type `any`, used to represent unknown types; and (3) object types. Object types are divided into literal types and interface types, respectively written as:

$$\{\ p_1\texttt{:}\ \tau_1\ \ldots\ p_n\texttt{:}\ \tau_n\ \}\quad|\quad \texttt{interface I}\ \{\ p_1\texttt{:}\ \tau_1\ \ldots\ p_n\texttt{:}\ \tau_n\ \},$$

where τ_i specifies the type of property p_i. Interface types are used to declare an alias `I` to an object literal type, and can also be used to specify subtyping relationships by extending other interface types.

Function types are defined as object literal types with a special call signature in their definition:

$$\{\ p_1\texttt{:}\ \tau_1\ \ldots\ p_n\texttt{:}\ \tau_n\texttt{,}\ (x_i\texttt{:}\ \psi_i\ \ldots\ x_m\texttt{:}\ \psi_m)\texttt{:}\ \psi'\ \},$$

where $\psi_1\ \ldots\ \psi_m$ specify the types of parameters $x_1\ \ldots\ x_m$, and ψ' defines the function's return type. This typing approach for functions resembles the real specification of JavaScript, where functions are represented by callable objects, often referred to as function objects.

Listing 3.4 illustrates the proposed type system by encoding the initial example of Listing 3.1 in FTS. In this snippet, $\mathbb{R}_2$ numbers are typed using the `I_R2` interface, declared as:

```
interface I_R2 { x: number, y: number, addR2: { this: I_R2, (n: I_R2): I_R2 } }
```

```
function R2(x: number, y: number): I_R2 {
  return { x: x, y: y, addR2: addR2 };
}

function addR2(n: I_R2): I_R2 {
  this.x = this.x + n.x;
  this.y = this.y + n.y;
  return this;
}

n1 = R2(1, 2);
n2 = R2(2, 3);
n1.addR2(n2);
```

Listing 3.4: Real numbers in a two-dimensional plane ($\mathbb{R}_2$) defined in Featherweight TypeScript [41].

where `x` and `y` represent the numeric properties that store both values of $\mathbb{R}_2$ numbers, and `addR2` corresponds to the method for adding $\mathbb{R}_2$ numbers. The type of `addR2` is defined as an object literal type with two properties: (1) the special identifier `this` of type `I_R2`, which allows the method to internally access the properties of the caller object, such as `x` and `y`; and (2) the call signature `(n: I_R2): I_R2`, which allows `addR2` to be called as a method that receives and returns an `I_R2` object.

Bierman et al. [41] showed that it extremely hard to develop a safe type system for JavaScript without making the language considerably less expressive. In particular, there are some important JS patterns that cannot be typed without unsafe typing features, such as the covariance of property types (despite their mutability) and parameter types (in addition to the contravariance that is always safe). ECMA-SL is similar to JavaScript in terms of having a large existing codebase that cannot be broken by the introduction of type annotations. However, one of the requirements for our type system is soundness, which prevents us from allowing unsafe typing features to minimize disruptions to the existing codebase. We further explore these challenges and associated design choices in Section 4.1.

4

Typed ECMA-SL

Contents

In this chapter, we introduce Typed ECMA-SL (TESL), an extension of the ECMA-SL language with types. In Section 4.1, we start by outlining the challenges associated with designing a static type system for TESL, as well as explaining the design choices made to accommodate those challenges. Then, in Section 4.2, we present the simplified formal model for Typed ECMA-SL, including its syntax and typing rules. Lastly, in Section 4.3, we describe the operational semantics for Typed ECMA-SL, and in Section 4.4, we prove the soundness of our type system with respect to the described semantics.

4.1 Challenges and Design Choices

In this section, we describe the most significant challenges encountered during the design of the Typed ECMA-SL language and its associated type system. After that, we discuss the most significant design choices made during the development of Typed ECMA-SL.

4.1.1 Challenges

As mentioned in Section 2.3, there are multiple challenges associated with statically typing highly dynamic languages, such as ECMA-SL. Additionally, the context in which ECMA-SL is currently being employed introduces further obstacles that should be taken into consideration. In the following, we describe the key challenges encountered in the development of the type system.

[C1] Large Existing Codebase One of the main challenges of extending ECMA-SL with types revolves around the existence of a large codebase written in ESL. Currently, we maintain two reference interpreters, one for the 5th version and another for the 6th version of the ES standard. Both interpreters comprise tens of thousands of lines of code. Consequently, any changes to the ECMA-SL language must be done in a way that minimizes disruption to these existing programs.

[C2] Pervasive Use of Union Types The ECMA-SL language is primarily used in the development of JS reference interpreters. In these interpreters, it is common to have variables that can hold multiple types of data. From a TypeScript standpoint, we would type these variables with union types.

```
type JSVal_t = number | string | boolean | JSObject_t | undefined | ...
```

Listing 4.1: Type declaration for JavaScript values.

Listing 4.1 illustrates the type declaration for JavaScript values, `JSVal_t`, written in TS. JavaScript supports multiple types of values, including integers and strings. To declare the type of a JS value, TESL needs to support union types. Nevertheless, union types are difficult to implement and introduce complexity to the language, as: (1) they require complex mechanisms to be simplified; and (2) they are potential sources of unsoundness. We delve into the details of union types further in the document.

[C3] Object Mutability + Aliasing One of the most significant challenges in statically typing highly dynamic languages pertains to the safe combination of object mutability and aliasing. Object mutability denotes the object's ability to change its type during the program's execution. This ability is useful, for example, to allow for type refinement. Type refinement refers to the process of converting a type into one of its subtypes based on some operation performed by the program. In TypeScript, for example, these refinements are usually associated with assignments or conditional exprssions [41, 44, 45].

```
1 let x: { foo: number | string } = { foo: 10 };  // x.foo: number | string
2 x.foo = 20;                                      // x.foo: number (refinement)
3
4 let a: number = x.foo;                           // Valid (x.foo: number)
```

Listing 4.2: Refining a union field type with a single reference.

Listing 4.2 demonstrates the process of refining a union field to one of its primitive types. In this snippet, we start by creating an object that we assign to variable `x` (line 1). Then, we assign the number `20` to `x.foo`, effectively refining its type to `number` (line 2). This means that, even though the field is of type `number | string`, the system can guarantee that `x.foo` is currently holding a number. As a result, it is safe to assign the value of `x.foo` to a variable of type `number` (line 4).

Without the ability to refine union field types to a single primitive type, these fields become obsolete, as they can only be assigned to other union types. However, these refinements create issues when the object being mutated has multiple references pointed to it. Performing a strong update on such an object could lead to scenarios where two references to the same object end up with different types.

```
1 let x: { foo: number | string } = { foo: 10 };  // x.foo: number | string
2 x.foo = 20;                                      // x.foo: number (refinement)
3
4 let y: { foo: number | string } = x;             // y.foo: number
5 y.foo = "abc";                                   // y.foo: string (refinement)
6
7 let a: number = x.foo;    // Valid (x.foo: number)
8 let b: number = y.foo;    // TypeError (y.foo: string)
```

Listing 4.3: Unsoundness of refining a union field type with multiple references.

Listing 4.3 illustrates the challenges associated with strong updates to object types by extending the previous example with multiple references. In this snippet, before assigning `x.foo` to the numeric variable, we create a new reference `y` to the object (line 4), which we then use to perform a field update (line 5), causing the refinement of `y.foo` to `string`. Notice, however, that the system is only able to update the type of the `y.foo`, leaving `x.foo` with the old type `number`, which is no longer valid. As a result, when we execute the assignments in lines 7 and 8 (both of which should be invalid since `foo` now holds a string), we only get the type error for the assignment involving the updated reference `y`.

The problem of safely combining object mutability and aliasing, as demonstrated by the previous example, exists within the TypeScript language. While TypeScript allows this behaviour, it leads to a fundamental problem: the unsoundness of its type system. In contrast, soundness is one of the main requirements of the type system for Typed ECMA-SL. Consequently, we will need to address this problem in the most effective manner while also maintaining the soundness of our system. This constitutes one of the biggest challenges in the development of the type system for Typed ECMA-SL.

4.1.2 Design Choices

During the design of Typed ECMA-SL, we followed a philosophy of prioritizing straightforward solutions that allowed us to type the reference interpreter with minimal changes. This decision was essential due to the existing large codebase (Challenge C1). In the following, we present each of the main design choices for Typed ECMA-SL. Some of these choices are associated with specific challenges, while others may not have a direct association but were essential in shaping the overall system.

[D1] Typed Core ECMA-SL vs ECMA-SL One of our main choices pertains to the target of the typing process. We have decided to work with ECMA-SL over Core ECMA-SL, as ECMA-SL is a higher-level language. When compiling a high-level language into a lower-level language, programs partially lose their structure, as compilers tend to simplify complex constructs into simpler ones. For example, as previously illustrated by Listing 2.2, the ECMA-SL `match-with` construct is expressed as a sequence of `if-then-else` statements when compiled into Core ECMA-SL.

Typing an intermediate language instead of the source language raises the challenge of providing efficient feedback to the developer. To properly do this, one would have to *lift* the error message from the intermediate language to the source language, which is far from trivial.

[D2] Fully Fledged vs Limited Typed Inference Another important decision in the development of Typed ECMA-SL is related to type declarations. We consider two options: (1) requiring the programmer to explicitly write type declarations, similarly to what happens in languages like C and Java; and (2) adopting a Haskell-like approach by using type inference to deduce variable types based on how they are used by the programmer. With the latter, if we have a function `f` that returns a number, the type system can deduce that any variable initialized with the result of `f()` is of type `number`.

The main benefit of type inference is the potential to eliminate the need for any refactoring of the existing ECMA-SL codebase (Challenge C1). Nevertheless, we opted to require explicit type declarations for two primary reasons: (1) it increases code readability as the types become present in the code itself; and (2) it is simpler and less error-prone to develop a system with explicit type annotations rather than to deduce all of them with a type inference algorithm, which may even not be possible in some cases.

[D3] Mandatory Typing vs Optional Typing Since Typed ECMA-SL requires type annotations, we also need to decide whether to make typing optional or mandatory. Due to its size, typing the entire ECMA-SL codebase must be done incrementally. Therefore, it is essential that the type system does not disrupt the existing code, meaning that typing needs to be optional at a function level. In other words, it need to be possible to have a program where some functions are typed and others are not. Futhermore, we will also support unchecked function signatures to declare the signature of functions that are not typed.

[D4] Support for Algebraic Datatypes To properly type the reference interpreter, Typed ECMA-SL requires a form of *sum types*, often referred to as disjoint union types. In the ES standard, most of the language constructs can be modelled using sum types. Take, for example, the type of JavaScript statements. Each kind of statement, such as the `if-then-else`, `while`, and `return`, can be represented by an object with fields that store the components required by the statement. Using sum types, we can safely combine all these object types into a single unified type, deconstructing it when necessary.

In Typed ECMA-SL, we decided to implement *sigma types*, a simplification of sum types specific to the memory model of ECMA-SL. Sigma types are very similar to disjoint unions of object types, with support for a limited form of type recursion [46, 47]. They can only be deconstructed using the `match-with` operation (explained in Section 2.2.2), making them a safe way of handling unions (Challenge C2).

```
typedef JSStmt_t := Σ[α]
  | { type: "ExprStmt", expr: JSExpr_t }
  | { type: "WhileStmt", body: α, ... }
  | ...
```

Listing 4.4: Partial declaration for the type of a JavaScript statement.

Listing 4.4 shows a simplified version of the type declaration for JavaScript statements, `JSStmt_t`. This example is written in TESL, which is introduced and explained in Section 4.2. In essence, each JS statement is modelled with an object type, which are then combined into the sigma type `JSStmt_t`. Additionally, the `while` statement resorts to the bound variable α to recursively type its `body` field.

[D5] Forbit Strong Updates Recall the problem of combining object mutability with aliasing, described in Challenge C3. This problem is known to be extremely difficult to solve and has been the subject of a large corpus of research [22, 48, 49, 50, 51]. TypeScript, for example, is one of the languages that undergoes this issue, and, despite currently being one of the most used programming languages, it does not provide a sound solution for it (as shown in Listing 4.3). However, soundness is of of the core requirements of Typed ECMA-SL, particularly because the language will be employed in the specification of JavaScript. Therefore, we need to address this issue in the most effective way possible while simultaneously minimizing disruptions to the existing codebase (Challenge C1).

One way we could attempt to solve this challenge is by implementing an object ownership system inspired by programming languages like Rust. In such a system, multiple references to objects are allowed, but only the owner reference can update union fields or refine their types. This is important to guarantee that no other references can override refinements previously executed by the object's owner.

Listing 4.5 addresses the unsound example of Listing 4.3 by employing the proposed object ownership system. When we create the reference `y` to the object (line 4), the system transfers the object's ownership from `x` to `y`. This operation also nullifies all prior refinements (such as the one in line 2) of the fields of `x`. As shown in line 5, we can still refine the type of a union field using the owner reference

```
1 let x: { foo: number | string } = { foo: 10 };  // x.foo: number | string
2 x.foo = 20;                                     // x.foo: number (refinement)
3
4 let y: { foo: number | string } = x;            // y becomes the owner
5 y.foo = "abc";                                  // y.foo: string (refinement)
6 x.foo = "def";                                  // TypeError (x is not the owner)
7
8 let a: number = x.foo;    // TypeError (x.foo: number | string)
9 let b: number = y.foo;    // TypeError (y.foo: string)
```

Listing 4.5: Refining a union field type with multiple references, using ownership types.

`y`. However, attempting to update `foo` using `x` (line 6) results in a type error, as this operation could potentially invalidate the strong update of line 5. Finally, both the assignments in lines 8 and 9 generate type errors. This is the expected behaviour because `foo` is currently holding a string. Recall that the system without ownership (Listing 4.3) could not detect the first error, as `x.foo` was still latched to an old refinement. Conversely, the ownership system invalidates all refinements for all but one reference.

While this object ownership system appears promising, its development proved to be excessively complex for our time constraints. As we delve into more intricate language constructs, such as function calls, `if-then-else` statements, and nested objects, defining the proper rules of the system becomes extremely challenging. Nevertheless, there is a lot of interest for further research in this area, as successfully addressing the safe combination of object mutability and aliasing could potentially improve static type systems for dynamic languages. After considering all options, we opted to prevent object mutability altogether within Typed ECMA-SL. While this approach may require additional effort when typing the reference interpreters, it offers a much more straightforward solution.

4.2 Formal Model

This section presents the formal model for a simplified version of Typed ECMA-SL. Due to its complexity, formalizing the real language and associated type system would both entail significant challenges and demand a substantial investment of time and effort. Nevertheless, this formalization is valuable as it allows us to reason about some key aspects of the type system and partially prove its soundness.

We divide this section into two parts. Firstly, we describe the syntax for the simplified version of TESL. After that, we outline the typing rules for expressions, statements, and other typing features.

4.2.1 Syntax

A Typed ECMA-SL program is a collection of functions and type declarations. TESL functions are written as $function\ g(x_1 : \tau_1,\ ...,\ x_n : \tau_n)\{s\}$, where g is the function identifier, $x_1,\ ...,\ x_n$ are the function's formal parameters with types $\tau_1,\ ...,\ \tau_n$, and s is the function's body. Type declarations are written as $typedef\ \alpha := \tau$, and are used to define type variables α that behave as aliases for types τ.

$e \in \mathit{Expr}$	$::=$	$v \mid x \mid \ominus\, e \mid e_1 \oplus e_2 \mid e \;\textsf{as}\; \tau$
$s \in \mathit{Stmt}$	$::=$	$\textsf{skip} \mid s_1 \;;\; s_2 \mid x \coloneqq e \mid x \coloneqq e.f \mid x \coloneqq \{f_i : e_i \mid_{i=1}^{n}\} \mid x \coloneqq g(e_1,\ ...,\ e_n) \mid$
		$x.f \coloneqq e \mid \textsf{return}\; e \mid \textsf{if}\; (e)\; \{s_1\}\; \textsf{else}\; \{s_2\} \mid \textsf{while}\; (e)\; \{s\} \mid$
		$\textsf{match}\; e\; \textsf{with}\; \{\psi_1 \to s_1,\ ...,\ \psi_n \to s_n\}$
$\psi \in \mathit{Pat}$	$::=$	$\{f_i : \xi_i \mid_{i=1}^{n}\}$
$\xi \in \mathit{PatB}$	$::=$	$x \mid v$

Table 4.1: Typed ECMA-SL Syntax - Expressions and Statements. The non-terminals $\langle v \rangle$, $\langle x \rangle$, $\langle g \rangle$, and $\langle f \rangle$ range over values, variable names, function names, and field names, respectively.

Table 4.1 outlines the syntax for expressions and statements in Typed ECMA-SL. Expressions $e \in \mathit{Expr}$ include: values v, variables x, unary $\ominus$ and binary $\oplus$ operators, and the type casting as operator. In terms of values, the system supports integers, floats, strings, booleans, object locations $l \in \mathit{Loc}$, and the `null` and `undefined` special values.

Statements $s \in \mathit{Stmt}$ include: the `skip`, sequences, variable assignments, field lookups, the creation of new objects, function calls, field assignments, and control flow elements such as the `return`, `if-then-else`, `while`, and `match-with` statements. Except for the `match-with` construct, which is explained in Section 2.2.2, all other language constructs are self-explanatory and behave as standard.

$t \in \mathit{TPrim}$	$::=$	$\texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \texttt{boolean} \mid \texttt{null} \mid \texttt{undefined} \mid \texttt{top}$
$\sigma \in \mathit{TObj}$	$::=$	$\{f_i : \tau_i \mid_{i=1}^{n}\} \mid \{f_i : \tau_i \mid_{i=1}^{n}, * : \tau^*\}$
$\tau \in T$	$::=$	$t \mid \sigma \mid \nu \mid \alpha \mid \cup\{\tau_1,\ ...,\ \tau_n\} \mid \Sigma[\alpha]\,\{\sigma_1,\ ...,\ \sigma_n\}$

Table 4.2: Typed ECMA-SL Syntax - Supported Types. The non-terminals $\langle f \rangle$, $\langle v \rangle$, and $\langle \alpha \rangle$, range over field names, values, and type variable names, respectively.

Supported Types Table 4.2 depicts the types available within Typed ECMA-SL. Given that ECMA-SL is essentially a simplification of JavaScript, the types supported in Typed ECMA-SL closely resemble those found in TypeScript. The language encompasses a set of primitive types $t \in \mathit{TPrim}$, which includes integers and floats (equivalent to TypeScript's `number` type), strings, booleans, and the special `null`, `undefined`, and `top` types. Additionally, the language supports: object types $\sigma \in \mathit{TObj}$, literal types $\nu \in \mathit{TLit}$, type variables $\alpha \in \mathit{TVar}$, union types $\cup\{\tau_1,\ ...,\ \tau_n\}$, and *sigma* types $\Sigma[\alpha]\,\{\sigma_1,\ ...,\ \sigma_n\}$. In the following, we relax the representation of union and sigma types to enhance clarity.

Object types comprise a one-to-one mapping between field names $f \in \mathit{Fld}$ and their corresponding types $\tau \in T$. On top of that, they allow the declaration of a special summary field ($*$) that specifies a default summary type (τ^*) for all fields not explicitly declared within the object's type. Summary fields are particularly useful, for example, when defining the type for environment records.[1] In TESL, this type can be written as `typedef EnvRecord_t := { *: JSVal_t }`, where `JSVal_t` is the type of JS values.

Literal types describe types that represent concrete values. In Typed ECMA-SL, integers, floats, strings and booleans values are all typed with their respective literal value. For example, the type of the integer `10`, is the literal type `10`. Formally, the set of literal types can be expressed as:

$$TLit = Int \cup Flt \cup Str \cup Bool.$$

```
1 typedef JSNum_t := int ∪ float;
2
3 num: JSNum_t := getNum();
4 if (typeof(num) = int) {
5   i: int := num
6 } else {
7   f: float := num
8 }
```

(a) Union types in Typed ECMA-SL.

```
1  typedef JSStmt_t := Σ[α]
2    | { type: "ExprStmt", expr: JSExpr_t }
3    | { type: "WhileStmt", body: α, ... }
4    | ...
5
6  stmt: JSStmt_t := getStmt();
7  match stmt with
8  | { type: "ExprStmt", expr: e } → {}
9  | { type: "WhileStmt", body: s } → {
10   stmt': JSStmt_t := s
11 }
```

(b) Sigma types in Typed ECMA-SL.

Figure 4.1: Examples of the formal language of Typed ECMA-SL.

Figure 4.1a illustrates union types described in the formal model of Typed ECMA-SL. Union types are used to combine multiple types into a single unified type, allowing variables and fields to hold values from any of those types. In line 1, we define the type variable `JSNum_t` as a union of integers and floats. Subsequently, in line 4, we employ the built-in `typeof` operator to check whether `num` is holding an integer value. Because of this test, the type system can safely refine the type of `num` to `int` inside the `if` block, allowing for the compilation of line 5. Additionally, the system can refine `num` to type `float` inside the `else` block, allowing the compilation of line 7. This is considered a safe operation because, at this point, it is impossible for `num` to be holding an integer value, or the execution would have proceeded to the *true* case of the `if-then-else` statement.

Figure 4.1b shows the declaration and deconstruction of a sigma type. In the formal model of TESL, sigma types comprise unions of object types specifically designed to be unfolded by the `match-with` statement. Moreover, they can be recursively defined by typing their objects fields with the bound variable α, which references the sigma type itself. In line 1, we declare the `JSStmt_t` type as a sigma (sum) of JS statements, including the expression and `while` statement. `While` statements need to store their `test` condition, which is of type `JSExpr_t`, as well as their `body`, which is of type `JSStmt_t`. Note that `JSStmt_t` is the type currently being declared, turning this into a recursive type declaration. As a result, the `body` field needs to be typed with the bound variable α, which will be recursively unfolded to `JSStmt_t` during the `match-with` statement (lines 9 and 10).

[1] In programming languages, environment records are used to store the bindings between identifiers and their corresponding values within a specific execution context.

4.2.2 Type System

Before defining our system, we need to establish some essential preliminary definitions. In particular, we resort to store typing environments Γ to bind each program variable with its corresponding type, and global typing contexts Δ to associate each function identifier with its corresponding function type. Additionally, we introduce the notion of an upper bound between two store typing environments, merging them to create a new one. The formal definitions for these functions are provided below.

Definition 1 (Store Typing Environment)**.** A store typing environment is a partial function $\Gamma : Var \rightharpoonup T$, which maps variables $x \in Var$ to types $\tau \in T$.

Definition 2 (Global Typing Context)**.** A global typing context is a partial function $\Delta : FuncID \rightharpoonup TFunc$, which maps function identifiers $g \in FuncID$ to function types in $TFunc$. In Typed ECMA-SL, function types are written as $(\tau_1,\ ...,\ \tau_n) \rightarrow \tau_r$, with $\tau_1,\ ...,\ \tau_n, \tau_r \in T$.

Definition 3 ($\Gamma_1 \sqcup \Gamma_2$)**.** The upper bound between two store typing environments Γ_1 and Γ_2, written as $\Gamma_1 \sqcup \Gamma_2 : Var \rightharpoonup T$, is defined as:

$$\Gamma_1 \sqcup \Gamma_2(x) = \begin{cases} \Gamma_1(x) \cup \Gamma_2(x) & \text{if } x \in \Gamma_1 \wedge x \in \Gamma_2 \\ \Gamma_1(x) \cup \texttt{undefined} & \text{if } x \in \Gamma_1 \wedge x \notin \Gamma_2 \\ \Gamma_2(x) \cup \texttt{undefined} & \text{if } x \notin \Gamma_1 \wedge x \in \Gamma_2 \end{cases}$$

There are some additional considerations regarding the formal model of the type system. Firstly, we assume that all type variables $\alpha \in TVar$ are fully resolved into their respective types before employing the type checker. To convert a value into its type, we define the function $type : Val \mapsto T$, which maps values $v \in Val$ to types $\tau \in T$. Additionally, we introduce the function $typeof : TLit \mapsto TPrim$, which maps literal types $\nu \in TLit$ to their respective primitive types $t \in TPrim$. For example, the result of $typeof(10)$ is the primitive type `int`. We provide the definition for these functions in Appendix B.

Lastly, to handle field types effectively, we introduce the partial function $ft : TObj \times Fld \rightharpoonup T$, which maps pairs between object types $\sigma \in TObj$ and field names $f \in Fld$ to field types $\tau \in T$. The resulting type τ is the declared field type if the field exists, or the default summary type (when specified) if the field does not exist. Figure 4.2 outlines the typing rules for this function.

FIELD TYPE

$$ft(\{f_i : \tau_i \mid_{i=1}^{n}, f : \tau\},\ f) \triangleq \tau$$

SUMMARY FIELD TYPE

$$\frac{f \notin f_1,\ ...,\ f_n}{ft(\{f_i : \tau_i \mid_{i=1}^{n}, * : \tau^*\},\ f) \triangleq \tau^* \cup \texttt{undefined}}$$

Figure 4.2: Typing Rules - Object Fields: $ft(\sigma, f) \triangleq \tau$

Typing Rules for Expressions Let Γ be a store typing environment, e an expression, and τ a type. We denote that Γ types the expression e with type τ as $\Gamma \vdash e : \tau$, provided there exists a valid derivation for the judgement according to the typing rules outlined in Figure 4.3.

$$\begin{array}{ccc}
\textsc{Value} & \textsc{Variable} & \textsc{Unary Operation} \\
\dfrac{\mathit{type}(v) = \nu}{\Gamma \vdash v : \nu} & \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} & \dfrac{\Gamma \vdash e : \tau_e \qquad \ominus(\tau_e) = \tau}{\Gamma \vdash \ominus\, e : \tau}
\end{array}$$

$$\begin{array}{cc}
\textsc{Binary Operation} & \textsc{Type Casting} \\
\dfrac{\Gamma \vdash e_i : \tau_i \,|_{i=1}^{2} \qquad \oplus(\tau_1, \tau_2) = \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau} & \dfrac{\Gamma \vdash e : \tau_e \qquad \tau_e \leq \tau}{\Gamma \vdash e \text{ as } \tau : \tau}
\end{array}$$

Figure 4.3: Typing Rules - Expressions: $\Gamma \vdash e : \tau$

To illustrate the typing rules for expressions, consider the UNARY OPERATION rule. The type τ that results from applying the unary operation $\ominus$ to expression e is given by the application of the operator to τ_e. The type τ_e is determined by typing the expression e, as $\Gamma \vdash e : \tau_e$.

The remaining rules are analogous to this one. In the rule for typing values, we employ the type function to determine the type associated with the provided value. Additionally, we introduce a rule for TYPE CASTING that allows us to type expressions with a more generic type than their original. Note that the model does not support downcasting, as this would become a source of unsoundness.

Typing Rules for Statements Let g be a function's identifier, Δ a global typing context, Γ and Γ' two store typing environments, and s a statement. The typing judgement $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ indicates that s occurs within the body of g and that under the global typing context Δ, the execution of s on a variable store satisfying the initial typing environment Γ results in a variable store satisfying the final variable typing environment Γ'. The respective typing rules are depicted in Figure 4.4.

To illustrate the typing rules for statements, consider the FIELD LOOKUP rule. To type this construct, the type system starts by typing the expression e with the object type σ, as we require the provided expression to be an object. Following that, it employs the ft function to determine the type of field f (referred to as τ_f), within the context of σ. Lastly, the system updates the store typing environment Γ with the new binding from variable x to type τ_f.

The remaining rules are standard, with the exception of the IF-THEN-ELSE and WHILE LOOP rules, that employ the Conditional Type Refinement (CTR) mechanism to refine types according to the conditional expression, and MATCH-WITH rule that employs the Type Pattern Binding (TPB) operator to update the store typing environment with the pattern bindings. Also, note that field assignments do not modify the store typing environment, as we do not allow for strong updates on field types (Design Choice D5).

Skip

$$g, \Delta \vdash \{\Gamma\}\ \mathsf{skip}\ \{\Gamma\}$$

Sequencing

$$\frac{g, \Delta \vdash \{\Gamma_{i-1}\}\ s_i\ \{\Gamma_i\}\ |_{i=1}^{2}}{g, \Delta \vdash \{\Gamma_0\}\ s_1\ ;\ s_2\ \{\Gamma_2\}}$$

Variable Assignment

$$\frac{\Gamma \vdash e : \tau_e}{g, \Delta \vdash \{\Gamma\}\ x \coloneqq e\ \{\Gamma[x \mapsto \tau_e]\}}$$

Function Call

$$\frac{\Delta(g') = (\tau_1,\ ...,\ \tau_n) \to \tau_r \qquad \Gamma \vdash e_i : \tau'_i\ |_{i=1}^{n} \qquad \tau'_i \leq \tau_i\ |_{i=1}^{n}}{g, \Delta \vdash \{\Gamma\}\ x \coloneqq g'(e_1,\ ...,\ e_n)\ \{\Gamma[x \mapsto \tau_r]\}}$$

New Object

$$\frac{\Gamma \vdash e_i : \tau_i\ |_{i=1}^{n} \qquad \sigma = \{f_i : \tau_i\ |_{i=1}^{n}\}}{g, \Delta \vdash \{\Gamma\}\ x \coloneqq \{f_i : e_i\ |_{i=1}^{n}\}\ \{\Gamma[x \mapsto \sigma]\}}$$

Field Lookup

$$\frac{\Gamma \vdash e : \sigma \qquad ft(\sigma, f) = \tau_f}{g, \Delta \vdash \{\Gamma\}\ x \coloneqq e.f\ \{\Gamma[x \mapsto \tau_f]\}}$$

Field Assignment

$$\frac{\begin{array}{c}\Gamma(x) = \sigma \qquad ft(\sigma, f) = \tau_f \\ \Gamma \vdash e : \tau_e \qquad \tau_e \leq \tau_f\end{array}}{g, \Delta \vdash \{\Gamma\}\ x.f \coloneqq e\ \{\Gamma\}}$$

Return

$$\frac{\begin{array}{c}\Delta(g) = (\tau_1,\ ...,\ \tau_n) \mapsto \tau_r \\ \Gamma \vdash e : \tau_e \qquad \tau_e \leq \tau_r\end{array}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{return}\ e\ \{\Gamma\}}$$

If-Then-Else

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \qquad \tau_e \leq \mathit{bool} \qquad \beta_\Gamma(e) = \Gamma_1 \\ \beta_\Gamma(\neg e) = \Gamma_2 \qquad g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma'_i\}\ |_{i=1}^{2}\end{array}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{if}\ (e)\ \{s_1\}\ \mathsf{else}\ \{s_2\}\ \{\Gamma'_1 \sqcup \Gamma'_2\}}$$

While Loop

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \qquad \tau_e \leq \mathit{bool} \\ \beta_\Gamma(e) = \Gamma' \qquad g, \Delta \vdash \{\Gamma'\}\ s\ \{\Gamma\}\end{array}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{while}\ (e)\ \{s\}\ \{\Gamma\}}$$

Match-With

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \qquad \tau = \Sigma[\alpha]\,\{\sigma_1,\ ...,\ \sigma_n\} \qquad \Psi^{\alpha,\tau}(\psi_i, \sigma_i, \Gamma) = \Gamma_i\ |_{i=1}^{n} \\ g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma'_i\}\ |_{i=1}^{n} \qquad \Gamma' = \sqcup\{\Gamma'_1,\ ...,\ \Gamma'_n\}\end{array}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{match}\ e\ \mathsf{with}\ \{\psi_i \to s_i\ |_{i=1}^{n}\}\ \{\Gamma'\}}$$

Figure 4.4: Typing Rules - Statements: $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$

In the following, we provide a description of both the Conditional Type Refinement (CTR) and Type Pattern Binding (TPB) mechanisms, along with their respective typing rules. After that, we describe the Subtyping relationship supported by the formal model of Typed ECMA-SL.

Conditional Type Refinement (CTR) The evaluation of conditional expressions within `if-then-else` and `while` statements often leads to an enhanced comprehension of the values stored within the scope. In Figure 4.1a, we showed how the `typeof` operator could be employed to narrow a variable's type from a union to a single primitive type. Value comparisons can also be used to further refine variable types, particularly from primitive to literal types. For instance, the conditional expression in `if (x = 10) {...}` effectively refines the type of `x` from `int` to `10`, within the *true* branch of the `if-then-else` statement.

$$\frac{\beta_\Gamma(e_1) = \Gamma_1 \qquad \beta_{\Gamma_1}(e_2) = \Gamma_2}{\widehat{\beta}_\Gamma(e_1 \text{ and } e_2) \triangleq \Gamma_2} \qquad \frac{\mathit{type}(v) = \tau \qquad \tau \leq \Gamma(x)}{\widehat{\beta}_\Gamma(x = v) \triangleq \Gamma[x \mapsto \tau]} \qquad \frac{\mathit{type}(v) = \tau \qquad \tau \leq \Gamma(x)}{\widehat{\beta}_\Gamma(v = x) \triangleq \Gamma[x \mapsto \tau]}$$

$$\frac{\tau \leq \Gamma(x)}{\widehat{\beta}_\Gamma(\text{typeof}(x) = \tau) \triangleq \Gamma[x \mapsto \tau]} \qquad \frac{\Gamma(x) = \cup\{\tau_1,\ ...,\ \tau_n\} \qquad \tau' = \cup(\{\tau_1,\ ...,\ \tau_n\} \setminus \tau)}{\widehat{\beta}_\Gamma(\text{typeof}(x) \neq \tau) \triangleq \Gamma[x \mapsto \tau']}$$

$$\frac{\tau \leq \Gamma(x)}{\widehat{\beta}_\Gamma(\tau = \text{typeof}(x)) \triangleq \Gamma[x \mapsto \tau]} \qquad \frac{\Gamma(x) = \cup\{\tau_1,\ ...,\ \tau_n\} \qquad \tau' = \cup(\{\tau_1,\ ...,\ \tau_n\} \setminus \tau)}{\widehat{\beta}_\Gamma(\tau \neq \text{typeof}(x)) \triangleq \Gamma[x \mapsto \tau']}$$

$$\beta_\Gamma(e) \triangleq \begin{cases} \widehat{\beta}_\Gamma(e) & \text{if } e \in \mathit{dom}(\widehat{\beta}_\Gamma) \\ \Gamma & \text{otherwise} \end{cases}$$

Figure 4.5: Typing Rules - Conditional Type Refinement: $\beta_\Gamma(e) \triangleq \Gamma'$

In Typed ECMA-SL, we model the CTR mechanism with the function $\beta_\Gamma(e) \triangleq \Gamma'$, which uses expressions $e \in \mathit{Expr}$ to refine variables within the store typing environment Γ, generating a new store typing environment Γ'. Figure 4.5 outlines the typing rules for this function. Although limited, this model effectively addresses the most common refinements of the language and supports the combination of multiple refinements through the use of the logical `and` operator.

Type Pattern Binding (TPB) In the context of `match-with` statements, patterns $\psi \in \mathit{Pat}$ establish a correspondence between fields $f \in \mathit{Fld}$ and pattern bindings $\xi \in \mathit{PatB}$. Formally, the set of pattern bindings can be expressed as: $\mathit{PatB} = \mathit{Val} \cup \mathit{Var}$. Besides being used to define which expressions are valid for a particular `match-with` case, patterns may also create new variables in the scope of that case. In Figure 4.1b, the pattern `{ type: "ExprStmt", expr: e }` (line 8) introduces variable `e` within the first `match-with` case, initializing it with the value of field `expr`. Therefore, the system needs to update Γ accordingly, storing a new binding between variable `e` and the type of field `expr`.

$$\begin{array}{ll} \Psi^{\alpha,\tau}(\{\}, \sigma', \Gamma) \triangleq \Gamma & \\ \Psi^{\alpha,\tau}(\{f : v\} \uplus \psi', \{f : \tau_f\} \uplus \sigma', \Gamma) \triangleq \Psi^{\alpha,\tau}(\psi', \sigma', \Gamma) & \text{if } \mathit{type}(v) \leq \tau_f \\ \Psi^{\alpha,\tau}(\{f : x\} \uplus \psi', \{f : \tau_f\} \uplus \sigma', \Gamma) \triangleq \Psi^{\alpha,\tau}(\psi', \sigma', \Gamma[x \mapsto \tau_f[\alpha/\tau]]) & \end{array}$$

Figure 4.6: Typing Rules - Type Pattern Binding: $\Psi^{\alpha,\tau}(\psi, \sigma, \Gamma) \triangleq \Gamma'$

In Typed ECMA-SL, we model the TPB mechanism with the function $\Psi^{\alpha,\tau}(\psi, \sigma, \Gamma) \triangleq \Gamma'$, which uses patterns $\psi \in \mathit{Pat}$ and object types $\sigma \in \mathit{TObj}$ (pertaining to the sigma type τ) to update the store typing environment Γ, generating the new environment Γ'. Figure 4.6 outlines the typing rules for this function. In this figure, the operator $\uplus$ is used to represent the disjoint union of sets (patterns and objects).

In the pattern, each field f_i is mapped to the pattern binding ξ_i, which is either a value $v \in Val$ or a variable $x \in Var$. If ξ_i is a value, then the system does not modify Γ, but it must ensure that the type of v conforms to the field's type. This prevents values from being bound to fields of an unrelated type.

On the other hand, if ξ_i is a variable, the system must update Γ with the new variable, whose type is determined by unfolding the type of the associated field. In this context, unfolding the type involves replacing any bound variables α found within the field's type with the initial sigma type. To illustrate this process, consider lines 9 and 10 in Figure 4.1b. In this example, the type of the new variable `s` corresponds to the type of `body` (typed as the bound variable α), after being unfolded. Unfolding this type allows the system to recover the original sigma type `JSStmt_t`, ensuring proper type recursion.

Subtyping Figure 4.7 depicts the typing rules concerning the subtyping relationship supported in Typed ECMA-SL. The formal model of the language establishes subtyping for union types, literal (value) types, object types (with and without the summary field), and sigma types.

Reflexivity

$$\tau \leq \tau$$

Transitivity

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

Union Subtyping

$$\frac{\tau \leq \tau_i}{\tau \leq \cup\{\tau_1,\ \dots,\ \tau_n\}}$$

Top Subtyping

$$\tau \leq \texttt{top}$$

Value Subtyping

$$\nu \leq \mathit{typeof}(\nu)$$

Horizontal Subtyping

$$\{f_i : \tau_i \mid_{i=1}^{n},\ f'_j : \tau'_j \mid_{j=1}^{m}\} \leq \{f_i : \tau_i \mid_{i=1}^{n}\}$$

$$\{f_i : \tau_i \mid_{i=1}^{n},\ * : \tau^*\} \leq \{f_i : \tau_i \mid_{i=1}^{n}\}$$

Summary Subtyping

$$\{f_i : \tau_i \mid_{i=1}^{n},\ f'_j : \tau^* \mid_{j=1}^{m}\} \leq \{f_i : \tau_i \mid_{i=1}^{n},\ * : \tau^*\}$$

Sigma Case Subtyping

$$\frac{\sigma_i \leq \sigma'_i \mid_{i=1}^{n}}{\Sigma[\alpha]\,\{\sigma_1,\ \dots,\ \sigma_n\} \leq \Sigma[\alpha]\,\{\sigma'_1,\ \dots,\ \sigma'_n\}}$$

Sigma Folding

$$\frac{\Sigma[\alpha]\,\{\sigma_1,\ \dots,\ \sigma_n\} = \tau \qquad \sigma \leq \sigma_i[\tau/\alpha]}{\sigma \leq \Sigma[\alpha]\,\{\sigma_1,\ \dots,\ \sigma_n\}}$$

Figure 4.7: Typing Rules - Subtyping: $\tau \leq \tau'$

As expected, the subtyping relation in Typed ECMA-SL is both reflexive and transitive. The system defines subtyping for literal types ν through the use of the typeof function (described at the beginning of this section), and all types are inherently subtypes of `top`. As for union types, a type τ is considered a subtype of the union type $\cup\{\tau_1,\ \dots,\ \tau_n\}$ if it is a subtype of at least one of the types τ_i pertaining to the union (e.g., `int` $\leq \cup\{$ `int`, `float` $\}$). Furthermore, the system features horizontal subtyping, wherein an object type σ is a subtype of σ' if it includes all fields declared for σ' with their respective types. The same principle applies when σ includes a summary field, as in this scenario, a summary field is equivalent to an arbitrary number of additional fields, all of which share the same type.

To ensure soundness, the model does not allow field covariance [41,46]. Field covariance establishes a subtyping relationship between objects that specify a more specific type for a given field compared to the field's type in the base object. Under field covariance, an object value like `{ foo: 10 }` is considered a subtype of `{ foo: int }` (since `10` is a subtype of `int`), allowing it to be assigned to an entity of this type. Without field covariance, the previous assignment generates a type error because the types of `foo` in both objects do not precisely match. This restriction significantly reduces the expressiveness of the system, particularly when initializing objects. To circumvent it, we can employ the type casting operator and instead define the object value as `{ foo: (10 as int) }`.

The system also defines a subtyping rule referred to as SUMMARY SUBTYPING. According to this rule, an object type σ is deemed a subtype of σ' if its extra fields match the summary type specified for σ'. Using this rule, we can assign an object of type `{ foo: int, bar: string }` to an entity of type `{ foo: int, *: string }` because the extra field `bar` is of type `string`.

Lastly, the model introduces two subtyping rules for sigma types. SIGMA CASE SUBTYPING describes a simple subtyping relationship between the sigma cases. The SIGMA FOLDING rule, however, is particularly interesting, as it provides a mechanism to fold one of the disjoint sigma cases back into the sigma type itself. To illustrate this, consider the partial declaration for the `List_t` type, defined as $\Sigma[\alpha]$ `{ next: null } | { next:` α `}`. Under this rule, $\{$ `next: List_t` $\}$ is a subtype of `List_t` because it is a subtype of the second disjoint sigma case $\{$ `next:` α $\}$ after replacing α with `List_t`.

4.3 Operational Semantics

In this section, we introduce our big-step operational semantics for Typed ECMA-SL. For simplicity, we restrict ourselves to modelling the correct behaviour of the program without accounting for erroneous executions. This approach allows us to focus on the intended functionality of Typed ECMA-SL.

To ensure a clear presentation, we break down this section into two distinct parts. First, in Section 4.3.1, we model the semantics of the intra-procedural fragment of Typed ECMA-SL. This fragment excludes both function calls and the `return` statement. Subsequently, in Section 4.3.2, we extend the semantics of the language to incorporate these omitted elements.

4.3.1 Intra-Procedural Fragment

Before outlining the operational semantics for Typed ECMA-SL, we need to establish some preliminary definitions. We define Typed ECMA-SL states, used to represent the current state of a Typed ECMA-SL execution. Additionally, we use heap typing environments $\mathcal{H}$ to associate each heap location with the type of object it points to, and we extend the $type$ function (defined in Appendix B) to reason about the type of a location at runtime. The respective formal definitions are provided below.

Definition 4 (Typed ECMA-SL State)**.** An ECMA-SL state is composed of a heap $\eta : Loc \times Fld \rightharpoonup Val$, which maps pairs of locations $l \in Loc$ and field names $f \in Fld$ to values $v \in Val$, and a store $\gamma : Var \rightharpoonup Val$, which maps variables $x \in Var$ to values $v \in Val$.

In line with established methodologies for modelling the semantics of JavaScript [16, 29], we do not model the heap as a function from locations to objects. Instead, we refrain from explicitly representing objects in our formalism and choose to conceptualize them as *heap regions* [28, 52]. In the following, we use the notation $\eta(l)$ to mean $\{(l, f) \mid (l, f) \in dom(\eta)\}$ and $dom(\eta(l))$ to mean $\{f \mid (l, f) \in dom(l)\}$.

Definition 5 (Heap Typing Environment)**.** A heap typing environment is a partial function $\mathcal{H} : Loc \rightharpoonup T$, which maps heap locations $l \in Loc$ to types $\tau \in T$.

In the following, we use $\mathcal{H}(l, f)$ to refer to the type of field f in the object pointed by location l. Formally, this can be expressed as: $\mathcal{H}(l, f) = \tau_f \iff \exists\, \tau.\, \mathcal{H}(l) = \tau \wedge \tau = \{..., \; f : \tau_f, \; ...\}$.

Regarding the $type$ function, we extend it as the partial function $type_{\mathcal{H}} : Val \rightharpoonup T$, also allowing it to determine the type of a location value $l \in Loc$ at runtime. The extended function maps values $v \in Val$ to their respective types $\tau \in T$ within the context of the heap typing environment $\mathcal{H}$, and is defined as:

$$type_{\mathcal{H}}(v) = \begin{cases} \mathcal{H}(l) & \text{if } v \text{ is a location } l \\ type(v) & \text{otherwise.} \end{cases}$$

Semantics for Expressions Let γ be a store, e an expression, and v a value. We denote that under γ, the evaluation of e yields v as $[\![e]\!]_\gamma \triangleq v$, provided there exists a valid derivation for $[\![e]\!]_\gamma \triangleq v$ according to the semantics outlined in Figure 4.8.

VALUE
$$[\![v]\!]_\gamma \triangleq v$$

VARIABLE
$$[\![x]\!]_\gamma \triangleq \gamma(x)$$

UNARY OPERATION
$$[\![\ominus\; e]\!]_\gamma \triangleq \underline{\ominus}([\![e]\!]_\gamma)$$

BINARY OPERATION
$$[\![e_1 \oplus e_2]\!]_\gamma \triangleq \underline{\oplus}([\![e_1]\!]_\gamma, [\![e_2]\!]_\gamma)$$

TYPE CASTING
$$[\![e \text{ as } \tau]\!]_\gamma \triangleq [\![e]\!]_\gamma$$

Figure 4.8: Operational Semantics - Expressions: $[\![e]\!]_\gamma \triangleq v$

To illustrate the semantics of expressions, consider the UNARY OPERATION rule. To evaluate the unary operation $\ominus(e)$, the semantics start by evaluating the argument expression e and then apply the semantic function of the given unary operator $\underline{\ominus}$ to the obtained value.

The remaining rules are analogous to this one. Values evaluate to themselves and variables evaluate to the value that they are currently storing within γ. Note that the type casting expression evaluates to the value of the expression itself, since type casting is designed to be used solely by the type system.

Semantics for Statements Let $\mathcal{H}$ and $\mathcal{H}'$ be two heap typing environments, η and η' two heaps, γ and γ' two stores, and s a statement. The semantic judgement $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$ indicates that the evaluation of statement s in the heap η and store γ results in the heap η' and the store γ'. This judgement is instrumented with the heap typing environment $\mathcal{H}$ to keep track of the object types during the execution. The semantic rules for the intra-procedural fragment of TESL are outlined in Figure 4.9.

Skip

$$\langle \mathcal{H}, \eta, \gamma, \mathsf{skip} \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma \rangle$$

Sequencing

$$\frac{\langle \mathcal{H}_{i-1}, \eta_{i-1}, \gamma_{i-1}, s_i \rangle \Downarrow_i \langle \mathcal{H}_i, \eta_i, \gamma_i \rangle \,|_{i=1}^{2}}{\langle \mathcal{H}_0, \eta_0, \gamma_0, s_1;\ s_2 \rangle \Downarrow_i \langle \mathcal{H}_2, \eta_2, \gamma_2 \rangle}$$

Variable Assignment

$$\frac{[\![e]\!]_\gamma = v}{\langle \mathcal{H}, \eta, \gamma, x := e \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma[x \mapsto v] \rangle}$$

Field Lookup

$$\frac{[\![e]\!]_\gamma = l \qquad \eta(l, f) = v}{\langle \mathcal{H}, \eta, \gamma, x := e.f \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma[x \mapsto v] \rangle}$$

New Object

$$\frac{\begin{array}{c} l \notin dom(\eta) \qquad [\![e_i]\!]_\gamma = v_i \,|_{i=1}^{n} \qquad \{f_i : type_{\mathcal{H}}(v_i) \,|_{i=1}^{n}\} = \tau \\ \mathcal{H}' = \mathcal{H}[l \mapsto \tau] \qquad \eta' = \eta[(l, f_i) \mapsto v_i] \,|_{i=1}^{n} \end{array}}{\langle \mathcal{H}, \eta, \gamma, x := \{f_i : e_i \,|_{i=1}^{n}\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma[x \mapsto l] \rangle}$$

Field Assignment

$$\frac{[\![x]\!]_\gamma = l \qquad (l, f) \in dom(\eta) \qquad [\![e]\!]_\gamma = v \qquad \mathcal{H}(l)[f \mapsto type_{\mathcal{H}}(v)] = \tau}{\langle \mathcal{H}, \eta, \gamma, x.f := e \rangle \Downarrow_i \langle \mathcal{H}[l \mapsto \tau], \eta[(l, f) \mapsto v], \gamma \rangle}$$

If-Then-Else (True)

$$\frac{[\![e]\!]_\gamma = true \qquad \langle \mathcal{H}, \eta, \gamma, s_1 \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}{\langle \mathcal{H}, \eta, \gamma, \mathsf{if}\,(e)\{s_1\}\,\mathsf{else}\,\{s_2\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}$$

If-Then-Else (False)

$$\frac{[\![e]\!]_\gamma = false \qquad \langle \mathcal{H}, \eta, \gamma, s_2 \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}{\langle \mathcal{H}, \eta, \gamma, \mathsf{if}\,(e)\{s_1\}\,\mathsf{else}\,\{s_2\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}$$

While Loop (True)

$$\frac{\begin{array}{c} [\![e]\!]_\gamma = true \qquad \langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}'', \eta'', \gamma'' \rangle \\ \langle \mathcal{H}'', \eta'', \gamma'', \mathsf{while}\,(e)\{s\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle \end{array}}{\langle \mathcal{H}, \eta, \gamma, \mathsf{while}\,(e)\{s\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}$$

While Loop (False)

$$\frac{[\![e]\!]_\gamma = false}{\langle \mathcal{H}, \eta, \gamma, \mathsf{while}\,(e)\{s\} \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma \rangle}$$

Match-With

$$\frac{[\![e]\!]_\gamma = l \qquad match(\psi_j, \eta(l), \gamma) = \gamma_j \qquad \langle \mathcal{H}, \eta, \gamma_j, s_j \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}{\langle \mathcal{H}, \eta, \gamma, \mathsf{match}\,e\,\mathsf{with}\,\{\psi_i \to s_i \,|_{i=1}^{n}\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle}$$

Figure 4.9: Operational Semantics - Statements: $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

To illustrate the semantic rules for statements, consider the Field Lookup rule. To evaluate this construct, the system starts by evaluating the expression e, obtaining the object location l. Following that, it retrieves the value v stored within field f of the object pointed to by l. Lastly, the system updates the program store γ with the new binding from variable x to value v.

The remaining rules are standard, with the exception of the MATCH-WITH rule. In order to update the store γ according to the bindings in the matched pattern ψ_j, the semantics employs the partial function $match : Pat \times Obj \times \gamma \rightharpoonup \gamma'$. This function updates a store γ according to the pattern $\psi \in Pat$ and an object value (or heap region) $o = \{f_1 : v_1,\ ...,\ f_n : v_n\}$, generating a new program store γ'. Figure 4.10 outlines the semantic rules for the $match$ function. In this figure, the operator $\uplus$ is used to represent the disjoint union of sets.

$$
\begin{aligned}
&match(\{\}, o', \gamma) \triangleq \gamma \\
&match(\{f : v\} \uplus \psi', \{f : v\} \uplus o', \gamma) \triangleq match(\psi', o', \gamma) \\
&match(\{f : x\} \uplus \psi', \{f : v\} \uplus o', \gamma) \triangleq match(\psi', o', \gamma[x \mapsto v])
\end{aligned}
$$

Figure 4.10: Operational Semantics - Pattern Binding: $match(\psi, o, \gamma) \triangleq \gamma'$

4.3.2 Function Calls and Returns

In this section, we extend the operational semantics with support for function calls and the `return` statement. To do so, we need to extend our semantic judgement with program contexts Ω to retrieve the function's formal parameters and body, and statement outcomes $\phi \in \Phi$ [53] to capture the flow of execution. The extended semantic judgement has the form $\langle \Omega, \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma', \phi \rangle$ and indicates that, within the program context Ω, the evaluation of statement s in the heap η and store γ has the outcome ϕ resulting in a heap η' and the store γ'. The respective formal definitions are provided below.

Definition 6 (Program Context)**.** A program context is a function $\Omega : FuncID \mapsto (Var^* \times Stmt)$ that maps function identifiers $g \in FuncID$ to function definitions $Var^* \times Stmt$. Function definitions are written as $(x_1,\ ...,\ x_n) \rightarrow s$, with $x_1,\ ...,\ x_n \in Var$ and $s \in Stmt$.

A statement outcome $\phi \in \Phi$ represents the outcome of evaluating a Typed ECMA-SL statement. It can take one of two forms: (1) the continuation outcome $Cont$, which indicates that the execution may continue with the next statement; and (2) the return outcome $Ret(v)$, which indicates that the current function has returned with value v. Formally, the set of statement outcomes can be expressed as:

$$\Phi = \{Cont\} \cup \{Ret(v) \mid v \in Val\}.$$

The semantic rules for the extended fragment of Typed ECMA-SL are outlined in Figure 4.11. This figure depicts the extended semantics for sequences, function calls, and the `return` statement. Additionally, we provide the extended VARIABLE ASSIGNMENT rule to illustrate the natural adaptation of a TESL statement to the extended semantics. The remaining rules can be adapted in an analogous way.

Illustratively, consider the FUNCTION CALL rule. To evaluate this construct, the system starts by retrieving the formal parameters $x_1,\ ...,\ x_n$ and body s of the called function g. Following that, it evaluates

Sequencing (Cont)

$$\frac{\begin{array}{c}\langle \Omega, \mathcal{H}_0, \eta_0, \gamma_0, s_1 \rangle \Downarrow_i \langle \mathcal{H}_1, \eta_1, \gamma_1, \mathit{Cont} \rangle \\ \langle \Omega, \mathcal{H}_1, \eta_1, \gamma_1, s_2 \rangle \Downarrow_i \langle \mathcal{H}_2, \eta_2, \gamma_2, \phi \rangle\end{array}}{\langle \Omega, \mathcal{H}_0, \eta_0, \gamma_0, s_1;\ s_2 \rangle \Downarrow_i \langle \mathcal{H}_2, \eta_2, \gamma_2, \phi \rangle}$$

Sequencing (Ret)

$$\frac{\langle \Omega, \mathcal{H}_0, \eta_0, \gamma_0, s_1 \rangle \Downarrow_i \langle \mathcal{H}_1, \eta_1, \gamma_1, \mathit{Ret}(v) \rangle}{\langle \Omega, \mathcal{H}_0, \eta_0, \gamma_0, s_1;\ s_2 \rangle \Downarrow_i \langle \mathcal{H}_1, \eta_1, \gamma_1, \mathit{Ret}(v) \rangle}$$

Function Call

$$\frac{\begin{array}{c}\Omega(g) = (x_1,\ ...,\ x_n) \to s \qquad [\![e_i]\!]_\gamma = v_i \mid_{i=1}^{n} \\ \gamma'' = [x_i \mapsto v_i \mid_{i=1}^{n}] \qquad \langle \Omega, \mathcal{H}, \eta, \gamma'', s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma', \mathit{Ret}(v) \rangle\end{array}}{\langle \Omega, \mathcal{H}, \eta, \gamma, x := g(e_1,\ ...,\ e_n) \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma[x \mapsto v], \phi \rangle}$$

Return

$$\frac{[\![e]\!]_\gamma = v}{\langle \Omega, \mathcal{H}, \eta, \gamma, \mathsf{return}\ e \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma, \mathit{Ret}(v) \rangle}$$

Variable Assignment

$$\frac{[\![e]\!]_\gamma = v}{\langle \Omega, \mathcal{H}, \eta, \gamma, x := e \rangle \Downarrow_i \langle \mathcal{H}, \eta, \gamma[x \mapsto v], \mathit{Cont} \rangle}$$

Figure 4.11: Operational Semantics - Function Calls and Returns: $\langle \Omega, \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma', \phi \rangle$

the arguments $e_1,\ ...,\ e_n$ and creates the new store γ'' for executing the function, exclusively containing the bindings between parameters and arguments. Afterwards, the semantics evaluates the body of the function, and then binds the returned value v to variable x within the original store γ.

4.4 Soundness Proof

In this section, we prove type safety for the type system formalized in Section 4.2 with respect to the semantics defined in Section 4.3. Essentially, we say that our type system is *safe* in that the execution of a well-typed TESL statement preserves state satisfiability. We provide a comprehensive definition of state satisfiability in Section 4.4.1. Following that, in Section 4.4.2, we present the main soundness proof, based upon a few properties of well-typed TESL programs introduced within this same section.

4.4.1 State Satisfiability

Recall Typed ECMA-SL states described in Section 4.3.1. In order to define state satisfiability, we make use of the following auxiliary definitions:

- **Value Satisfiability:** describes what it means for a value v to satisfy a given type τ under a heap typing environment $\mathcal{H}$ — written $v \vDash_{\mathcal{H}} \tau$;
- **Store Satisfiability:** describes what it means for a store γ to satisfy a given store typing environment Γ under a heap typing environment $\mathcal{H}$ — written $\gamma \vDash_{\mathcal{H}} \Gamma$;

- **Heap Satisfiability:** describes what it means for a heap η to satisfy a given heap typing environment $\mathcal{H}$ — written $\eta \vDash \mathcal{H}$.

We say that a state (η, γ) satisfies a typing environment $(\mathcal{H}, \Gamma)$ if, and only if, $\eta \vDash \mathcal{H}$ *and* $\gamma \vDash_{\mathcal{H}} \Gamma$. In the following, we use the notation $\eta, \gamma \vDash \mathcal{H}, \Gamma$ to represent the state satisfiability condition.

Definition 7 (Value Satisfiability)**.** A value v is said to satisfy a type τ with respect to a heap typing environment $\mathcal{H}$, written $v \vDash_{\mathcal{H}} \tau$, if and only if:

$$type_{\mathcal{H}}(v) \leq^{*} \tau.$$

The value satisfiability relation needs to ensure that a value v, given a heap typing environment $\mathcal{H}$, is compatible with τ. In other words, the type τ needs to be capable of storing a value v of type $type_{\mathcal{H}}(v)$ (Section 4.3). The $type_{\mathcal{H}}$ function returns the object type $\mathcal{H}(l)$ when v is a location l, and the simple primitive type of v in all other cases. Note that it is not enough to say that the type of v is a subtype of τ (i.e., $type_{\mathcal{H}}(v) \leq \tau$) because our subtyping relationship defined in Section 4.2.2 does not allow field covariance. As a result, a value `{ foo: 10 }` would not satisfy the type `{ foo: int }` since the types of field `foo` are not the same (`10` $\neq$ `int`). To solve this, we employ the extended subtyping relationship $\leq^{*}$ that also allows for field covariance. The typing rules for this relationship are defined in Appendix B.

Definition 8 (Store Satisfiability)**.** Given a heap typing environment $\mathcal{H}$, a store γ is said to satisfy a store typing environment Γ, written $\gamma \vDash_{\mathcal{H}} \Gamma$, if and only if:

- $dom(\gamma) = dom(\Gamma)$
- $\forall_{x \in dom(\Gamma)}\ \gamma(x) \vDash_{\mathcal{H}} \Gamma(x)$.

The store satisfiability relation requires that: (1) the domain of the store γ coincides with the domain of the store typing environment Γ; and (2) all values in γ satisfy their respective types in Γ. Note that this relation only requires the heap typing environment $\mathcal{H}$ to resolve the type of object locations at runtime.

Definition 9 (Heap Satisfiability)**.** A heap η is said to satisfy a heap typing environment $\mathcal{H}$, written $\eta \vDash \mathcal{H}$, if and only if:

- $dom(\eta) = dom(\mathcal{H})$
- $\forall_{l \in dom(\eta)}\ dom(\eta(l)) = dom(\mathcal{H}(l))$
- $\forall_{l \in dom(\eta)}\ \forall_{f \in dom(\eta(l))}\ type_{\mathcal{H}}(\eta(l, f)) = \mathcal{H}(l, f)$

The heap satisfiability relation requires that: (1) the domain of the heap η coincides with the domain of the heap typing environment Γ; (2) the fields of a given location l within the heap η coincide with the fields of the corresponding object type in $\mathcal{H}(l)$; and (3) the value of each field f in a location l within the heap η matches its respective type in the $\mathcal{H}$.

4.4.2 Type Safety

As mentioned in the beginning of the section, we claim that our type system is *safe* in that the execution of a well-typed statement consistently preserves state satisfiability. In other words, if a statement s of a function g is typeable with respect to a given global typing context Δ and store typing environments Γ and Γ', written $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, and if one executes s in a state (η, γ) such that $\eta, \gamma \vDash \mathcal{H}, \Gamma$, for a given heap typing environment $\mathcal{H}$, then a completed execution will produce a state satisfying the final heap and store typing environments. Formally, this can be expressed as:

$$\eta, \gamma \vDash \mathcal{H}, \Gamma \ \wedge\ g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\} \ \wedge\ \langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle \implies \eta', \gamma' \vDash \mathcal{H}', \Gamma'$$

To ensure the safety of our type system, we rely on certain auxiliary properties regarding: (1) the soundness of expression typing; (2) the soundness of conditional type refinements; and (3) the soundness of pattern binding. In the remaining section, we describe these properties and subsequently present the main soundness proof of the type system.

Well-Typed Expressions In order to establish type safety for TESL statements, we first need to establish type safety for TESL expressions. This involves demonstrating that if an expression e is typed with τ within a store typing environment Γ, and if the evaluation of e in a store γ satisfying Γ results in the value v, then v must satisfy τ. Lemma 1 formally establishes this property.

Lemma 1 (Type Safety for Expressions)**.** For all store γ, store typing environments Γ, heap typing environment $\mathcal{H}$, and expression e, it holds that:

$$\gamma \vDash_{\mathcal{H}} \Gamma \ \wedge\ \Gamma \vdash e : \tau \ \wedge\ [\![e]\!]_\gamma = v \implies v \vDash_{\mathcal{H}} \tau$$

Proof. The proof follows by induction on the structure of e. Assume that $\gamma \vDash_{\mathcal{H}} \Gamma$ (**H1**), $\Gamma \vdash e : \tau$ (**H2**), and $[\![e]\!]_\gamma = v$ (**H3**). Therefore, we have that:

[VALUE] $e = v'$ (**H4**). From **H2** and **H4**, it follows that:

- **I1.1**: $\tau = type(v')$

From **H3** and **H4**, it follows that:

- **I2.1**: $v' = v$

From **I1.1** and **I2.1**, it follows that:

- **I3.1**: $\tau = type(v)$

Equation **I3.1** establishes the result.

[VARIABLE] $e = x$ (**H4**). From **H2** and **H4**, it follows that:

- **I1.1**: $\Gamma(x) = \tau$

From **H3** and **H4**, it follows that:

- **I2.1**: $\gamma(x) = v$

Equations **I1.1**, **I2.1**, and **H1** establish the result.

[UNARY OPERATION] $e = \ominus\ e'$ (**H4**). From **H2** and **H4**, it follows that there exists $\tau' \in T$ such that:

- **I1.1**: $\Gamma \vdash e' : \tau'$
- **I1.2**: $\ominus(\tau') = \tau$

From **H3** and **H4**, it follows that there exists $v' \in Val$ such that:

- **I2.1**: $[\![e']\!]_\gamma = v'$
- **I2.2**: $\underline{\ominus}(v') = v$

Applying the induction hypothesis to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $v' \vDash_{\mathcal{H}} \tau'$

Applying the definition of UNARY OPERATION to **I3.1**, we conclude that:

- **I4.1**: $\underline{\ominus}(v') \vDash_{\mathcal{H}} \ominus(\tau')$

Equations **I1.2**, **I2.2**, and **I4.1** establish the result.

[BINARY OPERATION] $e = e_1 \oplus e_2$ (**H4**). From **H2** and **H4**, it follows that there exists $\tau_1, \tau_2 \in T$ such that:

- **I1.1**: $\Gamma \vdash e_1 : \tau_1$
- **I1.2**: $\Gamma \vdash e_2 : \tau_2$
- **I1.3**: $\oplus(\tau_1, \tau_2) = \tau$

From **H3** and **H4**, it follows that there exists $v_1, v_2 \in Val$ such that:

- **I2.1**: $[\![e_1]\!]_\gamma = v_1$
- **I2.2**: $[\![e_2]\!]_\gamma = v_2$
- **I2.3**: $\underline{\oplus}(e_1, e_2) = v$

Applying the induction hypothesis to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $v_1 \vDash_{\mathcal{H}} \tau_1$

Applying the induction hypothesis to **I1.2**, **I2.2**, and **H1**, we conclude that:

- **I4.1**: $v_2 \vDash_{\mathcal{H}} \tau_2$

Applying the definition of BINARY OPERATION to **I3.1** and **I4.1**, we conclude that:

- **I5.1**: $\underline{\oplus}(v_1, v_2) \vDash_{\mathcal{H}} \oplus(\tau_1, \tau_2)$

Equations **I1.3**, **I2.3**, and **I5.1** establish the result.

[TYPE CASTING] $e = e'$ as τ **H4** From **H2** and **H4**, it follows that there exits $\tau' \in T$ such that:

- **I1.1**: $\Gamma \vdash e' : \tau'$
- **I1.2**: $\tau' \leq \tau$

From **H3** and **H4**, it follows that:

- **I2.1**: $[\![e']\!]_\gamma = v$

Applying the induction hypothesis to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $v \vDash_{\mathcal{H}} \tau'$
- **I3.2**: $v \leq^* \tau'$

By correctness of extended subtyping, equations **I1.2** and **I3.2** establish the result. □

Well-Typed Conditional Type Refinements In order to establish type safety for `if-then-else` and `while` statements, we first need to establish type safety for the Conditional Type Refinement (CTR) mechanism. Recall that this mechanism is employed to refine the type of a variable based on the variable's usage within conditional expressions, depending on whether the associated expression evaluates to true or false. Proving type safety for the CTR mechanism involves demonstrating that if an expression e evaluates to true under a store γ satisfying Γ, and if the CTR mechanism generates Γ' by refining the types in Γ based on e, then γ must also satisfy Γ'. Similarly, if e evaluates to false, γ must satisfy a Γ' generated based on $\neg e$. Lemma 2 formally establishes type safety for the two cases of this property.

Lemma 2 (Type Safety for Conditional Type Refinements)**.** For all heap η, store γ, heap typing environment $\mathcal{H}$, store typing environments Γ and Γ', and expression e, it holds that:

$$\begin{aligned}\eta, \gamma \vDash \mathcal{H}, \Gamma \wedge [\![e]\!]_\gamma = true \ \ \wedge \beta_\Gamma(e) = \Gamma' \quad &\Longrightarrow \quad \eta, \gamma \vDash \mathcal{H}, \Gamma' \\ \eta, \gamma \vDash \mathcal{H}, \Gamma \wedge [\![e]\!]_\gamma = false \wedge \beta_\Gamma(\neg e) = \Gamma' \quad &\Longrightarrow \quad \eta, \gamma \vDash \mathcal{H}, \Gamma'\end{aligned}$$

(The proof follows by induction on the structure of e)

Well-Typed Pattern Binding In order to establish type safety for `match-with` statements, we first need to establish type safety for the Type Pattern Binding (TPB) mechanism. Recall that this mechanism is employed to update the program store with the variables specified by the matched pattern. Proving type safety for the TPB mechanism involves demonstrating that if an object value o satisfying the sigma case σ of the sigma type τ is matched against a pattern ψ under the store γ satisfying Γ, and if the TPB mechanism generates Γ' by extending Γ with the bindings of ψ, then the new store γ' must satisfy Γ'. Lemma 3 formally establishes type safety for this property.

Lemma 3 (Type Safety for Pattern Binding)**.** For all heap η, stores γ and γ', heap typing environment $\mathcal{H}$, store typing environments Γ and Γ', pattern ψ, object value o, object type σ, and sigma type τ, it holds that:

$$\begin{gathered}\eta, \gamma \vDash \mathcal{H}, \Gamma \ \wedge \ match(\psi, o, \gamma) = \gamma' \ \wedge \ \Psi^{\alpha,\tau}(\psi, \sigma, \Gamma) = \Gamma' \ \wedge \ \tau = \Sigma[\alpha]\, \{...,\ \sigma,\ ...\} \ \wedge \ o \leq^* \sigma[{}^{\alpha}/_{\tau}] \\ \Longrightarrow \quad \eta, \gamma' \vDash \mathcal{H}, \Gamma'\end{gathered}$$

(The proof follows by induction on the number of elements of ψ)

Type Safety for Typed ECMA-SL Theorem 1 states that the type system for Typed ECMA-SL satisfies the type safety property introduced at the beginning of this section. To prove this theorem, we rely on the three lemmas introduced earlier. In the following, we prove the theorem for some semantic rules from the intra-procedural fragment of the language. The remaining statements are addressed in Appendix A.2.

Theorem 1 (Soundness - Type Safety)**.** For all heaps η, η', stores γ, γ', heap typing environments $\mathcal{H}, \mathcal{H}'$, store typing environments Γ, Γ', function g, global typing context Δ, and statement s it holds that:

$$\eta, \gamma \vDash \mathcal{H}, \Gamma \ \wedge \ g, \Delta \vdash \{\Gamma\} \ s \ \{\Gamma'\} \ \wedge \ \langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle \implies \eta', \gamma' \vDash \mathcal{H}', \Gamma'$$

Proof. The proof follows by induction on the derivation of the judgement $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$. Assume that $\eta, \gamma \vDash \mathcal{H}, \Gamma$ (**H1**), $g, \Delta \vdash \{\Gamma\} \ s \ \{\Gamma'\}$ (**H2**), and $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$ (**H3**). Therefore, we have that:

[SKIP] $s = \mathsf{skip}$ (**H4**). From **H2** and **H4**, it follows that:

- **I1.1**: $\Gamma' = \Gamma$

From **H3** and **H4**, it follows that:

- **I2.1**: $\mathcal{H}' = \mathcal{H}$
- **I2.2**: $\eta' = \eta$
- **I2.3**: $\gamma' = \gamma$

Equations **I1.1**, **I2.1**, **I2.2**, **I2.3**, and **H1** establish the result.

[FIELD LOOKUP] $s = x := e.f$ (**H4**). From **H2** and **H4**, it follows that there exists $\sigma \in TObj$ and $\tau_f \in T$ such that:

- **I1.1**: $\Gamma \vdash e : \sigma$
- **I1.2**: $ft(\sigma, f) = \tau_f$
- **I1.3**: $\Gamma' = \Gamma[x \mapsto \tau_f]$

From **H3** and **H4**, it follows that there exists $l \in Loc$ and $v \in Val$ such that:

- **I2.1**: $[\![e]\!]_\gamma = l$
- **I2.2**: $\eta(l, f) = v$
- **I2.3**: $\gamma' = \gamma[x \mapsto v]$
- **I2.4**: $\mathcal{H}' = \mathcal{H}$
- **I2.5**: $\eta' = \eta$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $l \vDash_{\mathcal{H}} \sigma$
- **I3.2**: $\mathcal{H}(l) \leq^* \sigma$

Applying the correctness of covariant object subtyping to **I1.2** and **I3.2**, we conclude that:

- **I4.1**: $ft(\mathcal{H}(l), f) \leq^* \tau_f$

Applying the typing rules of the ft function to **H1**, we conclude that:

- **I5.1**: $type_{\mathcal{H}}(\eta(l, f)) = ft(\mathcal{H}(l), f)$

From **I2.2**, **I4.1**, and **I5.1**, it follows that:

- **I6.1**: $type_{\mathcal{H}}(v) \leq^* \tau_f$
- **I6.2**: $v \vDash_{\mathcal{H}} \tau_f$

From **I1.3**, **I2.3**, **I6.2**, and **H1**, it follows that:

- **I7.1**: $\gamma' \vDash_{\mathcal{H}} \Gamma'$

Equations **I2.4**, **I2.5**, **I7.1**, and **H1** establish the result.

[IF-THEN-ELSE (TRUE)] $s = \mathsf{if}\,(e)\{s_1\}\,\mathsf{else}\,\{s_2\}$ (**H4**) and $[\![e]\!]_\gamma = true$ (**H5**). From **H2** and **H4**, it follows that there exists $\tau_e \in T$, and store typing environments Γ_1, Γ_2, Γ_1', and Γ_2', such that:

- **I1.1**: $\Gamma \vdash e : \tau_e$
- **I1.2**: $\tau_e \leq bool$
- **I1.3**: $\beta_\Gamma(e) = \Gamma_1$
- **I1.4**: $\beta_\Gamma(\neg e) = \Gamma_2$
- **I1.5**: $g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma_i'\}\ |_{i=1}^{2}$
- **I1.6**: $\Gamma' = \Gamma_1' \sqcup \Gamma_2'$

From **H3** and **H4**, it follows that:

- **I2.1**: $\langle \mathcal{H}, \eta, \gamma, s_1 \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the Lemma 2 (*Type Safety for Conditional Type Refinements*) to **H1**, **H5**, and **I1.3**, we conclude that:

- **I3.1**: $\eta, \gamma \vDash \mathcal{H}, \Gamma_1$

Applying the induction hypothesis to **I1.5**, **I2.1**, and **I3.1**, we conclude that:

- **I4.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma_1'$

By correctness of union subtyping, equations **I1.6** and **I4.1** establish the result.

[MATCH-WITH] $\mathsf{match}\,e\,\mathsf{with}\,\{\psi_i \to s_i\ |_{i=1}^{n}\}$ (**H4**) From **H2** and **H4**, it follows that there exists $\tau \in T$, $\sigma_1,\ ...,\ \sigma_n \in TObj$, and store typing environments $\Gamma_1,\ ...,\ \Gamma_n, \Gamma_1',\ ...,\ \Gamma_n'$, such that:

- **I1.1**: $\Gamma \vdash e : \tau$
- **I1.2**: $\tau = \Sigma[\alpha]\,\{\sigma_1,\ ...,\ \sigma_n\}$
- **I1.3**: $\Psi^{\alpha,\tau}(\psi_i, \sigma_i, \Gamma) = \Gamma_i\ |_{i=1}^{n}$
- **I1.4**: $g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma_i'\}\ |_{i=1}^{n}$
- **I1.5**: $\Gamma' = \sqcup\{\Gamma_1',\ ...,\ \Gamma_n'\}$

From **H3** and **H4**, it follows that there exists $j \in [1, n]$, $l \in Loc$, and a store γ_j, such that:

- **I2.1**: $[\![e]\!]_\gamma = l$
- **I2.2**: $match(\psi_j, \eta(l), \gamma) = \gamma_j$
- **I2.3**: $\langle \mathcal{H}, \eta, \gamma_j, s_j \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $l \vDash_{\mathcal{H}} \tau$
- **I3.2**: $\mathcal{H}(l) \leq^* \tau$

Applying the correctness of sigma folding subtyping to **I1.2** and **I3.2**, we conclude that there exists $\sigma' \in TObj$ such that:

- **I4.1**: $\sigma' \in \{\sigma_1,\ ...,\ \sigma_n\}$
- **I4.2**: $\mathcal{H}(l) \leq^* \sigma'[\tau/\alpha]$

Applying the semantics of the `match-with` statement to **I1.3**, **I2.2**, **I4.2**, and **H1**, it follows that

- **I5.1**: $\eta(l) \leq^* \sigma_j[\tau/\alpha]$

Applying the Lemma 3 (*Type Safety for Pattern Binding*) to **H1**, **I1.2**, **I1.3**, **I2.2**, and **I5.1**, we conclude that:

- **I7.1**: $\eta, \gamma_j \vDash \mathcal{H}, \Gamma_j$

Applying the induction hypothesis to **I1.4**, **I2.3**, and **I7.1**, we conclude that:

- **I8.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma'_j$

By correctness of union subtyping, equations **I1.5** and **I8.1** establish the result. □

5

Implementation

Contents

In this chapter, we discuss the real implementation of the type system for the ECMA-SL language. In Section 5.1, we provide a high-level overview of the system's architecture. Then, in Section 5.2, we enumerate the features of the real type system that were either omitted or simplified in the formal model of the system. Lastly, in Section 5.3, we analyse the system's error reporting mechanism, responsible for providing user feedback regarding type-related issues found within TESL programs.

5.1 Architecture

Recall the ESL execution pipeline depicted in Figure 2.6. Our work will be focused on the ECMASL2Core component. As previously described, this component is responsible for compiling ECMA-SL programs into Core ECMA-SL. Its modular architecture is portrayed in Figure 5.1. The figure is colour-coded: green for modules that were fully implemented in the context of this thesis, yellow for partially extended or modified modules, and red for modules that were not modified.

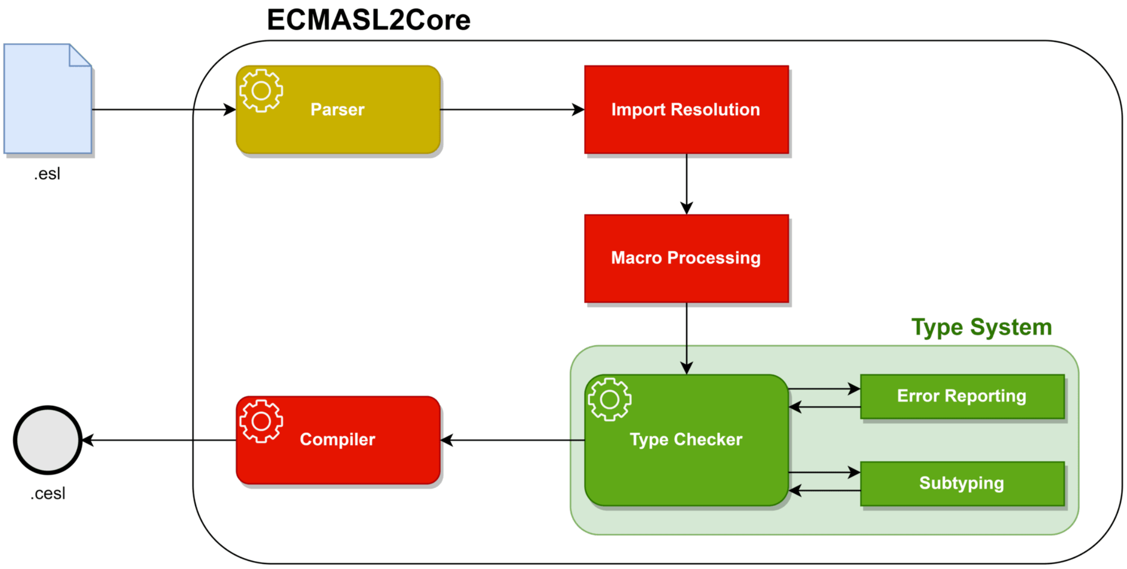


Figure 5.1: Architecture of the ECMA-SL compiler.

To compile an ESL program, the system starts by parsing the code and generating the program's AST. After this initial stage, the system resolves all imports and processes all macros. Macros are language constructs that resemble functions in structure but differ in how they operate. Rather than being evaluated at runtime, they are expanded by the compiler into their corresponding code at the locations where they are used. Macros allow developers to encapsulate code patterns within a single macro call, increasing maintainability and code reusability. We revisit ESL macros in Section 6.1.

Before finalizing the compilation into CESL, the system employs the new type checker module. This module contains about 2000 lines of code written in OCaml [20], accounting for approximately 15% of the total size of the ECMASL2Core component. The type checker recursively traverses the program's AST and evaluates the types of its elements, ensuring that they conform to the typing rules defined within the *Subtyping* submodule. Additionally, it relies on the *Error Reporting* submodule to generate error messages when necessary. This process ensures that the program adheres to the specified type annotations and provides effective feedback regarding type-related issues otherwise. After type checking the program, the compiler module transforms the program's AST into a Core ECMA-SL program.

5.2 Extra Features

Consider the formal model for the Typed ECMA-SL language described in Section 4.2. As previously stated, this model does not precisely adhere to the type system implemented in the context of this thesis, as several simplifications were required to facilitate the model's formalization and soundness proof. These differences include, among others, the simplification of some type system features, such as the Conditional Type Refinement (CTR) mechanism and the omission of certain types.

In this section, we describe the most important aspects that have been streamlined or excluded from the formal model of Typed ECMA-SL. For each of these aspects, we provide a concise description, a motivating example, and a rationale for their implementation in the type system. Note that the syntax used throughout the following examples slightly differs from that in the formal model of the language, as we are now employing the actual syntax of Typed ECMA-SL.

5.2.1 Controlled Type Updates

The formal model of Typed ECMA-SL describes a system with no explicit type declarations for variables. In this model, type annotations are exclusively employed in the definition of functions, type variables (α), and as an argument of the type casting expression. In contrast, the TESL implementation allows the declaration of a type for a variable during an assignment operation. While such annotations establish the type that all subsequent assignments to the variable must adhere to, the language maintains the flexibility to allow the redefinition of the variable's type through the use of a different type annotation.

```
1 x: int := 10;          /* typeof(x) => int */
2 x := 20;               /* [Valid] 20 is subtype of int */
3 x := "abc";            /* [Error] "abc" is not compatible with int */
4 x: string := "abc";    /* [Valid] typeof(x) => string */
```

Listing 5.1: Controlled type updates in Typed ECMA-SL.

Listing 5.1 illustrates controlled type updates in Typed ECMA-SL. In line 1, we declare `x` as an integer variable and initialize its value with the literal `10`. Line 2 highlights that we can assign any value to `x`, as long as the value conforms to the current type of the variable, which, in this context, is `int`. However, attempting an assignment that fails to meet this criterion results in the type error depicted in line 3. Finally, line 4 illustrates how the type of `x` can be redefined by employing a different type annotation during an assignment operation.

There are scenarios where updating the type of a variable is not possible. For example, consider a variable that was created outside a `while` loop. If we attempt to update its type while inside the looped block, the type system must generate a `BadTypeUpdate` error. Allowing for such an operation could potentially conflict with the typing rule for `while` statements, which requires the resulting typing environment to be compatible with the initial one.

While mainstream languages, such as C++ and Java, typically allow for the declaration of a variable's type, they often prohibit updates to it thereafter. In Typed ECMA-SL, controlled type updates serve as an important feature, as they introduce flexibility and expressiveness to the language. In particular, they allow us to accurately type functions within the reference interpreter, where the same variable is used at different points to store different types of values.

5.2.2 Advanced Conditional Type Refinement

When presenting the formal model of Typed ECMA-SL, we introduced the Conditional Type Refinement (CTR) mechanism. This mechanism allows the type of a variable to be refined according to how the variable is employed in conditional expressions within `if-then-else` and `while` statements. For instance, we can use the `typeof` operator inside a conditional expression to narrow down a union-type variable into one of its primitive cases. While the modelled mechanism works effectively, its implementation in Typed ECMA-SL is more sophisticated, as it supports additional types and logical operators.

```
1 x: 10 | string | boolean := 10;        /* typeof(x) => 10 | string | boolean */
2
3 if (typeof(x) = int || x = "abc") {}  /* typeof(x) => 10 | "abc" */
4 elif (x = true) {}                    /* typeof(x) => true */
5 else {}                               /* typeof(x) => string | false */
```

Listing 5.2: Conditional type refinements in an `if-then-else` statement.

Listing 5.2 illustrates some of the conditional refinements supported by the type system within an `if-then-else` statement. The program starts by creating the union-type variable `x`, which can either store the literal `10`, a string, or a boolean value. Lines 3 to 5 test the variable against several conditions, refining its type within each block of the `if-then-else` statement accordingly. In line 3, we evaluate whether the variable is holding an integer or the string literal `"abc"`. This test narrows the type of `x` within the `if` block to the union of the literal `10` (the only integer value that the variable can hold) and the literal `"abc"`. In line 4, we check if `x` is `true`, narrowing down its type within the `elif` block to the boolean literal `true`. Finally, in line 5, the system knows that `x` is neither `10` nor `"abc"`, since it would have passed the first test of the `if-then-else` statement otherwise. Additionally, it cannot be `true`, or it would have passed the second test of the `if-then-else` statement. Consequently, the only possible values that `x` may be holding are strings (excluding `"abc"`) and the boolean literal `false`.

Evidently, this system is highly expressive and useful when typing the reference interpreter. In the ES standard, there are multiple scenarios where we need to test values against multiple and complex conditions. One such example exists within the semantics of the `typeof` operator,[1] which needs to test a value against all JS primitive types and, if all tests are unsuccessful, assert that the value is an object.

[1] `typeof` operator - https://262.ecma-international.org/14.0/#sec-void-operator-runtime-semantics-evaluation.

5.2.3 Object Operations on Union Types

Some operations, such as field lookups, are specifically designed to work with object types. However, there are scenarios where it can be useful to perform these operations directly on union types. Take, for instance, a union of object types where all objects share a common field. In such cases, performing a field lookup on the shared field is a safe operation, given that the field exists in all objects. The resulting type of this operation is the union of all field types.

```
1 typedef FunctionObject_t := { internal: false, ... };
2 typedef InternalFunction_t := { internal: true, ... };
3
4 func: FunctionObject_t | InternalFunction_t := getFunc();
5 if (func.internal = true) {
6   /* call internal function */
7 } else {
8   /* call JS function */
9 }
```

Listing 5.3: Evaluation of different types of function calls in the reference interpreter.

Listing 5.3 simplifies the mechanism responsible for evaluating function calls within our reference interpreter. The interpreter supports two types of functions: (1) normal JS functions (line 1), stored as function objects as described by the ES standard; and (2) internal functions (line 2), used to store functions whose implementation resides in the interpreter's code itself (e.g., built-in functions like the `Array.prototype.push`). To distinguish between these two function types, the interpreter relies on the object's `internal` field. By inspecting its value (line 5), the interpreter can refine the type of `func`, and then execute the required steps to evaluate each function accordingly. Keep in mind that this conditional statement is only possible because the system supports field lookups on union type values.

The ability to conduct certain object operations directly on union types is valuable when handling complex unions of objects. In particular, union lookups offer a new approach for deconstructing union types, as shown by the previous example. While the ES standard does not mandate such usage, this feature introduces flexibility into Typed ECMA-SL programs, allowing developers to work with union types in a more versatile manner and enhancing the overall expressiveness of their code.

5.2.4 Sigma Types and Flexible Pattern Matching

Consider sigma types and the `match-with` statement described in the formal model of Typed ECMA-SL. In the type system's implementation, sigma types are defined as disjoint unions of object types, where each disjoint case includes a special field known as the *discriminant*. This field is responsible for ensuring the uniqueness of each disjoint case of the sigma type. Furthermore, rather than using the bound variable α to define a recursive sigma type, we create them by referencing the type alias (i.e., the type variable α) of the type currently being defined. In essence, sigma types are very similar to *variant types* found in modern functional languages, such as OCaml and Haskell.

Our implementation of the `match-with` offers greater flexibility and expressiveness compared to the one described for the formal model of Typed ECMA-SL. While the model required complete matches, where each disjoint case would appear once in the order in which they were defined within the sigma type, our implementation provides the ability to: (1) have multiple patterns for the same disjoint case; (2) define a default case; (3) detect incomplete matches; and (4) detect unused or invalid patterns.

```
 1 typedef JSStmt_t := sigma[type]
 2   | { type: "ExprStmt", expr: JSExpr_t }
 3   | { type: "IfStmt", ..., alternative: JSStmt_t | null }
 4   | ...
 5
 6 x: JSStmt_t := getStmt();
 7 match x with
 8 | { type: "IfStmt", alternative: null } -> {} /* if statement without an else block */
 9 | { type: "IfStmt", alternative: s2 } -> {}   /* if statement with an else block  */
10 | default -> {}                               /* all other statements */
```

Listing 5.4: Deconstruction of a JavaScript statement using the `match-with` construct.

Listing 5.4 shows a simplified declaration for the type of JS statements and a potential deconstruction for it using a `match-with`. One example of a JavaScript statement is the `if-then-else` (line 3). This statement stores an optional `alternative` field (i.e., the `else` block) which contains the code that will be executed when the test evaluates to false. If there is no `else` block within the `if-then-else` statement, then this field is set to `null`. In line 7, we match the JS statement against three different cases: (1) the `"IfStmt"` case without an `else` block; (2) the `"IfStmt"` case with an `else` block of type `JSStmt_t`, that gets assigned to variable `s2`; and (3) the default case, accounting for all other types of statements. Note that if lines 7 and 8 were reversed, the system would raise an `UnusedPatternCase` warning because all `if-then-else` statements would successfully match against the first pattern, rendering the second one unreachable. Furthermore, without a default case at the end of the `match-with` statement, the system would generate an `IncompleteMatch` error, as there would be no valid pattern for all other types of statements, such as the `"ExprStmt"`.

In the reference interpreter, there are multiple functions that rely on the `match-with` statement to deconstruct complex datatypes into their multiple cases. To properly type them, our type system needs to effectively handle complex patterns. Moreover, the ability to detect missing, invalid, and unused pattern cases is useful, as this functionality not only enhances the robustness of the code but also reduces the likelihood of bugs in complex pattern matching operations.

5.2.5 Additional Types

The types supported by the formal model of Typed ECMA-SL are insufficient for typing the values used throughout the ECMAScript standard. Furthermore, there are other types that, despite not having any associated value within the standard, are useful when typing the reference interpreters. In addition to the types described in the formal model of the language, the type system includes:

- The any type, which can be employed to bypass the type-checking mechanism since it stores values of any type and can also be assigned to a variable or field of any type. From an implementation perspective, all types are a subtype of any, and any is a subtype of all types.
- The void type, mainly as the return type of functions that perform some action or side effect but do not produce any meaningful value.
- The symbol type, which is used to type symbol values, such as 'NormalCompletion. ECMA-SL includes a special symbol datatype used to represent immutable atomic string values that are interned, meaning that two symbols with the same character content are considered the same value throughout the execution of a program.
- List types, which represent ordered collections of elements. Lists are homogeneous, meaning all elements within a list must be of the same type, and they are immutable, in that that they cannot be modified. For example, the list type [int] can be used to store a list of integers.
- Tuple types, which group together a fixed number of elements into a single value. They are also immutable, but unlike lists, tuples are heterogeneous, meaning they can have elements of different types. For example, the tuple type int * string can be used as the type of a pair containing an integer and a string value.

```
typedef JSBlock_t := { type: "BlockStmt", block: [JSStmt_t] }
```

Listing 5.5: Declaration of the type for a JavaScript block statement.

Listing 5.5 contains the type declaration for JavaScript block statements, JSBlock_t. In the ES standard, block statements are used to group multiple statements into a single compound statement. They are commonly found in control structures like loops and conditional statements. Essentially, we can perceive blocks as a list of statements; hence, we can type them with the list type [JSStmt_t].

These additional types are essential when typing the reference interpreter because their corresponding values frequently appear in the ES standard. The void type, despite not having any associated values, is also valuable when typing procedures within the reference interpreter that do not yield any result. These procedures are often associated with auxiliary functions that may not necessarily exist within the text of the standard. Finally, the any type is particularly useful during the interpreter's typing process. It allows developers to temporarily assign a type to a variable or field, enabling the interpreter to compile even in cases where it might not otherwise. They essentially serve as an additional mechanism to allow developers to gradually type the reference interpreters, allowing for changes to be made incrementally rather than all at once.

5.2.6 Syntactic Checks

In Section 2.3, we identified the lack of easy syntactic checks as one of the most significant limitations of the ECMA-SL language. Our implementation effectively addresses all the syntactic checks listed earlier, such as identifying unknown variables and functions, as well as ensuring that all control paths end with a `return` or `throw` statement. These mechanisms are available even for untyped programs.

```
1  typedef Obj_t := { foo: int, foo: string };   /* duplicate field error */
2
3  function main() {
4    x: int := unknownFun();    /* unknown function error */
5    y: int := unknownVar;      /* unknown variable error */
6    if (y = 10) {
7      return y
8    } else {
9      /* missing return statement */
10   }
11 }
```

Listing 5.6: Syntactic checks in Typed ECMA-SL.

Listing 5.6 illustrates some of the issues that can now be identified in Typed ECMA-SL programs. In this example, we start by creating the object type `Obj_t`, which contains two fields with the same name `foo`. This is considered a syntactical error, as objects cannot have two distinct fields with the same name. In line 4, we attempt to initialize variable `x` with the result of calling the undefined function `unknownFunc()`, resulting in an `UnknownFunction` error. The error encountered in line 5 is similar to the previous one, except that we instead attempt to reference the undefined variable `unknownVar`. Note that, even though an error was detected during the initialization of `y`, the type system still allows this variable to be referenced in lines 6 and 7 without producing another `UnknownVariable` error. Finally, if the program enters the second branch of the `if-then-else`, the `main()` function will end without a `return` or `throw` statement, which is not allowed according to the rules of the language.

The ability to identify syntactic errors represents a significant enhancement to the language. Without these mechanisms, such errors might remain hidden until runtime, potentially leading to bugs that are hard to locate. With these features in place, development in Typed ECMA-SL is expected to become considerably more efficient and straightforward.

5.2.7 Unchecked Function Signatures

As mentioned in Section 4.1.2, Typed ECMA-SL allows the declaration of unchecked function signatures. These signatures define the types for function parameters and return value without the need to type or even implement the body of the function. They offer two key advantages: (1) the ability to type function calls without having to type the function definition; and (2) the ability to call functions that are undefined at the moment of compilation without producing the `UnknownFunction` syntactical error (Section 5.2.6).

```
1 function Build_AST(): JSStmt_t;
2
3 function main() {
4   ast: JSStmt_t := Build_AST();
5   return JS_Interpreter_Program(ast)
6 }
```

Listing 5.7: Use of unchecked function signatures to call an undefined function.

Listing 5.7 shows a simplified version of the ECMARef's entry point. The interpreter starts by calling the `Build_AST()` function to create the AST of the JavaScript program that will be executed. There are two ways of achieving this: (1) generate the `Build_AST()` function in ESL, add it to the interpreters' code, and then compile everything to CESL; or (2) generate the `Build_AST()` function in CESL and then merge it with the CESL version of the interpreter. Naturally, the second approach is significantly faster than the first, as it does not require the interpreter's code to be recompiled every time we execute a JS program. However, compiling the interpreter without defining the `Build_AST()` function raises the `UnknownFunction` error; hence the need for the function signature in line 1.

Unchecked function signatures are an essential feature of Typed ECMA-SL. They offer several advantages, including the ability to gradually type the reference interpreter by providing temporary typed function signatures before the actual functions are fully typed. Furthermore, they also enable significant efficiency optimizations for the overall project, such as the one described in the previous example. However, it is important not to use them excessively when typing the interpreter, as they can potentially introduce bugs by leaving parts of the code unchecked.

5.3 Error Reporting

The primary goal of any type system is to offer developers informative feedback regarding type errors. To accomplish this goal effectively, it is essential to have a robust error reporting mechanism. This mechanism aids developers in finding type-related issues, thereby speeding up the development process. In particular, our objective is to ensure that developers can immediately identify the location, cause, and solution for any type error simply by examining the error's description. In essence, Typed ECMA-SL error messages try to resemble those found in modern high-level languages and include:

- **Error Category:** Similarly to modern languages like C++ and Rust, TESL distinguishes two error categories: type errors and type warnings. The distinction is subtle, as both errors and warnings can potentially lead to crashes at runtime. However, while type errors inevitably lead to crashes, type warnings may not necessarily cause issues. As a result, we allow programs to be compiled with warnings, providing some flexibility to developers. For instance, an incomplete `match-with` statement (Section 5.2.4) leads to a crash when attempting to process one of its missing cases. Nevertheless, the programmer may know that certain cases within the respective sigma type are impossible based on the code's structure, and thus, we may allow the compilation of the program. Note that, by default, the type system does not compile programs with warnings. To change this behaviour, programs need to be compiled with the `tesl-ignore-warns` flag.

- **Cascading Error Descriptions:** Primary error messages may be followed by a series of simpler messages that provide additional context regarding the underlying cause of the error. For example, the assignment `x: { foo: int } := { foo: "abc" }` results in the following error description:

  TypeError: Value of type `'{ foo: "abc" }'` is not assignable to type `'{ foo: int }'`.

  Caused by: Types of property `'foo'` are incompatible.

  Caused by: Value of type `'string'` is not assignable to type `'int'`.

- **Complete Error Location:** This includes the name of the source file, the line number, and the starting and ending characters within the line where the error occurred.

- **Error Source:** This includes a snippet of the code where the error occurred. If the system can identify the segment of the code responsible for the error, then it will be highlighted.

```
1 function f(): string | boolean;
2
3 x: { foo: int | float } := { foo: f() };
```

(a) Typed ECMA-SL program.

```
TypeError: Value of type '{ foo: (string | boolean) }' is not assignable to type
'{ foo: (int | float) }'.
Caused by: Types of property 'foo' are incompatible.
Caused by: Value of type '(string | boolean)' is not assignable to type '(int | float)'.
Caused by: Value of type 'string' is not assignable to type '(int | float)'.
File "main.esl", line 3, characters 35-37:
3 |   x: { foo: (int | float) } := { foo: f() }
                                        ^^^
```

(b) Output of the compilation.

Figure 5.2: Compilation of a Typed ECMA-SL program with an assignment type error.

Figure 5.2 depicts a program with an assignment type error, along with the corresponding error message generated by the TESL compiler. Consider the code snippet in Figure 5.2a. In line 3, we attempt to initialize variable `x` with the object value `{ foo: f() }`. Since `f` refers to the function signature `f: () → string | boolean` declared in line 1, the type of the new object is determined to be `{ foo: string | boolean }`. However, this type is not compatible with the one declared for variable `x`, leading to the type error shown in Figure 5.2b. Note that our error reporting mechanism is able to find and present the causes behind the incompatibility between the object types `{ foo: string | boolean }` and `{ foo: int | float }`. Additionally, the compiler is able to identify and highlight the element of the statement responsible for the type error, which in this case is the `f()` call.

```
1 typedef Sigma_t := sigma[type]
2   | { type: "foo", foo: int }
3   | { type: "bar", bar: int };
4
5 y: Sigma_t := { type: "foo", foo: 10 };
6 match y with
7 | { type: "foo", foo: fd } -> {}
8 | { type: "foo", foo: 10 } -> {}
9 | default -> {};
```

(a) Typed ECMA-SL program.

```
Type Warning: This pattern-matching case is unused.
File "main.esl", line 8, characters 0-33:
8 |    | { type: "foo", foo: 10 } -> { ...
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

(b) Output of the compilation.

Figure 5.3: Compilation of a Typed ECMA-SL program with an unused `match-with` pattern.

Figure 5.3 illustrates a `match-with` statement containing an `UnusedPattern` warning. In line 6 of the code snippet (Figure 5.3a), we perform a matching operation on variable `y` of type `Sigma_t`. This `match-with` statement contains three different patterns: (1) the pattern of line 7 that accepts all instances of the sigma's first disjoint case `{ type: "foo" }` and assigns the value of the `foo` field to variable `fd`; (2) the pattern of line 8 that also accepts instances of the first disjoint case, but only when `foo` is holding the integer value `10`; and (3) the default pattern of line 9 that accepts all other cases. Note that all instances of the first disjoint case `{ type: "foo" }` are caught by the first pattern, rendering the second pattern unused and unreachable. The Typed ECMA-SL compiler recognizes this as a non-breaking issue and consequently generates the corresponding type warning shown in Figure 5.3b.

6
Evaluation

Contents

In this chapter, we evaluate the type system developed for the Typed ECMA-SL language. Recall that the primary goal of Typed ECMA-SL is to be used for typing the reference interpreters of the ESL project, particularly ECMARef6. Given the size and complexity of this task, we started by using Typed ECMA-SL to create a typed simplified interpreter for JavaScript. This interpreter supports the most fundamental JS constructs and is structured similarly to the reference interpreters. Successfully typing the simplified interpreter suggests that it will be also possible to use Typed ECMA-SL to fully type the reference interpreters. We describe this simplified interpreter and its typing process in Section 6.1. Following that, in Section 6.2, we identify a number of problems that our type system is able to identify within the ECMARef6 interpreter without adding type annotations to it.

6.1 Simplified JavaScript Interpreter

To understand the usefulness of the simplified JS interpreter in a broader context, it is helpful to revisit the role that JS reference interpreters can play in the analysis of JavaScript programs. While certain types of analysis require a highly precise model of the JavaScript language, such as the one implemented by the ECMARef6 interpreter, others do not demand such precision (e.g., the analysis of programs that do not use unusual language features). Naturally, greater precision in more complex models comes at a higher cost. Alternatively, one can use less precise but simpler language models. This is where our simplified interpreter comes into play, allowing us to abstract much of the JavaScript complexity, albeit at the cost of less precise analysis results.

In the context of this project, the simplified interpreter provides a good platform for assessing the effectiveness of our type system, as it allows us to test complex typing features required by the reference interpreters without having to type thousands of lines of code. Despite its relative simplicity compared to the complete reference interpreter, our simplified interpreter is able to handle the most common JS expressions and statements. They include, among others, variable declarations and assignments, unary and binary operators, the most used control flow statements, function calls, and objects. Additionally, the interpreter supports a simplified implementation of the built-in Array library, providing mechanisms to initialize arrays as well as operations for adding, accessing, and updating array elements.

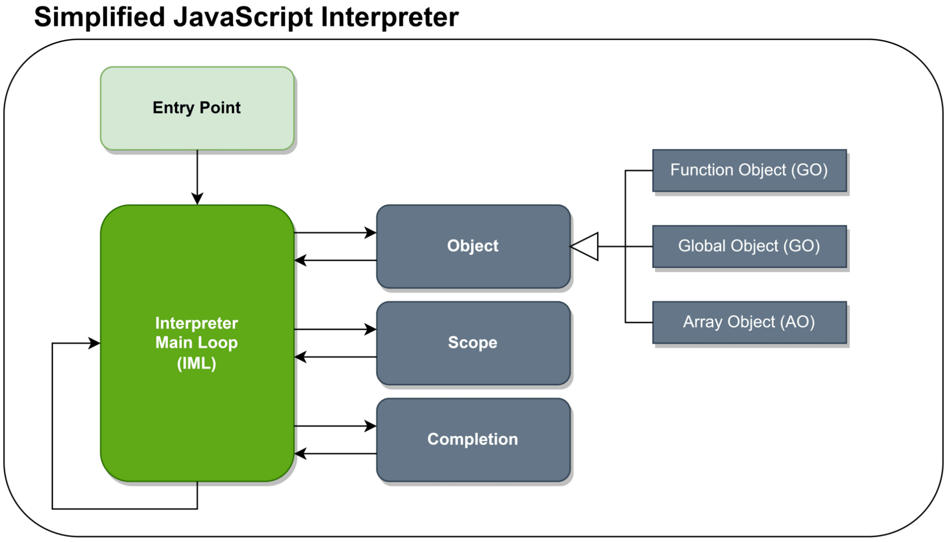


Figure 6.1: Modular architecture of the Simplified JavaScript interpreter.

Figure 6.1 depicts the architecture of the simplified interpreter developed within the context of this thesis. Our interpreter was written in Typed ECMA-SL, using a modular approach in which each module offers a well-defined interface for its functionality. This design approach provides the flexibility to change the implementation of a module without compromising the overall functionality of the interpreter. A fundamental module of the system is the Object module, which implements the behaviour of JS objects. This module serves as the foundation for others, including: (1) the Function Object (FO), which stores the components of a function; (2) the Global Object (GO), where internal built-in functions are stored; and (3) the Array Object (AO), containing our partial implementation of the built-in Array library.

The core functionality of the main interpreter resides within the Interpreter Main Loop (IML). This component comprises the primary functions responsible for deconstructing and evaluating the elements within the JavaScript Abstract Syntactic Tree, including:

- JS literals, such as numbers, strings, booleans, and objects;
- JS expressions, such as the `AssignmentExpression`, `BinaryExpression`, and `CallExpression`;
- JS statements, such as the `BlockStatement`, `IfStatement`, and `BreakStatement`; and
- JS declarations, such as the `VariableDeclaration` and `FunctionDeclaration`.

```
typedef JSStmt_t := sigma[type]
  | { type: "BlockStatement", body: [JSStmt_t] }
  | { type: "IfStatement", test: JSExpr_t, consequent: JSStmt_t, alternate?: JSStmt_t }
  | { type: "BreakStatement", label: null }
  | ...
```

(a) Type declaration for JavaScript statements.

```
typedef JSObject_t := {
  __js_props__: { *: JSVal_t },
  __proto__: JSObject_t | null,
  resolveProperty: "resolveObject",
  updateProperty: "updateObject"
}
```

(b) Type declaration for JavaScript objects.

```
typedef JSScope_t := {
  store: { *: JSVal_t },
  parent: JSScope_t | null,
  thisBinding: JSObject_t,
  globalObject: JSObject_t
}
```

(c) Type declaration for JavaScript scopes.

Figure 6.2: Type declarations in the Simplified JavaScript Interpreter.

Figure 6.2 illustrates three type declarations used to annotate the simplified JS interpreter. Figure 6.2a contains a partial definition of the type for JS statements, `JSStmt_t`. This declaration resorts to a sigma type (discussed in Section 5.2.7), where each disjoint case represents a different JS statement. In particular, we provide the declarations for the following JS statements: (1) `BlockStatement`, which uses the list type `[JSStmt_t]` to type its `body` field; `IfStatement`, which resorts to the optional typing operator `?:` to type its optional `alternative` field; and (3) `BreakStatement`, where its `label` field is typed as `null`, since the interpreter does not support labelled statements.

Figure 6.2b provides the type declaration for JS objects, `JSObject_t`. In our implementation, we model each JS object with two ESL objects: one of type `JSObject_t` for storing its internal properties, and another of type `{ *: JSVal_t }` for storing its named properties. The latter is modelled with a summary type that allows the object to store an arbitrary number of `JSVal_t` properties, and it is accessed via `__js_props__`. Additionally, the internal object has the following properties: (1) `__proto__`, of type `JSObject_t | null`, holding the internal prototype of the object, if any; (2) `resolveProperty`, of type `"resolveProperty"`, holding the name of the function used to resolve object properties; and (3) `updateProperty`, of type `"updateProperty"`, holding the name of the function used to update object properties. Illustratively, the structure of the object `{ foo: 10, bar: "abc" }` is given by Figure 6.3.

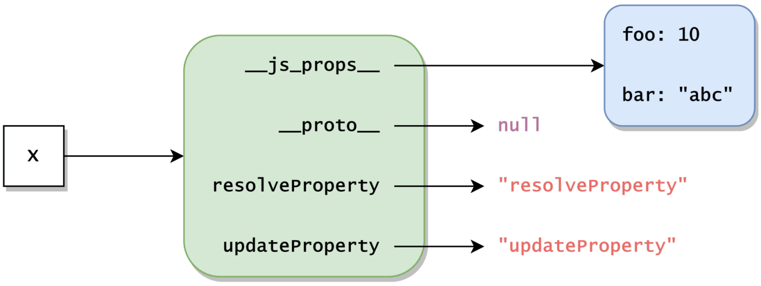


Figure 6.3: Representation of a JavaScript object in the Simplified JavaScript Interpreter.

Lastly, Figure 6.2c provides the type declaration for JS scopes, `JSScope_t`. Similarly to objects, scopes include a map between identifiers and JS values (`store` property). Additionally, they can be associated with a parent scope, which indicates the scope where the function was originally defined, and they also maintain a reference to the current *this* environment[1] and global object[2] to resolve the `this` expression and access JS built-in functions defined within the interpreter, respectively.

```
function interpreterStmt(scope: JSScope_t, s: JSStmt_t): JSCompletion_t {
  match s with
  | { type: "ReturnStatement", argument: arg } -> {
      retVal: JSCompletion_t := interpreterExpr(scope, arg);
      @checkAbruptCompletion(retVal);
      return ReturnCompletion(retVal)
    }
  | { type: "IfStatement", test: test, consequent: stmt1, alternate: stm2 } -> {
      testVal: JSCompletion_t := interpreterExpr(scope, test);
      @checkAbruptCompletion(testVal);
      if (typeof(testVal) != boolean) {
        raiseError ("Illegal test expression", testVal)
      };
      if (testVal) {
        return interpreter(scope, stmt1)
      } elif (!(stmt2 = null)) {
        return interpreterStmt(scope, stmt2)
      } else {
        return NormalCompletion('empty)
      }
    }
  | ...
}
```

Listing 6.1: Interpreter Main Loop (Statements) of the Simplified JavaScript Interpreter.

Listing 6.1 features a partial implementation of the `interpreterStmt(scope, s)` function within the Interpreter Main Loop. This function uses the `match-with` statement to deconstruct JS statements into their various types and evaluate each accordingly, returning a `JSCompletion_t`. In the JS specification, completions[3] are abstract data structures used to describe the runtime propagation of values and control

[1] *this* operator - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

[2] Global object - https://developer.mozilla.org/en-US/docs/Glossary/Global_object

[3] JavaScript completions - https://262.ecma-international.org/14.0/#sec-completion-record-specification-type

flow. The language describes normal completions for regular data propagation and abrupt completions for exceptional non-local transfers of control. The latter includes the return, break, continue, and throw completions associated with the `return`, `break`, `continue`, and `throw` statements, respectively.

The first statement in Listing 6.1 is the `return` statement. To process this statement, we start by evaluating its argument using the JS expression interpreter (line 4). Then, we need to check the type of the resulting completion, since evaluating the expression could have resulted in an error. To do so, we resort to the `@checkAbruptCompletion(c)` macro (line 5) that, once expanded, tests if `testVal` is a normal completion and extracts its value. The implementation of the macro is depicted in Listing 6.2. If `testVal` is a normal completion, then we encapsulate its value on a return completion and return it; otherwise, we simply return the obtained abrupt completion.

```
macro checkAbruptCompletion(c: JSCompletion_t) {
  if (getCompletionType(c) = 'Normal) {
    c: JSVal_t := getCompletionValue(c)
  } else {
    return c
  }
}
```

Listing 6.2: ESL macro to test the type and extract the value of normal completions.

The second statement shown in Listing 6.1 is the `if-then-else` statement. In this statement, we start by evaluating the conditional `test` expression and testing if the resulting value is a boolean (lines 9 to 13). If so, there are three possible evaluation scenarios: (1) if `test` evaluates to `true`, we execute the `consequent` statement; (2) if `test` evaluates to `false` and there is an `else` block, we evaluate the `alternate` statement; and (3) if `test` evaluates to `false` and there is no `else` block, we terminate the `if-then-else` statement with a normal completion without any meaningful value.

Module / **File**	#Lines	#Functions	#Typedefs	#Type Annotations	Typing Effort
Completion	60	8	2	16	31.42%
Object	38	3	2	10	56.19%
Function Object	29	1	2	8	86.24%
Global Object	14	2	0	4	7.56%
Array Object	48	3	1	14	27.20%
Internal Function	36	5	1	15	43.44%
Scope	85	8	1	27	27.08%
InterpreterExpr	347	7	23	65	37.45%
InterpreterStmt	131	1	12	10	65.84%
Entry Point	30	2	1	7	19.96%
9 Files	818	41	45	176	38.52%

Table 6.1: Effort metrics for the development and typing of the Simplified JavaScript Interpreter.

To understand the effort associated with typing our simplified interpreter for JavaScript, consider Table 6.1. This table shows the number of lines, functions, type definitions/signatures, and type annotations within each module of our simplified JS interpreter. Additionally, we include an extra column indicating the typing effort for each module. The typing effort corresponds to the percentual increase in size (number of characters) between the untyped module and its typed version. Note that the last row contains the totals for each column, except for typing effort which shows the average instead.

As shown in the table, the increase in size between the untyped interpreter and its typed version is, on average, 38.52%. Upon inspecting the implementation of each module, we consider this to be a pessimistic estimation of the effort required to type the reference interpreters, since the main contributors to the typing effort are type declarations rather than type annotations. This observation is corroborated by the global object module, which has the lowest typing effort and does not contain any type definitions. We deduce that the effort of typing the reference interpreters will be significantly smaller, as type definitions can be reused more often throughout a much larger codebase.

Type System Feature	#Uses	Example
Controlled Type Updates	1	Extract the value from a completion to the same variable (Listing 6.2).
Conditional Type Refinements	10	Check if the object prototype is `null` and use it otherwise (`Object.resolveObject`).
Object Operations on Unions	2	Differentiating between internal functions and JS functions (Listing 5.3).
Sigma Types	2	Type definition for JS statements (Figure 6.2a).
Pattern Matching with Sigmas	8	Main interpreter loop for JS statements (Figure 6.2a.)
Additional Types	42	Definition of the type for the body of a `BlockStatement` (Listing 5.5).
Unchecked Function Signatures	1	Signature of the `buildAst()` function (Listing 5.7).
Optional Typing	1	Type declaration for the alternate block in `if-then-else` statements (Figure 6.2a).
Summary Types	2	Type declaration for stores (Figure 6.2c).
Recursive Types	29	Type declaration for JS statements (Figure 6.2a).

Table 6.2: Features of the type system used while typing of the Simplified JavaScript Interpreter.

Table 6.2 summarizes the key features of the TESL type system. It provides a count of occurrences for each feature within the typed version of our simplified JS interpreter, along with a brief example of the feature's usage. As you can see, every fundamental feature of the type system finds at least one application within the simplified interpreter. Additional types (in particular list types) and recursive types play a vital role when typing the interpreter, as they are constantly required by the JavaScript's AST. Essentially, they are needed to define the types of multiple JS expressions and statements. Conditional type refinements and pattern matching are also essential features throughout the interpreter.

6.2 Syntactical Issues in the ECMARef6

As mentioned in the beginning of this chapter, typing the ESL project reference interpreters, namely the ECMARef5 and ECMARef6, is a very time-consuming task that falls beyond the scope of this thesis. However, as discussed in Section 5.2.6, the TESL type system includes a mechanism to perform easy syntactic checks within an ESL program. These syntactic checks are associated with practices that, despite compilable, can potentially generate issues when executing the program. They include, among others, calling functions without providing all the required arguments, referencing undefined variables, and having execution paths that do not end with a `return` or `throw` statement. We can thus employ our type system to assess whether our reference interpreters are free of these syntactical issues.

Syntactical Error	Description	ECMARef5	ECMARef6
`DuplicateParam`	Function defined with two identically named parameters	-	-
`DuplicateFld`	Object defined with two identically named fields	-	-
`DuplicatePatFld`	Pattern defined with two identically named fields	-	-
`UnknownVar`	Referencing an uninitialized variable	19	27
`UnknownFunction`	Calling an undefined function	-	-
`NExpectedArgs`	Calling a function with missing/extra arguments	1	6
`UnreachableCode`	Code that can never be reached	-	23
`OpenCodePath`	Execution path that does not return or throw	9	12

Table 6.3: Syntactical issues detected within the ECMARef5 and ECMARef6 interpreters.

Table 6.3 enumerates the most important syntactical checks performed by Typed ECMA-SL and, for each of them, tallies the number of occurrences detected in the ECMARef5 and ECMARef6 interpreters. We can use this information to check our implementation and determine whether these syntactical errors stem from issues within our code or if they are inherent to the JS standard. This analysis reveals four distinct error categories within our reference interpreter: unknown variables, calls with the wrong number of arguments, unreachable code, and open code paths. The most troublesome errors are references to unknown variables and calls with missing arguments. Even though the program can still run with these issues, it does so by assuming that missing variables and arguments have undefined values. While this behaviour might be expected in some scenarios, it may lead to subtle bugs that are hard to find.

Figure 6.4 illustrates one of the syntactical errors that the TESL type system is able to detect within the ECMARef6 interpreter. The error is found at the `SuperPropertyEvaluationA(scope, Expression)` function (Figure 6.4a), which is used to evaluate the JavaScript's `super` expression. This function starts by calling the `JS_Interpreter_Expr(Expression, scope)` function (line 10), but it fails to provide the second argument required by it (i.e., the `scope` parameter). This oversight results in a bug at runtime

```
1 function JS_Interpreter_Expr(Expression, scope) {
2   return JS_Interpreter_PrimaryExpression(Expression, scope)
3 };
4
5 ...
92 function SuperPropertyEvaluationA(scope, Expression) {
93     /* 1. Let propertyNameReference be the result of evaluating Expression. */
94     propertyNameReference := JS_Interpreter_Expr(Expression);
95     ...
96 }
```

(a) ECMARef6 implementation.

```
TypeError: Expected 2 arguments, but got 1.
File "ES6_interpreter/section 12/section_12.3.esl", line 94, characters 29-59:
94 |   propertyNameReference := JS_Interpreter_Expr(Expression)
                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

(b) Output of the compilation.

Figure 6.4: Syntactical error (missing arguments) within the ECMARef6 interpreter.

since `super` expression is being evaluated using an undefined scope. However, our type system can prevent this issue by finding this fault at compile time, generating the error message in Figure 6.4b.

The fact that we were able to detect errors by applying the TESL type to the untyped versions of the reference interpreters serves as additional evidence of the system's value. It also suggests that there may be other bugs within the interpreters that will become apparent once we start to introduce type annotations. While some of these errors may be due to implementation issues, others are likely problems within the JS specification itself.

7

Conclusions

Contents

As JavaScript continues to evolve, managing its standard is becoming increasingly difficult. We believe that the best approach for addressing the complexity of the ES standard is to model it as an executable JS specification. To do this, the ECMA-SL project introduces the ECMA-SL intermediate language, which serves as a foundation for implementing JS reference interpreters. The project includes two interpreters, ECMARef5 and ECMARef6, that adhere to the 5th and 6th versions of the standard line-by-line. Despite being the most complete academic reference implementation of the standard, they are far from being a suitable replacement for the official ES standard, which is already in its 14th version. Extending the reference interpreter to newer versions of the standard is challenging, mainly due to the inherent limitations of the ECMA-SL language, the most important one being the lack of static typing.

In this thesis, we proposed Typed ECMA-SL, an extension of the ECMA-SL language with type declarations and a static type system. This extension was specifically designed to type the reference interpreters, and thus it only includes the features essential for this purpose. We believe that Typed ECMA-SL will substantially simplify development in ECMA-SL, as static type systems not only increase the readability and maintainability of code, but also reduce the likelihood of bugs when extending or refactoring a program. In addition to the system's implementation, we formalized a subset of Typed ECMA-SL and proved type safety for the formalized fragment.

Typing the reference interpreters using Typed ECMA-SL was beyond the scope of this thesis. Nevertheless, we used Typed ECMA-SL to type a simplified interpreter for the JavaScript language. The interpreter was developed with the purpose of evaluating the effectiveness of our type system by serving as a platform to test all the typing features required by the JS reference interpreter. This process also gave us an idea of the effort needed to fully type the exiting reference interpreters.

Additionally, we applied our type system to the untyped reference interpreters. Despite the lack of type annotations, our system was able to detect several problems within the interpreters, including references to unknown variables and function calls with the wrong number of arguments. This serves as yet another testament to the benefits of employing static type analysis in the context of the ESL project.

7.1 Future Work

We identify two categories of future work: one that falls within the scope of the ECMA-SL project and another that pertains to type systems and programming languages.

ECMA-SL Project Regarding the ECMA-SL project, the Typed ECMA-SL language is still a work in progress, as the type system does not support some less common constructs of the language, such as the `switch` statement and macros. Additionally, the TESL test suite, currently composed of around 50 unit tests, needs to be expanded to ensure that the type system behaves properly in complex typing environments. This is essential to prevent scenarios where we are attempting to fix a type error in the reference interpreter, only to later discover that the root cause lies within the type system itself. Once Typed ECMA-SL reaches a sufficient level of robustness, our next step is to use it to fully type our reference interpreters, so that they can then be extended to newer versions of the ES standard.

Type Systems & Programming Languages While analysing the challenges associated with designing a static type system for Typed ECMA-SL, we identified two topics that may warrant further research.

- **Unsoundness of TypeScript:** Due to the increasing popularity of TypeScript over JavaScript, it is valuable to assess whether the unsound patterns of TS [41] are the causes of bugs in industrial TS projects. If so, it may be beneficial to develop a sound version of TypeScript that addresses all of these patterns while maintaining, when possible, the expressiveness of the language.

- **Object Ownership System:** To address the unsound combination of object mutability and aliasing, we briefly discussed the possibility of using ownership types [48, 48] in Typed ECMA-SL. However, the combination of ownership types with extensible objects is extremely complicated, and thus we were unable to model and implement such a system in the context of this thesis. Nevertheless, this is an interesting research topic and could turn out to be a solution for the problem.

Bibliography

[1] L. Loureiro, “Ecma-sl - a platform for specifying an d running the ecmascript standard,” Master’s thesis, Instituto Superior Técnico, 2021.

[2] R. Rahal, “Ecmaref6: A reference interpreter for modern javascript,” Master’s thesis, Instituto Superior Técnico, 2023.

[3] ECMA-262, *ECMAScript Language Specification*, 5th ed., 2011, accessed on 06-January-2023. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-262_5.1_edition_june_2011.pdf

[4] ——, *ECMAScript Language Specification*, 6th ed., 2015, accessed on 06-January-2023. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf

[5] “Test-262 - ecmascript test suite,” 2022, accessed on 30-November-2022. [Online]. Available: https://github.com/tc39/test262

[6] J. Park, J. Park, S. An, and S. Ryu, “Jiset: Javascript ir-based semantics extraction toolchain,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 647–658.

[7] J. Park, S. An, W. Shin, Y. Sim, and S. Ryu, “Jstar: Javascript specification type analyser using refinement,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 606–616.

[8] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, “A trusted mechanised javascript specification,” *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 87–100, 2014.

[9] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[10] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[11] P. Godefroid, “Compositional dynamic test generation,” in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007, pp. 47–54.

[12] E. Ecma, “262: Ecmascript language specification,” *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[13] D. Park, A. Stefănescu, and G. Roşu, “Kjs: A complete formal semantics of javascript,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 346–356.

[14] A. Charguéraud, A. Schmitt, and T. Wood, “Jsexplain: A double debugger for javascript,” in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 691–699.

[15] L. Almeida, M. Gonzaga, J. F. Santos, and R. Abreu, “Rexstepper: a reference debugger for javascript regular expressions,” in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 41–45.

[16] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner, “Javert: Javascript verification toolchain,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–33, 2017.

[17] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, “Javert 2.0: compositional symbolic execution for javascript,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[18] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, “Jest: N+ 1-version differential testing of both javascript engines and specification,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 13–24.

[19] “Esprima - ecmascript parsing infrastructure for multipurpose analysis,” 2022, accessed on 30-November-2022. [Online]. Available: https://esprima.org/

[20] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, “The ocaml system release 4.13: Documentation and user’s manual,” Ph.D. dissertation, Inria, 2021.

[21] P. Thiemann, “Towards a type system for analyzing javascript programs,” in *European Symposium On Programming*. Springer, 2005, pp. 408–422.

[22] C. Anderson, P. Giannini, and S. Drossopoulou, “Towards type inference for javascript,” in *European conference on Object-oriented programming*. Springer, 2005, pp. 428–452.

[23] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, “Fast and precise type checking for javascript,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[24] A. Chaudhuri, “Flow: Abstract interpretation of javascript for type checking and beyond,” in *Proceedings of the 2016 acm workshop on programming languages and analysis for security*, 2016, pp. 1–1.

[25] K. Dewey, V. Kashyap, and B. Hardekopf, “A parallel abstract interpreter for javascript,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 34–45.

[26] D. Jang and K.-M. Choe, “Points-to analysis for javascript,” in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1930–1937.

[27] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, “Correlation tracking for points-to analysis of javascript,” in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 435–458.

[28] P. A. Gardner, S. Maffeis, and G. D. Smith, “Towards a program logic for javascript,” in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 31–44.

[29] J. F. Santos, P. Gardner, P. Maksimović, and D. Naudžiūnienė, “Towards logic-based verification of javascript programs,” in *International Conference on Automated Deduction*. Springer, 2017, pp. 8–25.

[30] S. Maffeis, J. C. Mitchell, and A. Taly, “An operational semantics for javascript,” in *Asian Symposium on Programming Languages and Systems*. Springer, 2008, pp. 307–325.

[31] A. Guha, C. Saftoiu, and S. Krishnamurthi, “The essence of javascript,” in *European conference on Object-oriented programming*. Springer, 2010, pp. 126–150.

[32] S. Just, A. Cleary, B. Shirley, and C. Hammer, “Information flow analysis for javascript,” in *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, 2011, pp. 9–18.

[33] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, “Staged information flow for javascript,” in *Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation*, 2009, pp. 50–62.

[34] D. Hedin and A. Sabelfeld, "Information-flow security for a core of javascript," in *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 2012, pp. 3–18.

[35] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[36] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi, "A tested semantics for getters, setters, and eval in javascript," in *Proceedings of the 8th Symposium on Dynamic Languages*, 2012, pp. 1–16.

[37] S. Maffeis and A. Taly, "Language-based isolation of untrusted javascript," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 2009, pp. 77–91.

[38] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating javascript with filters, rewriting, and wrappers," in *Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14*. Springer, 2009, pp. 505–522.

[39] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[40] "Typescript documentation," 2022, accessed on 06-January-2023. [Online]. Available: https://www.typescriptlang.org/docs/

[41] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 257–281.

[42] W. Choi, S. Chandra, G. Necula, and K. Sen, "Sjs: A type system for javascript with fixed object layout," in *International Static Analysis Symposium*. Springer, 2015, pp. 181–198.

[43] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 410–429, 2016.

[44] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 310–325.

[45] I. G. de Wolff and J. Hage, "Refining types using type guards in typescript," in *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2017, pp. 111–122.

[46] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[47] V. Gapeyev, M. Y. Levin, and B. C. Pierce, "Recursive subtyping revealed," *Journal of Functional Programming*, vol. 12, no. 6, p. 511–548, 2002.

[48] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1998, pp. 48–64.

[49] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad, "Ownership types: A survey," *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pp. 15–58, 2013.

[50] L. Caires and J. C. Seco, "The type discipline of behavioral separation," *ACM SIGPLAN Notices*, vol. 48, no. 1, pp. 275–286, 2013.

[51] P. Nunes, "A sound type system for the meta language of the javascript standard," Master's thesis, Instituto Superior Técnico, 2021.

[52] J. Fragoso Santos, T. Jensen, T. Rezk, and A. Schmitt, "Hybrid typing of secure information flow in a javascript-like language," in *Trustworthy Global Computing: 10th International Symposium, TGC 2015 Madrid, Spain, August 31–September 1, 2015 Revised Selected Papers 10*. Springer, 2016, pp. 63–78.

[53] J. Fragoso Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, "Gillian, part i: a multi-language platform for symbolic execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 927–942.

A

Type Safety for Typed ECMA-SL

A.1 Auxiliary Lemmas

Lemma 2 (Type Safety for Conditional Type Refinements)**.** For all heap η, store γ, heap typing environment $\mathcal{H}$, store typing environments Γ and Γ', and expression e, it holds that:

$$\begin{aligned}
\eta, \gamma \vDash \mathcal{H}, \Gamma \wedge [\![e]\!]_\gamma = true \;\; \wedge \beta_\Gamma(e) = \Gamma' \quad &\Longrightarrow \quad \eta, \gamma \vDash \mathcal{H}, \Gamma' \\
\eta, \gamma \vDash \mathcal{H}, \Gamma \wedge [\![e]\!]_\gamma = false \wedge \beta_\Gamma(\neg e) = \Gamma' \quad &\Longrightarrow \quad \eta, \gamma \vDash \mathcal{H}, \Gamma'
\end{aligned}$$

Proof. The proof follows by induction on the structure of e. Consider the first case of the lemma (the proof for the second case of the lemma follows the same structure as the one for the first). Assume that $\gamma, \eta \vDash \Gamma, \mathcal{H}$ (**H1**), $[\![e]\!]_\gamma = true$ (**H2**), and $\beta_\Gamma(e) = \Gamma'$ (**H3**). Therefore, we have that:

[CASE 1] $e \notin dom(\widehat{\beta}_\Gamma)$ (**H4**). From **H3** and **H4**, it follows that:

- **I1.1**: $\beta_\Gamma(e) = \Gamma$
- **I1.2**: $\Gamma' = \Gamma$

Equations **I1.2** and **H1** establish the result.

[CASE 2] $e \in dom(\widehat{\beta}_\Gamma)$ (**H4**). We proceed by case analysis on the semantics of the $\widehat{\beta}_\Gamma$ operator.

[CASE 2.1] $e = (e_1 \text{ and } e_2)$ (**H5**). Applying the semantics of the and operator to **H2** and **H5**, we conclude that:

- **I1.1**: $[\![e_1]\!]_\gamma = true$
- **I1.2**: $[\![e_2]\!]_\gamma = true$

From **H3** and **H5**, it follows that there exists store typing environments Γ_1, Γ_2, such that:

- **I2.1**: $\beta_\Gamma(e_1) = \Gamma_1$
- **I2.2**: $\beta_{\Gamma_1}(e_2) = \Gamma_2$
- **I2.3**: $\Gamma' = \Gamma_2$

Applying the induction hypothesis to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $\gamma, \eta \vDash \Gamma_1, \mathcal{H}$

Applying the induction hypothesis to **I1.2**, **I2.2**, and **I3.1**, we conclude that:

- **I4.1**: $\gamma, \eta \vDash \Gamma_2, \mathcal{H}$

Equations **I2.3** and **I4.1** establish the result.

[CASE 2.2] $e = (x = v)$ (**H5**). Applying the semantics of the comparison operator to **H2** and **H5**, we conclude that:

- **I1.1**: $\gamma(x) = v$

From **H3** and **H5**, it follows that there exists $\tau \in T$ such that:

- **I2.1**: $type(v) = \tau$
- **I2.2**: $\tau \leq \Gamma(x)$
- **I2.3**: $\Gamma' = \Gamma[x \mapsto \tau]$

From **I1.1**, **I2.1**, and **I2.3**, it follows that:

- **I3.1**: $\gamma(x) \vDash_\mathcal{H} \Gamma'(x)$

Equations **I3.1** and **H1** establish the result.

[CASE 2.3] $e = (\text{typeof}(x) = \tau)$ with $\tau \in TPrim$ (**H5**). Applying the semantics of the typeof operator to **H2** and **H5**, we conclude that:

- **I1.1**: $type_\mathcal{H}(\gamma(x)) \leq \tau$

From **H3** and **H5**, it follows that:

- **I2.1**: $\tau \leq \Gamma(x)$
- **I2.2**: $\Gamma' = \Gamma[x \mapsto \tau]$

From **I1.1** and **I2.2**, it follows that:

- **I3.1**: $\gamma(x) \vDash_\mathcal{H} \Gamma'(x)$

Equations **I3.1** and **H1** establish the result.

[CASE 2.4] $e = (\text{typeof}(x) \neq \tau)$ with $\tau \in TPrim$ (**H5**). Applying the semantics of the typeof operator to **H2** and **H5**, we conclude that:

- **I1.1**: $type_\mathcal{H}(\gamma(x)) \not\leq \tau$

From **H3** and **H5**, it follows that there exists $\tau_1,\ ...,\ \tau_n, \tau' \in T$, such that:

- **I2.1**: $\Gamma(x) = \cup\{\tau_1,\ ...,\ \tau_n\}$

- **I2.2**: $\tau' = \cup(\{\tau_1,\ ...,\ \tau_n\} \setminus \tau)$
- **I2.3**: $\Gamma' = \Gamma[x \mapsto \tau']$

From **H1**, it follows that:

- **I3.1**: $type_{\mathcal{H}}(\gamma(x)) \leq^* \Gamma(x)$

Applying the correctness of union subtyping to **I1.1**, **I2.1**, **I2.2**, and **I3.1**, we conclude that:

- **I4.1**: $type_{\mathcal{H}}(\gamma(x)) \leq^* \tau'$

Equations **I2.3**, **I4.1**, and **H1** establish the result.

[CASE 2.5] $e = (v = x)$ (**H5**). This proof follows the same structure as [CASE 2.2].

[CASE 2.6] $e = (\tau = \mathsf{typeof}(x))$ (**H5**). This proof follows the same structure as [CASE 2.3].

[CASE 2.7] $e = (\tau \neq \mathsf{typeof}(x))$ (**H5**). This proof follows the same structure as [CASE 2.4]. □

Lemma 3 (Type Safety for Pattern Binding)**.** For all heap η, stores γ and γ', heap typing environment $\mathcal{H}$, store typing environments Γ and Γ', pattern ψ, object value o, object type σ, and sigma type τ, it holds that:

$$\eta, \gamma \vDash \mathcal{H}, \Gamma \ \wedge\ match(\psi, o, \gamma) = \gamma' \ \wedge\ \Psi^{\alpha,\tau}(\psi, \sigma, \Gamma) = \Gamma' \ \wedge\ \tau = \Sigma[\alpha]\,\{...,\ \sigma,\ ...\} \ \wedge\ o \leq^* \sigma[{}^{\alpha}/_{\tau}] \\ \implies\ \eta, \gamma' \vDash \mathcal{H}, \Gamma'$$

Proof. The proof follows by induction on the number of elements of ψ. Assume that $\gamma, \eta \vDash \Gamma, \mathcal{H}$ (**H1**), $match(\psi, o, \gamma) = \gamma'$ (**H2**), $\Psi^{\alpha,\tau}(\psi, \sigma, \Gamma) = \Gamma'$ (**H3**), $\tau = \Sigma[\alpha]\,\{...,\ \sigma,\ ...\}$ (**H4**), and $o \leq^* \sigma[{}^{\alpha}/_{\tau}]$ (**H5**). Therefore, we have that:

[BASE CASE (BCASE)] $\psi = \{\}$ (**H6**). Applying the semantics of the $match$ function to **H2** and **H6**, we conclude that:

- **I1.1**: $\gamma' = \gamma$

Applying the typing rules of the $\Psi^{\alpha,\tau}$ function to **H3**, **H4**, **H6**, we conclude that:

- **I2.1**: $\Gamma' = \Gamma$

Equations **I1.1**, **I2.1**, and **H1** establish the result.

[INDUCTIVE CASE (ICASE)] $\psi = \{f : \xi\} \uplus \psi'$ (**H6**), $o = \{f : v\} \uplus o'$ (**H7**), and $\sigma = \{f : \tau_f\} \uplus \sigma'$ (**H8**). We proceed by case analysis on the type of the pattern binding ξ.

[ICASE 1] $\xi = v$ (**H9**). Applying the semantics of the $match$ function to **H2**, **H6**, **H7**, and **H9**, we conclude that:

- **I1.1**: $\gamma' = match(\psi', o', \gamma)$

Applying the typing rules of the Ψ function to **H3**, **H4**, **H6**, **H8**, and **H9**, we conclude that:

- **I2.1**: $\Gamma' = \Psi^{\alpha,\tau}(\psi', \sigma', \Gamma)$

Applying the induction hypothesis to **I1.1**, **I2.1**, **H1**, **H4**, and **H5**, we conclude that:

- **I3.1**: $\eta, \gamma' \vDash \mathcal{H}, \Gamma'$

Equation **I3.1** establishes the result.

[ICASE 2] $\xi = x$ (**H9**). Applying the semantics of the $match$ function to **H2**, **H6**, **H7**, and **H9**, we conclude that there exists a program store γ'' such that:

- **I1.1**: $\gamma' = match(\psi', o', \gamma'')$
- **I1.2**: $\gamma'' = \gamma[x \mapsto v]$

Applying the typing rules of the Ψ function to **H3**, **H4**, **H6**, **H8**, and **H9**, we conclude that there exists a store typing environment Γ'' such that

- **I2.1**: $\Gamma' = \Psi^{\alpha,\tau}(\psi', \sigma', \Gamma'')$
- **I2.2**: $\Gamma'' = \Gamma[x \mapsto \tau_f[\alpha/\tau]]$

Applying the correctness of covariant sigma folding subtyping to **H4**, **H5**, **H7**, and **H8**, it follows that:

- **I3.1**: $v \leq^* \tau_f[\alpha/\tau]$

From **I1.2**, **I2.2**, **I3.1** and **H1**, it follows that:

- **I4.1**: $\gamma''(x) \vDash_{\mathcal{H}} \Gamma''(x)$
- **I4.2**: $\eta, \gamma'' \vDash \mathcal{H}, \Gamma''$

Applying the induction hypothesis to **I1.1**, **I2.1**, **I4.2**, **H4**, and **H5** we conclude that:

- **I5.1**: $\eta, \gamma' \vDash \mathcal{H}, \Gamma'$

Equation **I5.1** establishes the result. □

A.2 Intra-Procedural Fragment

Theorem 1 (Soundness - Type Safety)**.** For all heaps η, η', stores γ, γ', heap typing environments $\mathcal{H}, \mathcal{H}'$, store typing environments Γ, Γ', function g, global typing context Δ, and statement s it holds that:

$$\eta, \gamma \vDash \mathcal{H}, \Gamma \ \wedge \ g, \Delta \vdash \{\Gamma\} \ s \ \{\Gamma'\} \ \wedge \ \langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle \implies \eta', \gamma' \vDash \mathcal{H}', \Gamma'$$

Proof. The proof follows by induction on the derivation of the judgement $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$. Assume that $\eta, \gamma \vDash \mathcal{H}, \Gamma$ (**H1**), $g, \Delta \vdash \{\Gamma\} \ s \ \{\Gamma'\}$ (**H2**), and $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$ (**H3**). Therefore, we have that:

[SKIP] $s = \mathsf{skip}$ (**H4**). From **H2** and **H4**, it follows that:

- **I1.1**: $\Gamma' = \Gamma$

From **H3** and **H4**, it follows that:

- **I2.1**: $\mathcal{H}' = \mathcal{H}$
- **I2.2**: $\eta' = \eta$
- **I2.3**: $\gamma' = \gamma$

Equations **I1.1**, **I2.1**, **I2.2**, **I2.3**, and **H1** establish the result.

[SEQUENCING] $s = s_1;\, s_2$ (**H4**). From **H2** and **H4**, it follows that there exists a store typing environment Γ'' such that:

- **I1.1**: $g, \Delta \vdash \{\Gamma\}\ s_1\ \{\Gamma''\}$
- **I1.2**: $g, \Delta \vdash \{\Gamma''\}\ s_2\ \{\Gamma'\}$

From **H3** and **H4**, it follows that there exists a store γ'', heap η'', and heap typing environment $\mathcal{H}''$ such that:

- **I2.1**: $\langle \mathcal{H}, \eta, \gamma, s_1 \rangle \Downarrow_i \langle \mathcal{H}'', \eta'', \gamma'' \rangle$
- **I2.2**: $\langle \mathcal{H}'', \eta'', \gamma'', s_2 \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the induction hypothesis to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $\eta'', \gamma'' \vDash \mathcal{H}'', \Gamma''$

Applying the induction hypothesis to **I1.2**, **I2.2**, and **I3.1**, we conclude that:

- **I4.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma'$

Equation **I4.1** establishes the result.

[VARIABLE ASSIGNMENT] $s = x := e$ (**H4**). From **H2** and **H4**, it follows that there exists a $\tau \in T$, such that:

- **I1.1**: $\Gamma \vdash e : \tau$
- **I1.2**: $\Gamma' = \Gamma[x \mapsto \tau]$

From **H3** and **H4**, it follows that there exists a $v \in Val$, such that:

- **I2.1**: $[\![e]\!]_\gamma = v$
- **I2.2**: $\gamma' = \gamma[x \mapsto v]$
- **I2.3**: $\mathcal{H}' = \mathcal{H}$
- **I2.4**: $\eta' = \eta$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $v \vDash_{\mathcal{H}} \tau$

From **I1.2**, **I2.2**, **I3.1**, **H1**, it follows that:

- **I4.1**: $\gamma' \vDash_{\mathcal{H}} \Gamma'$

Equations **I2.3**, **I2.4**, **I4.1**, and **H1** establish the result.

[FIELD LOOKUP] $s = x := e.f$ (**H4**). From **H2** and **H4**, it follows that there exists $\sigma \in TObj$ and $\tau_f \in T$ such that:

- **I1.1**: $\Gamma \vdash e : \sigma$
- **I1.2**: $ft(\sigma, f) = \tau_f$
- **I1.3**: $\Gamma' = \Gamma[x \mapsto \tau_f]$

From **H3** and **H4**, it follows that there exists $l \in Loc$ and $v \in Val$ such that:

- **I2.1**: $[\![e]\!]_\gamma = l$
- **I2.2**: $\eta(l, f) = v$
- **I2.3**: $\gamma' = \gamma[x \mapsto v]$
- **I2.4**: $\mathcal{H}' = \mathcal{H}$
- **I2.5**: $\eta' = \eta$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $l \vDash_{\mathcal{H}} \sigma$
- **I3.2**: $\mathcal{H}(l) \leq^* \sigma$

Applying the correctness of covariant object subtyping to **I1.2** and **I3.2**, we conclude that:

- **I4.1**: $ft(\mathcal{H}(l), f) \leq^* \tau_f$

Applying the typing rules of the ft function to **H1**, we conclude that:

- **I5.1**: $type_{\mathcal{H}}(\eta(l, f)) = ft(\mathcal{H}(l), f)$

From **I2.2**, **I4.1**, and **I5.1**, it follows that:

- **I6.1**: $type_{\mathcal{H}}(v) \leq^* \tau_f$
- **I6.2**: $v \vDash_{\mathcal{H}} \tau_f$

From **I1.3**, **I2.3**, **I6.2**, and **H1**, it follows that:

- **I7.1**: $\gamma' \vDash_{\mathcal{H}} \Gamma'$

Equations **I2.4**, **I2.5**, **I7.1**, and **H1** establish the result.

[NEW OBJECT] $s = x := \{f_i : e_i \mid_{i=1}^{n}\}$ (**H4**). From **H2** and **H4**, it follows that there exists $\tau_1,\ ...,\ \tau_n \in T$, and $\sigma \in TObj$, such that:

- **I1.1**: $\Gamma \vdash e_i : \tau_i \mid_{i=1}^{n}$
- **I1.2**: $\sigma = \{f_i : \tau_i \mid_{i=1}^{n}\}$
- **I1.3**: $\Gamma' = \Gamma[x \mapsto \sigma]$

From **H3** and **H4**, it follows that there exists $l \in Loc$ and $v_1,\ ...,\ v_n \in Val$, and $\tau \in T$, such that:

- **I2.1**: $l \notin dom(\eta)$
- **I2.2**: $[\![e_i]\!]_\gamma = v_i \mid_{i=1}^{n}$
- **I2.3**: $\{f_i : type_{\mathcal{H}}(v_i) \mid_{i=1}^{n}\} = \tau$
- **I2.4**: $\mathcal{H}' = \mathcal{H}[l \mapsto \tau]$
- **I2.5**: $\eta' = \eta[(l, f_i) \mapsto v_i] \mid_{i=1}^{n}$
- **I2.6**: $\gamma' = \gamma[x \mapsto l]$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.2**, and **H1**, we conclude that:

- **I3.1**: $v_i \vDash_{\mathcal{H}} \tau_i \mid_{i=1}^{n}$

- **I3.2**: $type_{\mathcal{H}}(v_i) \leq^* \tau_i \,|_{i=1}^{n}$

Applying the correctness of covariant object subtyping to **I1.2**, **I2.3**, **I2.4**, and **I3.2**, it follows that:

- **I4.1**: $\tau \leq^* \sigma$
- **I4.2**: $l \vDash_{\mathcal{H}'} \sigma$

From **I1.3**, **I2.6**, **I4.2**, and **H1**, it follows that:

- **I5.1**: $\gamma' \vDash_{\mathcal{H}'} \Gamma'$

From **I2.3**, **I2.4**, **I2.5**, and **H1**, it follows that:

- **I6.1**: $type_{\mathcal{H}'}(\eta'(l, f_i)) = \mathcal{H}'(l, f_i) \,|_{i=1}^{n}$
- **I6.2**: $\eta' \vDash \mathcal{H}'$

Equations **I5.1** and **I6.2** establish the result.

[FIELD ASSIGNMENT] $s = x.f \coloneqq e$ (**H4**). From **H2** and **H4**, it follows that there exists $\sigma \in TObj$, and $\tau_f, \tau_e \in T$, such that:

- **I1.1**: $\Gamma(x) = \sigma$
- **I1.2**: $ft(\sigma, f) = \tau_f$
- **I1.3**: $\Gamma \vdash e : \tau_e$
- **I1.4**: $\tau_e \leq \tau_f$
- **I1.5**: $\Gamma' = \Gamma$

From **H3** and **H4**, it follows that there exists $v \in Val$, $l \in Loc$ such that:

- **I2.1**: $[\![x]\!]_\gamma = l$
- **I2.2**: $(l, f) \in dom(\eta)$
- **I2.3**: $[\![e]\!]_\gamma = v$
- **I2.4**: $\mathcal{H}(l)[f \mapsto type_{\mathcal{H}}(v)] = \tau$
- **I2.5**: $\mathcal{H}' = \mathcal{H}[l \mapsto \tau]$
- **I2.6**: $\eta' = \eta[(l, f) \mapsto v]$
- **I2.7**: $\gamma' = \gamma$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $l \vDash_{\mathcal{H}} \sigma$
- **I3.2**: $\mathcal{H}(l) \leq^* \sigma$

Applying the correctness of covariant object subtyping to **I1.2** and **I3.2**, we conclude that:

- **I4.1**: $\mathcal{H}(l, f) \leq^* \tau_f$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.3**, **I2.3**, and **H1**, we conclude that:

- **I5.1**: $v \vDash_{\mathcal{H}} \tau_e$
- **I5.2**: $type_{\mathcal{H}}(v) \leq^* \tau_e$

Applying the correctness of transitive subtyping to **I1.4** and **I5.2**, we conclude that:

- **I6.1**: $type_{\mathcal{H}}(v) \leq^* \tau_f$

From **I2.4**, **I2.5**, **I4.1**, and **I6.1**, it follows that:

- **I7.1**: $\mathcal{H}'(l) \leq^* \sigma$

From **I1.5**, **I2.5**, **I2.7**, **I3.1**, **I7.1**, and **H1**, it follows that:

- **I8.1**: $\gamma \vDash_{\mathcal{H}'} \Gamma$

From **I2.4**, **I2.5**, **I2.6**, and **H1**, it follows that:

- **I9.1**: $\eta' \vDash \mathcal{H}'$

Equations **I8.1** and **I9.1** establish the result.

[IF-THEN-ELSE (TRUE)] $s = \text{if}\,(e)\{s_1\}\,\text{else}\,\{s_2\}$ (**H4**) and $[\![e]\!]_\gamma = true$ (**H5**). From **H2** and **H4**, it follows that there exists $\tau_e \in T$, and store typing environments Γ_1, Γ_2, Γ_1', and Γ_2', such that:

- **I1.1**: $\Gamma \vdash e : \tau_e$
- **I1.2**: $\tau_e \leq bool$
- **I1.3**: $\beta_\Gamma(e) = \Gamma_1$
- **I1.4**: $\beta_\Gamma(\neg e) = \Gamma_2$
- **I1.5**: $g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma_i'\}\ |_{i=1}^{2}$
- **I1.6**: $\Gamma' = \Gamma_1' \sqcup \Gamma_2'$

From **H3** and **H4**, it follows that:

- **I2.1**: $\langle \mathcal{H}, \eta, \gamma, s_1 \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the Lemma 2 (*Type Safety for Conditional Type Refinements*) to **H1**, **H5**, and **I1.3**, we conclude that:

- **I3.1**: $\eta, \gamma \vDash \mathcal{H}, \Gamma_1$

Applying the induction hypothesis to **I1.5**, **I2.1**, and **I3.1**, we conclude that:

- **I4.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma_1'$

By correctness of union subtyping, equations **I1.6** and **I4.1** establish the result.

[IF-THEN-ELSE (FALSE)] $s = \text{if}\,(e)\{s_1\}\,\text{else}\,\{s_2\}$ (**H4**) and $[\![e]\!]_\gamma = false$ (**H5**)
This proof follows the same structure as the IF-THEN-ELSE (TRUE) case.

[WHILE LOOP (TRUE)] $\text{while}\,(e)\{s\}$ (**H4**) and $[\![e]\!]_\gamma = true$ (**H5**) From **H2** and **H4**, it follows that there exists $\tau_e \in T$, and a store typing environment Γ'', such that:

- **I1.1** $\Gamma \vdash e : \tau_e$
- **I1.2** $\tau_e \leq bool$
- **I1.3** $\beta_\Gamma(e) = \Gamma''$
- **I1.4** $g, \Delta \vdash \{\Gamma''\}\ s\ \{\Gamma\}$

- **I1.5** $\Gamma' = \Gamma$

From **H3** and **H4**, it follows that there exits a heap typing environment $\mathcal{H}''$, a heap η'', and a store γ'', such that:

- **I2.1** $\langle \mathcal{H}, \eta, \gamma, s \rangle \Downarrow_i \langle \mathcal{H}'', \eta'', \gamma'' \rangle$
- **I2.2** $\langle \mathcal{H}'', \eta'', \gamma'', \mathsf{while}\,(e)\{s\} \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the Lemma 2 (*Type Safety for Conditional Type Refinements*) to **H1**, **H5**, and **I1.3**, we conclude that:

- **I3.1**: $\eta, \gamma \vDash \mathcal{H}, \Gamma''$

Applying the induction hypothesis to **I1.4**, **I2.1**, and **I3.1**, we conclude that:

- **I4.1**: $\eta'', \gamma'' \vDash \mathcal{H}'', \Gamma$

Applying the induction hypothesis to **H2**, **I2.2**, and **I4.1**, we conclude that:

- **I5.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma'$

Equation **I5.1** establishes the result.

[WHILE LOOP (FALSE)] $\mathsf{while}\,(e)\{s\}$ (**H4**) and $[\![e]\!]_\gamma = false$ (**H5**) From **H2** and **H4**, it follows that there exists $\tau_e \in T$, and a store typing environment Γ'', such that:

- **I1.1** $\Gamma \vdash e : \tau_e$
- **I1.2** $\tau_e \leq bool$
- **I1.3** $\beta_\Gamma(e) = \Gamma''$
- **I1.4** $g, \Delta \vdash \{\Gamma''\}\ s\ \{\Gamma\}$
- **I1.5** $\Gamma' = \Gamma$

From **H3** and **H4**, it follows that:

- **I2.1**: $\mathcal{H}' = \mathcal{H}$
- **I2.2**: $\eta' = \eta$
- **I2.3**: $\gamma' = \gamma$

Equations **I1.5**, **I2.1**, **I2.2**, **I2.3**, and **H1** establish the result.

[MATCH-WITH] $\mathsf{match}\, e\, \mathsf{with}\, \{\psi_i \to s_i \mid_{i=1}^{n}\}$ (**H4**) From **H2** and **H4**, it follows that there exists $\tau \in T$, $\sigma_1,\ ...,\ \sigma_n \in TObj$, and store typing environments $\Gamma_1,\ ...,\ \Gamma_n, \Gamma'_1,\ ...,\ \Gamma'_n$, such that:

- **I1.1**: $\Gamma \vdash e : \tau$
- **I1.2**: $\tau = \Sigma[\alpha]\,\{\sigma_1,\ ...,\ \sigma_n\}$
- **I1.3**: $\Psi^{\alpha,\tau}(\psi_i, \sigma_i, \Gamma) = \Gamma_i \mid_{i=1}^{n}$
- **I1.4**: $g, \Delta \vdash \{\Gamma_i\}\ s_i\ \{\Gamma'_i\} \mid_{i=1}^{n}$
- **I1.5**: $\Gamma' = \sqcup\{\Gamma'_1,\ ...,\ \Gamma'_n\}$

From **H3** and **H4**, it follows that there exists $j \in [1, n]$, $l \in Loc$, and a store γ_j, such that:

- **I2.1**: $[\![e]\!]_\gamma = l$

- **I2.2**: $match(\psi_j, \eta(l), \gamma) = \gamma_j$
- **I2.3**: $\langle \mathcal{H}, \eta, \gamma_j, s_j \rangle \Downarrow_i \langle \mathcal{H}', \eta', \gamma' \rangle$

Applying the Lemma 1 (*Type Safety for Expressions*) to **I1.1**, **I2.1**, and **H1**, we conclude that:

- **I3.1**: $l \vDash_{\mathcal{H}} \tau$
- **I3.2**: $\mathcal{H}(l) \leq^* \tau$

Applying the correctness of sigma folding subtyping to **I1.2** and **I3.2**, we conclude that there exists $\sigma' \in TObj$ such that:

- **I4.1**: $\sigma' \in \{\sigma_1,\ ...,\ \sigma_n\}$
- **I4.2**: $\mathcal{H}(l) \leq^* \sigma'[\tau/\alpha]$

Applying the semantics of the `match-with` statement to **I1.3**, **I2.2**, **I4.2**, and **H1**, it follows that

- **I5.1**: $\eta(l) \leq^* \sigma_j[\tau/\alpha]$

Applying the Lemma 3 (*Type Safety for Pattern Binding*) to **H1**, **I1.2**, **I1.3**, **I2.2**, and **I5.1**, we conclude that:

- **I7.1**: $\eta, \gamma_j \vDash \mathcal{H}, \Gamma_j$

Applying the induction hypothesis to **I1.4**, **I2.3**, and **I7.1**, we conclude that:

- **I8.1**: $\eta', \gamma' \vDash \mathcal{H}', \Gamma'_j$

By correctness of union subtyping, equations **I1.5** and **I8.1** establish the result. □

B

Auxiliary Functions

Type of Value The static type associated with a value is obtained with the function $type : Val \mapsto T$, which maps values $v \in Val$ into their respective static types $\tau \in T$. This function is defined as:

$$type(v) = \begin{cases} \nu & \text{if } v \in TLit \\ \texttt{null} & \text{if } v = null \\ \texttt{undefined} & \text{if } v = undefined \\ \texttt{top} & \text{otherwise,} \end{cases}$$

where ν represents the literal type associated with value v.

Primitive of Literal Type The primitive type associated with a literal type is obtained with the function $typeof : TLit \mapsto TPrim$, which maps literal types $\nu \in TLit$ to their respective primitive types $t \in TPrim$. This function is defined as:

$$typeof(\nu) = \begin{cases} \texttt{int} & \text{if } v \in Int \\ \texttt{float} & \text{if } v \in Flt \\ \texttt{string} & \text{if } v \in Str \\ \texttt{boolean} & \text{if } v \in Bool, \end{cases}$$

where v represents the value associated with the literal type ν.

Extended Subtyping Figure B.1 depicts the typing rules concerning the extended subtyping relationship supported in Typed ECMA-SL. This relationship extends the subtyping relationship defined in Section 4.2.2 with field covariance, meaning that object fields are allowed to be a subtype of the fields specified in the most generic object. The extended subtyping relationship it is used to verify whether a value of type τ can be stored by a variable or field of type τ' (Section 4.4.1).

$$\begin{array}{c}\textsc{Field Covariance}\\ \dfrac{\tau_i \leq^* \tau'_i \mid_{i=1}^{n}}{\{f_i : \tau_i \mid_{i=1}^{n}\} \leq^* \{f_i : \tau'_i \mid_{i=1}^{n}\}}\end{array} \qquad \begin{array}{c}\textsc{Subtyping}\\ \dfrac{\tau \leq \tau'}{\tau \leq^* \tau'}\end{array}$$

Figure B.1: Typing Rules - Extended Subtyping: $\tau \leq^* \tau'$