VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van een doctoraatsdiploma
Dissertation submitted in fulfilment of the requirement for the degree of Doctor of Philosophy in Sciences

STATICALLY CHECKING INTER-PROPERTY CONSTRAINTS
and its Applications in Web APIs

Nathalie Oostvogels

Promotor: Prof. Dr. Wolfgang De Meuter
Copromotor: Prof. Dr. Joeri De Koster

Faculty of Science and Bio-Engineering Sciences

Statically Checking Inter-property Constraints

and its Applications in Web APIs

Nathalie Oostvogels

Dissertation submitted in fulfillment of the requirement for the degree of Doctor of Sciences

April 2019

Jury:
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel (promotor)
Prof. Dr. Joeri De Koster, Vrije Universiteit Brussel (copromotor)
Prof. Dr. Ann Nowé, Vrije Universiteit Brussel (chair)
Prof. Dr. Dominique Devriese, Vrije Universiteit Brussel (secretary)
Prof. Dr. Ann Dooms, Vrije Universiteit Brussel
Prof. Dr. Tobias Wrigstad, Uppsala University
Dr. Manuel Serrano, INRIA

Printed by:
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 978 94 930 7921 2
NUR 989

Acknowledgements:
The work in this dissertation has been funded by a PhD Fellowship of the Research Foundation - Flanders (FWO) and the SBO project Tearless by the Agency for Innovation and Entrepreneurship (VLAIO).

Abstract

Software applications do not stand on their own: their code often uses libraries to incorporate functionality to facilitate the development process (such as abstractions and helper functions) or to integrate functionality from third parties. Libraries offer their functionality via an Application Programming Interface (API), which is a contract between the library and the user. It defines which methods are provided, and imposes constraints on the data or fields that are exchanged.

Constraints dictate the presence of fields, the type of fields, and the allowed values for fields. Constraints imposed on a singular field are clearly indicated in the documentation and well-supported in tooling and specification languages. Next to singular constraints, API documentation often describes relations *between* fields. For example, some fields may only be provided together, or the value of a field may impose constraints on other parts of the data. In this dissertation, we demonstrate the prevalence of such constraints in web APIs.

We show that there is no structural support for these *inter-property constraints*: they cannot be expressed by contemporary API specification languages such as OpenAPI. This lack of support extends to programming languages, whose type system can only express and validate single-property constraints, but not inter-property constraints. This dissertation presents a statically-typed programming language that fills this gap.

Our programming language, TIPC, features a natural extension to interface definitions which allows enforcing presence constraints between properties. TIPC ensures that objects with inter-property constraints satisfy these constraints throughout the program. Furthermore, it enables programmers to refine interface types using its flow-sensitive type system. We present a formal specification of the syntax, operational semantics and type system of TIPC, along with soundness proofs. As a proof of concept, we use TIPC as a model to extend the TypeScript compiler with support for inter-property constraints.

Finally, we extend the OpenAPI specification language with support for fully generalised inter-property constraints. This language serves as a proving ground for inter-property constraint aware tooling. Concretely, a first artefact presents an intercepting middleware that verifies constraints on both the caller side and the library side. A second artefact shows how interfaces enabled with inter-property constraints can be generated from an API specification. Both artefacts enable the automatic verification of inter-property constraints in applications, respectively at runtime and at compile-time.

Samenvatting

Softwareprogramma's staan niet op zichzelf: ze integreren vaak functionaliteit van bestaande codebibliotheken. De functionaliteit die zulke bibliotheken aanbieden, kan het ontwikkelingsproces bevorderen of functionaliteit van derde partijen beschikbaar stellen. Deze functionaliteit wordt beschikbaar gesteld via een API (een programmeerinterface), die een contract vormt tussen de codebibliotheek en de gebruiker. Dit contract beschrijft welke methodes aangeboden worden en welke vereisten er worden gelegd op de data of velden die worden uitgewisseld.

Vereisten leggen restricties op de aanwezigheid van velden, op hun type en hun toegestane waardes. Vereisten op een enkel veld zijn goed aangeduid in de documentatie en zijn goed ondersteund door programmeerhulpmiddelen en specificatietalen. Naast vereisten op een enkel veld, legt de documentatie van API's ook nog eisen op *tussen* velden. Sommige velden mogen bijvoorbeeld enkel samen voorkomen, of de waarde van een veld kan de vereisten op een ander veld beïnvloeden. In deze verhandeling tonen wij het voorkomen van zulke vereisten in de documentatie van web API's.

We tonen dat er geen structurele ondersteuning bestaat voor *inter-veld vereisten*: ze kunnen niet worden uitgedrukt in hedendaagse specificatietalen zoals OpenAPI. Ook programmeertalen bieden geen ondersteuning: hun typesystemen kunnen enkel vereisten over een enkel veld uitdrukken. Deze verhandeling presenteert een statisch getypeerde programmeertaal die deze kloof dicht.

Onze programmeertaal, TIPC, heeft een natuurlijke uitbreiding van interface definities die ervoor zorgt dat aanwezigheidsvereisten tussen velden kunnen worden opgelegd. TIPC garandeert dat objecten met inter-veld vereisten tijdens het hele programma aan deze eisen voldoen. Bovendien zorgt TIPC ervoor dat programmeurs de interface types specifieker kunnen maken doordat het typesysteem rekening houdt met if-testen in het programmaverloop. We presenteren een formele specificatie van de syntaxis, operationele semantiek en het typesysteem van TIPC, samen met een bewijs van correctheid. We integreren het TIPC model in de TypeScript compiler, waardoor we een uitbreiding op TypeScript verkrijgen die inter-veld vereisten kan garanderen.

Tenslotte breiden we de OpenAPI specificatietaal uit met ondersteuning voor algemene inter-veld vereisten. Deze taal dient als een basis om hulpprogramma's met inter-veld constraints te testen. Een eerste artifact presenteert een middleware die nagaat of aan de vereisten wordt voldaan, zowel aan de kant van de gebruiker als de kant van de codebibliotheek. Een tweede artifact toont hoe interfaces met inter-veld vereisten gegenereerd kunnen worden uit API specificaties. Beide artifacten zorgen ervoor dat er automatisch wordt nagegaan of inter-veld vereisten worden voldaan, zowel tijdens het uitvoeren als vooraf.

Acknowledgments

First of all, I would like to thank the two people who promote this work: Wolf and Joeri. Wolf, I want to sincerely thank you for giving me the opportunity to pursue a PhD. You let me take my time to figure out what the topic of this PhD was going to be, which shaped this PhD to what it is today. I would like to thank the members of the jury, Dominique Devriese, Ann Dooms, Ann Nowé, Manuel Serrano, and Tobias Wrigstad, for the interesting discussions, which resulted in new insights that definitely improved this dissertation. Next to the promotors and jury members, I would like to thank the extra proofreaders of my text: Quentin and Dries.

I would like to thank the unique bunch of people whom I may call my colleagues. There are a few (ex-)colleagues (going on friends) I want to thank in particular. For the most part of this PhD, I have had the privilege of working together with one of my *bestest* friends. Simon: it has been an honour to have you by my side to teach a few hundred 18-year olds the ins and outs of algorithms and data structures. A big thanks to *het olijke (office-sharing) duo*: Thierry and Janwillem. You guys were always up for PhD-related discussions which shaped and structured my thoughts and worries about inter-property constraints, as well as the (equally as important) non-PhD-related discussions. Jesse, thank you for your brutally honest view on things. Laure, your absence at SOFT (and the emptiness of your bowl of candy) is not going unnoticed. Lara, thanks for the talks about the little things in life, the sad things, the happy things, and the big things.

One of Wolf's selling points of a PhD is that it comes along with the chance of exploring the world, and he was certainly right. A big thanks (again) to my travel buddies throughout the years: chasing Oregon's waterfalls with Thierry & Laure, finally going on "Romereis" with Laure & Janwillem (I can recommend them as tour guides!), and lunching at the foot of a Canadian glacier with Janwillem.

Many friends and family have supported me during the past few years. I want to truly thank you all for all the diversions in the form of board gaming nights, food and/or Sunday afternoon teas. It is impossible (and too dangerous) to list everyone, but special shoutout to the OG: Killian, Koen, and Murielle!

Before this acknowledgment section ends and the actual dissertation starts, I want to thank the most important people in my life. In het dankwoord van mijn masterthesis heb ik mijn oma bedankt voor haar oneindige voorraad aan chocotoffs. Ik ben blij dat dat na al die jaren nog niet veranderd is: nog eens bedankt oma!

Mama & papa: zonder jullie onvoorwaardelijke steun bij de start van zowel mijn hogeschoolopleiding, mijn schakeljaar, en mijn doctoraat had ik hier nooit

gestaan. Ik ben jullie hiervoor oneindig dankbaar. Mama, jouw doorzettingsvermogen en leergierigheid zijn altijd een voorbeeld voor mij geweest. Bedankt om samen met mij de Visual Basic cursus van het middelbaar al te maken, omdat ik niet kon wachten tot wanneer de school daar mee begon. Dat heeft – zonder twijfel – de fundering gelegd voor dit doctoraat, een luttele 14 jaar later.

I am forever grateful for the privilege of having the best sister in the entire world. Sofie: you are the definition of a BFF, or to say it with a Pinterest quote: there is no better friend than a sister, and there is no better sister than you.

Dries: being with someone who is writing a paper or a PhD is not the easiest (as I know all too well ;-)). Thank you for your limitless patience and unconditional support. Life is best with you by my side.

Contents

Ik heb het nog nooit gedaan,
dus ik denk dat ik het wel kan.

– Pipi Langkous

Chapter 1

Introduction

Libraries form the building blocks for software applications: they promote reusability and abstraction by encapsulating code, as well as provide functionality from third parties. The communication between a software application and a library happens through an Application Programming Interface (API), which offers the functionality of a library through a set of *methods*. API documentation describes the functionality of each API method, together with a list of expected *fields* and a description of what the fields should look like. Often, the documentation describes the desired form of the fields as a set of requirements.

Satisfying the requirements before calling the API is essential for the API call to be valid. Manually verifying every constraint of every API call in a software application is a time-consuming and tedious job. Luckily, there are several ways to automate the verification of these constraints. For example, requirements can be translated to *types* in a statically typed programming language: this way, the type system automatically verifies the requirements at compile-time.

1.1 Research Context

Early type systems only describe the basic type of the values that could be stored in a variable. These type systems prevent standard type errors such as the multiplication of booleans or text. They also verify that function calls have the correct amount of arguments and that the arguments have the correct type. Type systems designed for object-oriented languages have object types which define the set of properties an object should have, together with their type [Abadi and Cardelli, 1996; Pierce, 2002].

Throughout the years, more complex types have been introduced, such as union types and intersection types [Pottinger, 1980]. For example, union types

enable the developer to indicate that a variable may be a string *or* a number. Linear types [Girard, 1987] can be used to guarantee that there is *only one* reference to a variable at any time in the program. Dependent types [De Bruijn, 1970; Howard, 1980; Martin-Löf and Sambin, 1984] introduce types that may depend on *values*. This enables verification of advanced constraints such as: *an index must be in the bounds of an array.*

Using these more expressive types, developers can express more sophisticated programs while retaining the compile-time guarantee that their code satisfies the envisioned invariants. However, the power of a type system is a balancing act between expressivity of the type system and the expressivity of the programming language. This is an example of choosing the right tools for the job. For example, a programming language for proving theorems wants the type system to be as powerful as possible, while type systems retrofitted onto a dynamically typed programming language mainly want guarantees that existing programs are still supported by the programming language.

1.2 Problem Statement

Some statically typed programming languages give developers the possibility to make a distinction between *required* or *optional* object properties. This adds expressivity to the definition of these objects and definitions, but also introduces new kinds of errors that occur when undefined properties are accessed, or required properties are removed. Some statically typed object-oriented languages, such as TypeScript, already ensure the type-safe usage of optional properties.

On top of the distinction between required and optional properties or parameters, the documentation of those objects and functions often contain extra information on which combination of properties or parameters are considered a *valid* combination. For example, a search function might require that *at least one* of the filter criteria is specified. Similarly, an object might only be considered valid if a *group* of properties are all present or all absent: in the Twitter API, a tweet can optionally contain a location, which is indicated with a latitude and longitude property. These two properties may only be provided *together*. As these constraints describe restrictions *between* parameters or properties, we define these as *inter-property constraints*.

Inter-property constraints are prevalent in documentation of web APIs, but also occur in the standard libraries of both dynamically and statically typed languages. However, statically typed programming languages commonly used to develop applications —even those retrofitted for dynamic programming languages— are unable to express such a dependency. Properties can only be marked as op-

tional, which does not suffice to express constraints on the presence of properties that depend on the presence of other properties. Advanced type systems such as dependent types are able to express inter-property constraints. However, dependent types also put nontrivial requirements on the functions of the programming language, which have to be total.

The lack of support for inter-property constraints in commonly-used programming languages leads to type systems that are unable to catch unsatisfied inter-property constraints. Instead, errors are delegated to the runtime. We illustrate this with three examples of `Tweet` objects. The first object is valid: providing the location of a tweet is optional. The second object is also valid as it has both location properties. However, the third object is invalid due to the missing `longitude` property. There currently exists no commonly-used programming language in which we can enforce this constraint.

```
1 var valid1 : Tweet = {text: "Hello, world!"};
2 var valid2 : Tweet = {text: "Hello,", latitude: 50, longitude: 4};
3 var invalid: Tweet = {text: "world!", latitude: 50};
```

As a consequence, programs can contain invalid function calls and objects, and usage of fields which are actually absent.

To guarantee that constraints over multiple fields are satisfied, the programming language must be extended with language constructs for inter-property constraints. Caution is required, as the type system of that programming language must have the logic necessary to guarantee the correct usage of data on which inter-property constraints are imposed. Moreover, the addition of inter-property constraints should have a minimal impact on the expressivity of the programming language such that existing programs are not affected. This lowers the barrier of entry for defining and using inter-property constraints.

1.3 Thesis

Existing statically typed programming languages allow programmers to express constraints over the **presence** of properties or parameters. However, it is not possible to express a dependency logic **between** parameters. A type system that also verifies such inter-property constraints will provide more type safety for developers.

This dissertation supports this thesis by identifying the existence of inter-property constraints in the documentation of existing APIs, showing the practical need for support for inter-property constraints. This dissertation presents a technique for incorporating support for inter-property constraints in the type system

of an existing programming language. Moreover, this dissertation integrates inter-property constraints in a machine-readable specification language and shows how development tools can use this integration to make the web development cycle more robust.

1.4 Approach

We have defined inter-property constraints as constraints between multiple properties of an object. In order to have an accurate view of real-world inter-property constraints, we first study the documentation of several APIs[1] and make an inventory of common patterns and structures in how the constraints are combined.

Given this knowledge on commonly occurring kinds of inter-property constraints, we continue by presenting a type system that supports "presence constraints" over multiple properties of an object. The type system supports any inter-property constraint that can be expressed using propositional logic. It uses several concepts from propositional logic to guarantee type safety. The addition of constraints to interfaces has consequences on several facets of the type system.

- **Definition of objects.** When defining an object type, developers can define constraints on the combinations of properties that are allowed. Object properties can be required to be present or absent, and constraints between the presence of properties are expressed using operators from propositional logic. For example, the constraint `present(latitude) <-> present(longitude)` means that these two location fields must either be present or absent *together*. When object types can inherit from other object types, constraints from the entire inheritance chain are taken into account.

- **Creation of objects.** From a syntactical and semantical point of view, there is no difference between creating an object of a state-of-the-art object type, or an object type with support for inter-property constraints. However, the type system has to perform extra checks: objects can only be of an object type when its constraints are satisfied. The type system uses the concept of a *valuation* and *logical entailment* from propositional logic to perform these checks.

- **Accessing a property of an object.** Similar to the creation of objects, there is no syntactic or semantic difference on how object properties are accessed. On the other hand, guaranteeing the safe access of object properties gets more complex. The type system needs to verify whether the property

[1] Google Maps, Twitter, YouTube, Flickr, Facebook, Amazon

is present or absent. This is inferred by the type system using logical entailment. The type system only assigns the intended type to a property when that property is certain to be present. The type system indicates that a property is known to be absent by assigning it an *absent* type (such as `undefined` in TypeScript and `null` in Java). However, the type system cannot assign any type for properties of which it is uncertain on whether they are present. Therefore, the type system uses flow sensitivity to gain extra information about the presence or absence of properties.

- **Updating a property of an object.** Updating properties of advanced object types also gets more complex, as updating a property may invalidate constraints imposed on other properties. As a consequence, some properties can only be updated safely *together*. For example, removing the `latitude` property may only happen in conjunction with removing the `longitude` property. To solve this, we introduce a new language construct that enables the updating of multiple properties simultaneously, such that an object is never in an invalid state between consecutive assignment statements.

In the design of this type system, we strive to create a type system that is first and foremost usable in web application development and applicable to existing programs and APIs. To this end, we will incorporate inter-property constraints into an existing programming language for the web, without restricting the expressivity of the language. Moreover, the extra type annotations required for inter-property constraints need to be kept at a minimum and as simple as possible. This enables the uptake of inter-property constraints in existing programs.

A side track of this dissertation consists of incorporating the concept of inter-property constraints into other parts of the web development cycle. For example, by translating the textual documentation of a web API to a machine-readable version, several tools (such as code generators) can be generated from the documentation. These tools facilitate the development of web applications.

1.5 Contributions

This dissertation presents a statically typed programming language with support for inter-property constraints. The two main contributions of this dissertation are the following.

Our first contribution is the **identification and classification of inter-property constraints** in documentation of the largest internet companies: Facebook, Twitter, YouTube, Google Maps, Amazon, and Flickr. We present a survey

of real-world documentation that identifies several instances of inter-property constraints, by investigating 688 web API entry points. Moreover, we introduce a classification of commonly found kinds of inter-property constraints.

The second contribution of this dissertation is a new **statically typed programming language with support for inter-property constraints**, called TIPC. This programming language lays the foundation for a programming paradigm that supports inter-property constraints as a distinct entity. As a proof of concept, inter-property *presence* constraints will be incorporated in an existing programming language. More specifically, in TIPC developers are able to express object types in which the presence or absence of properties may depend on each other. Dependencies between properties are defined using propositional logic. The type system of TIPC uses concepts such as valuations and logical entailment from propositional logic to ensure its correctness. The introduction of a new way to define object types has an impact on how objects are created and how its properties are accessed and updated. We provide **proofs of correctness** that prove the type system is sound with respect to enforcing complex dependency logic defined by the programmer when an object is created, modified, or accessed.

Next to these main contributions, this dissertation also explores the incorporation of inter-property constraints in web development. In this context, this dissertation also has three contributions.

Although inter-property constraints occur commonly in the documentation of web APIs, machine-readable languages for web API documentation currently have no support for expressing inter-property constraints. Our third contribution defines a **machine-readable specification language** that supports constraints over multiple parameters. The extra language constructs allow the tools that accompany those languages to support inter-property constraints as well.

This dissertation presents two adaptations of existing specification tools that also serve as a validation. The first tool serves as a **validation of the extended specification language**. Given a number of web API specifications, it generates all the constraints on the data of a given API method and verifies whether a given object satisfies these constraints. As a proof of concept, the preprocessor is accompanied by a tool which verifies —at runtime— that requests in web applications satisfy the constraints imposed on them by the specifications. The second tool serves as a **validation of the type system**. Given a web API specification, this tool generates a stub for the server side implementation of the API. As the implementation is written in TIPC, inter-property constraints are verified at compile-time when using the server stub as a start for the server-side implementation or as mock-up of the server for client-side development.

1.6 Roadmap

This dissertation is structured as follows.

Chapter 2: Inter-property Constraints presents a study of the occurrence of constraints between properties in the documentation of APIs. Examples of inter-property constraints are classified into four categories, depending on how the constraints between properties are combined. Finally, we expand on how APIs react to unsatisfied inter-property constraints.

Chapter 3: Requirements for Inter-property Constraints in Programming Languages describes how inter-property constraints can be incorporated in the interface declaration syntax. Code examples are shown by means of a new programming language (TIPC), but are applicable to other languages as well. To this end, this chapter compiles a set of requirements that form a blueprint for incorporating inter-property constraints in a statically typed programming language.

Chapter 4: Statically Checking Inter-property Constraints informally presents how the TIPC type system satisfies the requirements of Chapter 3. TIPC uses concepts from propositional logic to ensure that new objects satisfy inter-property constraints and that (property) updates do not invalidate inter-property constraints.

Chapter 5: TypeScript's Idiosyncrasies gives information on the idiosyncrasies of TypeScript, the programming language which forms the basis for TIPC. This chapter discusses features that are characteristic to TypeScript and features that are relevant with regards to inter-property constraints.

Chapter 6: TIPC: Formalisation presents the formalisations of TIPC. It defines the syntax, operational semantics and typing rules and presents a proof of soundness.

Chapter 7: TypeScript$_{\text{IPC}}$: Implementation of TIPC describes the implementation of TIPC, called TypeScript$_{\text{IPC}}$. First, this chapters gives an overview of TypeScript compiler, which forms the basis for TypeScript$_{\text{IPC}}$. Next, the differences between TIPC and TypeScript$_{\text{IPC}}$ are discussed. Finally, this chapter gives an overview of the changes made to every phase of the TypeScript compiler in order to incorporate inter-property constraints.

Chapter 8: Related Work situates the work presented in this dissertation in the research on type systems. We discuss how TIPC compares to advanced type systems, as well as existing work on type systems for TypeScript and JavaScript, and occurrence typing.

Chapter 9: Inter-property Constraints in Practice gives an overview of machine-readable API specification languages and discusses their lack of support for inter-property constraints. Next, this chapter introduces a new specification language with support for constraints between properties. Finally, we introduce two tools that verify inter-property constraints, both at runtime and at compile time (using TIPC).

Chapter 10: Conclusion presents our conclusions and discusses avenues for future work.

1.7 Supporting Publications and Technical Contributions

There are two publications that support this dissertation directly:

> Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter Constraints in Contemporary Web APIs. In *Proceedings of the International Conference on Web Engineering*, ICWE 2017, pages 323–335. Springer International Publishing, 2017. ISBN 978-3-319-60131-1

This paper discusses inter-property constraints. It conducts an empirical study that shows that these constraints are common in popular web APIs. Examples of inter-property constraints are categorised into three groups: exclusive constraints, dependent constraints and group constraints. After showing that existing specification languages are not able to express inter-property constraints, the paper introduces a new constraint-centric API specification language that addresses these shortcomings.

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Static Typing of Complex Presence Constraints in Interfaces. In *Proceedings of the 32nd European Conference on Object-Oriented Programming*, ECOOP 2018, pages 14:1–14:27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018b. ISBN 978-3-95977-079-8. doi: 10.4230/LIPIcs.ECOOP.2018.14

This paper introduces a new programming language with object types that support constraints over multiple properties. The programming language is a variant of TypeScript with a novel type system that enables programmers to express complex presence constraints on properties. This paper shows how complex constraints on the presence of interface properties can be statically enforced. We prove that it is sound with respect to enforcing the complex dependency logic used by the programmer when an object is created, modified or accessed.

The following artifact publication supports this dissertation:

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Static Typing of Complex Presence Constraints in Interfaces (Artifact). *Dagstuhl Artifacts Series*, 4(3):3:1–3:2, 2018a. ISSN 2509-8195. doi: 10.4230/DARTS.4.3.3

We have implemented the type system presented in this dissertation as an extension to TypeScript. This implementation is evaluated as part of the ECOOP artifact evaluation on consistency, completeness, reusability and the quality of the documentation. The implementation can also be found at `https://github.com/noostvog/typescriptipc`. We will describe the implementation in more detail in Chapter 7.

Early versions of this dissertation were presented at the following venues:

- Verification of Communication in Web Applications: paper presented at Tools for JavaScript Analysis 2016;
- Typing Third Party Web Service Usage: poster presented at European Conference on Object-Oriented Programming 2016;
- Dynamic Verification of Inter-parameter Constraints in Web Applications: paper presented at International Workshop on Dynamic Analysis 2017;
- Dynamic Verification of Inter-parameter Constraints in Web Applications: poster presented at Systems, Programming, Languages and Applications: Software for Humanity 2017.

Chapter 2

Inter-property Constraints

In application development, libraries are often used to encapsulate functionality, promote reusability or provide functionality to other parties. To communicate with a library, developers use the functions that are provided by its Application Programming Interface (API). Sometimes such an API is accessible as a service over the network, as is the case with web APIs. The API defines the contract between a library and its caller: it imposes requirements on inputs and gives guarantees on outputs.

For example, a standard API that provides functionality for mathematical operators will only accept a call to the plus operator when the provided parameters are all numbers. When dividing numbers, the API requires that the denominator is not zero. For accessing an index in an array, the API requires that the index is not negative.

The examples so far impose constraints on the *value* of the input provided for an API call. However, APIs can also impose constraints on the *presence* of inputs: sometimes, only a part of the properties is required, while the rest is optional. E.g., an API for string manipulation expects three inputs for a call to `substring`: the string, the start index and the end index. The third input typically is an optional one: when it is not provided, `substring` takes the end of the string as default.

As the capabilities offered by an API grow, so do the constraints on the input. This has led to constraints *between* the inputs of an API function. These *inter-property constraints* are common for web APIs, where the presence of one property can determine the structure of other properties in the object of which it is a member, or where the presence of a property even *excludes* other properties. For example, a search function may need at least one criterion to return results.

In this chapter, we survey API documentation from several (web) APIs for

inter-property constraints. The combination of constraints on several properties can be categorised based on operators found in propositional logic. Section 2.1 elaborates on the four logical operators that are commonly used to describe inter-property constraints. To express complex inter-property constraints on properties found in API documentation, several inter-property constraints are combined. We give several examples of combined constraints in Section 2.2. In Section 2.3, we show that inter-property constraints are commonly found in web APIs and Section 2.4 gives an overview on how web APIs deal with unsatisfied inter-property constraints.

Most of this chapter is published in Oostvogels et al. [2017]. However, this chapter elaborates even further on inter-property constraints: it includes more examples (from web APIs, but also from other APIs and languages) of the categories in the paper (exclusivity, dependency and double implication constraints), as well as a new category of inter-property constraints (Section 2.1.4).

2.1 Categories of Inter-property Constraints

In this section, we list several examples of constraints between properties, categorised by the logical operator used to combine the constraints.

Inter-property constraints are often not formally listed as a constraint in the documentation. Instead, they are informally described. In order to identify inter-property constraints in the documentation of web APIs, we have searched for words that might indicate a constraint between properties. For exclusivity constraints, we searched for mentions of *either*, *exactly*, *subsumed* and *one of*. Dependency constraints can be recognised in API documentation by mentions of *additional* and *providing*. Double implication constraints are often indicated with *corresponding* and *providing*.

2.1.1 Exclusivity Constraints

We call an inter-property constraint an *exclusivity constraint* if exactly one of a set of properties is required. E.g., a user can be identified in two ways in the Twitter API: either by his/her screen name (the Twitter handle) or by his/her user ID. As a consequence, every method (or *entry point*) in the Twitter API that needs to identify a user, will have both a `screen_name` property and a `user_id` property. Table 2.1 shows the entry point for sending a direct message in the Twitter API[i] Next to the message itself (the `text` property), the receiver of the private message also needs to be identified. Although both properties are indicated as optional,

Footnotes with roman numbers can be found at the end of this chapter (page 26).

there has to be an exclusivity constraint imposed on these two properties in order to indicate that exactly one of those properties has to be present.

Table 2.1: Excerpt from Twitter API documentation

Property name	Optional?	Description
`user_id`	optional	The ID of the user who should receive the direct message.
`screen_name`	optional	The screen name of the user who should receive the direct message.
`text`	required	The text of your direct message.
Note: One of `user_id` or `screen_name` are required.		

As a second example, Facebook users can publish three kinds of status updates on their wall: a message, a link or a place. The entry point for publishing a status update on a user feed contains the properties `message`, `link` and `place` among others[ii]. Facebook does not indicate whether the properties are required or optional, but in the description they indicate that there is an exclusivity constraints on those tree properties: *"either link, place or message must be supplied"*;

A third example comes from Stripe, an online payment processor: they provide an API to web developers to integrate payments in their websites. One of their most important entry points is `create_charge`[iii], which is used to charge the credit or debit card of customers. The specification for that entry point lists 14 properties: the payer of the transaction can be indicated with either `source` (the data of a credit or debit card) or using the ID of an already registered `customer` (*"either source or customer is required"*).

We show a final example in the YouTube API, which provides functionality to search or retrieve information from its videos, channels, playlists, etc. The entry point to retrieve information about YouTube playlists[iv] contains three ways to identify the playlist that needs to be retrieved: `channelId` to retrieve all playlists from a channel, `id` to retrieve a specific playlist by their unique ID or to retrieve their own playlists (`mine`). The API documentation states to *"specify exactly one of the following parameters"*).

Outside of the documentation of web APIs, exclusivity constraints are also used when there are several ways to identify something. For example, network interfaces can be identified in two ways in the Windows Desktop API: by either providing the `InterfaceLuid` or the `InterfaceIndex`[v]. The API documentation defines this as follows: *"If the InterfaceLuid is specified, then this member is used to determine the interface. If no value was set for the InterfaceLuid member, then the InterfaceIndex member is next used to determine the interface."*

2.1.2 Dependency Constraints

The second category of inter-property constraints are *dependency* constraints, where constraints on a property depend on a characteristic of another property (which we call the *base properties*). In other words, when a constraint is satisfied, this implies that another constraint should also be satisfied. This dependency can be on either the presence of a parameter or its value. There are four sub-categories of dependency constraints.

- Present-Present (PP) dependency constraint: the presence of a property depends on the presence of the base property;
- Present-Value (PV) dependency constraint: the presence of a property depends on the value of the base property;
- Value-Present (VP) dependency constraint: the accepted set of values for a property depends on the presence of the base property;
- Value-Value (VV) dependency constraint: the accepted set of values for a property depends on the value of the base property.

We elaborate on the four kinds of dependency constraints in the rest of this section.

2.1.2.1 Present-Present Dependency Constraints

Table 2.2 shows an example of a PP-dependency constraint in the Facebook API. It shows an excerpt of the entry point in the Facebook Graph API to post a status update[vi]. When the status update is a `link`, the API provides extra properties that can be used to give extra information to accompany that link: a `name`, `caption` and `description` or `picture`. These four properties may only be included when `link` (the base property) itself is also present. Thus, we say that the `link` property is the *base* property for these four other properties in a dependency constraint.

There are two ways to identify a list in the Twitter API[vii]: A developer can either provide the ID of the list, or provide a `slug` (a URL-friendly version of the list name). In the case where the list is identified using a slug, the properties `owner_id` and `owner_screen_name` must also be provided. Moreover, those two properties are only taken into account if the `slug` property is present as well. Note that this is actually a combination of two inter-property constraints: Section 2.2 elaborates on this.

Table 2.2: Dependent constraints in the Facebook API

Property name	Optional?	Description
`link`	optional	The URL of a link to attach to the post. Additional fields associated with `link` are shown below.
`picture`	optional	Determines the preview image associated with the link.
`name`	optional	Overwrites the title of the link preview.
`caption`	optional	Overwrites the caption under the title in the link preview.
`description`	optional	Overwrites the description in the link preview

2.1.2.2 Present-Value and Value-Present Dependency Constraints

A second and third category of dependency constraints are *present-value dependency constraints* (PV-dependency constraint) and *value-present dependency constraints)* (VP-dependency constraints) on properties. In this category, the presence of absence of a property depends on the value of another property, or the other way around. In other words, the value of a property imposes a presence constraint on another property, or the presence of a property imposes a constraint on the allowed values of another property. PV-constraints can always be translated to VP-constraints and the other way around, as `P -> V` is equal to `¬V -> ¬P`.

An example of a PV-dependency constraint can be found in the Google Maps API, which among others provides functionality to render directions[viii]. To customise the rendered directions, Google provides several options. The property `infoWindow` can be used to customise the way information is rendered when a position marker is clicked. However, the property `infoWindow` is ignored when `suppressInfoWindows` is *true*. Conversely: the presence of `infoWindow` depends on the value of `suppressInfoWindows`.

The YouTube API contains another example of a PV-dependency constraint. When managing the moderator status of comment[ix], comments of a specific author can be automatically banned using the `banAuthor` property. This parameter is only valid if the `moderationStatus` property (which indicates the status of the comment itself) is also set to *"rejected"*.

Searching for an item using the Amazon Product Advertising API[x] relies heav-

ily on present-value dependency constraints: not all properties are relevant to certain kinds of searches. The `searchIndex` property is the main property: it indicates the product category for the search. Several other properties only make sense for certain values of `searchIndex`: the property `power` (which is a kind of book search) can only be used when the `searchIndex` is set to *"books"*. The properties `condition`, `minimumPrice` and `maximumPrice` can only be used when the `searchIndex` is different from *"all"* and *"blended"*.

When the result of a search has to be sorted by distance, the Google Maps API[xi] also imposes an PV-dependency constraint: when the property `rankBy` is set to *"distance"*, the `location` property is required. Moreover, this also requires the *absence* of two other properties: `radius` and `bounds`.

Next to web APIs, PV-dependency and VP-dependency constraints are also found in other libraries and APIs. In the Chart.js library, a JavaScript library to draw charts, the property `lineTension` will be ignored if the steppedLine value is set to anything other than *false*[xii].

2.1.2.3 Value-Value Dependency Constraints

The third category of dependency constraints is when the set of allowed values for a property depends on the value of another property.

In the previous section, we have shown that the Amazon API for product advertisement contains several PV-dependency constraints. The API also contains a VV-dependency constraint: when searching for an item, the property `condition` cannot be set to *"new"* when the `availability` property is set to *"available"*.

Another example of a VV-dependency constraint is when there are two properties to indicate a time frame: `startDate` and `endDate`. The allowed values for `startDate` depend on the value of `endDate`: it would not make sense to have a start date *after* the end date.

Similarly, a banking application API might prevent transferring money from and to the same account.

2.1.3 Double Implication Constraints

We classify inter-property constraints as *double implication* constraints when **a set of properties should always occur (or be omitted) together**. This corresponds to a double implication, or equivalence, between two constraints.

Table 2.3 shows a double implication constraint found in the Twitter API: when creating a new tweet[xiii], the user's current location can (optionally) be provided via the `lat` and `long` properties. However, it is an error to pass along

Table 2.3: A double implication constraint in the Twitter API

Property name	Optional?	Description
`lat`	optional	The latitude of the location this Tweet refers to. This parameter will be ignored unless it is inside the range −90.0 to +90.0 (North is positive) inclusive. It will also be ignored if there isn't a corresponding `long` parameter.
`long`	optional	The longitude of the location this Tweet refers to. The valid ranges for longitude is −180.0 to +180.0 (East is positive) inclusive. This parameter will be ignored if outside that range, if it is not a number, or if there is not a corresponding `lat` parameter.

only `lat` *or* only `long`: either both properties are included to specify the location or both properties are omitted.

In Section 2.3.2, we show that double implication constraints are found in many APIs. In Flickr[xiv], for example, the coordinates of a person in a picture can be provided using the properties `person_x`, `person_y`, `person_width` and `person_height`. There is a double implication constraint on these properties: it is optional to give the coordinates of a person, but if you do, all four properties need to be provided.

In the Google Maps API, areas can be identified using a `radius` and a `location`. These properties are dependent on each other, because the area can only be defined when both properties are known.

Double implication constraints are not only imposed on locations: the YouTube API imposes a double implication constraint on two properties that can only be used when the API user is a *content owner*. For example, when creating a playlist[xv], the property `onBehalfOfContentOwnerChannel` must be present when there is a value for the `onBehalfOfContentOwner` property, and the other way around.

The Amazon Product Advertisement API has several double implication constraints for an item search. The `Availability` and `Condition` properties are both optional filters when searching for an item, but both properties should only be used together. Another double implication constraint is imposed when the `RelatedItems` property is used. In that case, the `RelationshipType` also has to be provided to indicate how the items have to be related. When modifying the

shopping cart using the Amazon API, the quantity of an item in the shopping cart can be modified using the properties `CartItemId` and `Quantity`. These two properties can only be used in conjunction with each other.

2.1.4 NAND Constraints

Although most examples of inter-property constraints in this chapter are from web API documentation, inter-property constraints also occur in other contexts. In the Python standard library, the function `os.utime`[xvi] sets both the access and modification time of a file. The documentation describes that the function takes two optional parameters to set the time: `times` and `ns`. Moreover, it states that "*It is an error to specify tuples for both `times` and `ns`*".

2.2 Combined Constraints

The previous section showed how constraints between properties can be categorised using logical connectives, more specifically: XOR, (double) implications and NAND. However, sometimes several logical connectives between properties need to be combined to express the constraints that are found in documentation. Defining a combination of constraints that correctly represent the requirements expressed in the documentation is not trivial. In this section, we provide several examples of constraints on the presence of properties that cannot be expressed using a single logical connective.

The following quote from the documentation of the Twitter API explains how to refer to a user list on Twitter.

> *"You can identify a list by its `slug` instead of its `list_id`. If you decide to do so, note that you will also have to specify the list owner using the `owner_id` or `owner_screen_name` parameters."*

This sentence denotes a dependency constraint between `slug` (an URL-friendly version of the list name) and *two* fields (`owner_screen_name` and `owner_id`), which have an exclusivity constraint imposed on them in turn. There is also an exclusivity constraint between these three fields (the slug and its owner) and the `list_id` field. Figure 2.1 shows a visualisation of this constraint.

Using logical connectives, we would like to write this down as follows:

$$\begin{cases} \texttt{list\_id XOR slug} \\ \texttt{slug <-> (owner\_screen\_name XOR owner\_id)} \end{cases}$$

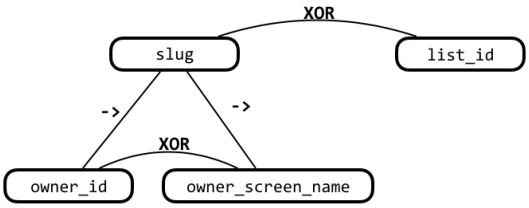


Figure 2.1: Visualisation of a combined constraint

Using `present`, the propositions in this logical formula denote the presence (or absence) of a property in a request. Analogous, ¬`present(x))` requires that the property `x` is not part of the request properties. For example: the constraint `present(user_id) XOR present(screen_name)` can only be satisfied when either the ID or the name is provided as part of the request properties.

The constraint listed above is an almost literal translation from the documentation to a logical formula: either the ID or the slug must be used to identify the list (left-hand side of the AND), and in the case of a slug, either an ID or a screen name must be used to identify the receiver. Moreover, these two properties should only be present when slug is present as well.

However, this is subtly wrong: the constraint is also valid if every field except `slug` is present. This is not the desired outcome as the owner of the slug needs to be identified as well. In the case that all fields are present except for `slug`, the logical formula showed earlier is resolved as follows:

```
1 (true XOR false) AND (false <-> (true XOR true))
2  true            AND (false <-> false)
3  true            AND true
4  true
```

Listing 2.1: Valuation of logical formula with all fields present except `slug`

It is possible to come up with alternative formulations, but the reader needs to construct a truth table in order to convince him or herself.

Nested logical formulas often give unexpected results and should be used with care. Instead, this constraint may be written down as a set of smaller, non-nested constraints. This set of constraints is not as concise, but it is correct.

$$\begin{cases} \texttt{slug XOR list\_id} \\ \texttt{slug -> (owner\_screen\_name XOR owner\_id)} \\ \texttt{owner\_screen\_name -> slug} \\ \texttt{owner\_id -> slug} \end{cases}$$

[2] Inter-property Constraints

Recalling the example where every field except for `slug` is present, this logical formula is resolved as follows. This logical formula evaluates to `false` when all fields except `slug` are present, as desired.

```
1 (false XOR true) AND (false -> (true XOR true)) AND
2                  (true -> false) AND (true -> false)
3 true AND (false -> false) AND false AND false
4 true AND true AND false AND false
5 false
```

Listing 2.2: Valuation of the correct logical formula, with all fields present except `slug`

During the course of experimenting with inter-property constraints (and accompanying examples), we found it beneficial to decompose nested constraints into conjunctions of simpler constraints. By separating constraints that do not strictly need to be combined, they often reflect the desired outcome better.

Section 2.1.3 listed an example of a double implication constraint in the Google Maps API: to indicate an area, `radius` and `location` have to be used *together*: `present(radius) <-> present(location)`. However, some entry points in the Google Maps API allow a location to be identified using `bounds` instead. Furthermore, when `bounds` is used to identify an area, the two properties `radius` and `location` will be ignored. In those cases, it is incorrect to use a double implication constraint between `radius` and `location`, when it becomes a part of an exclusivity constraint.

```
bounds XOR (radius <-> location)
```

This constraint will also be valid when none of the three area properties are provided:

```
1 false XOR (false <-> false)
2 false XOR true
3 true
```

The correct logical expression for this constraint uses the AND connective between `radius` and `bounds`:

```
bounds XOR (radius AND location)
```

This constraint resolves as follows when all three properties are absent:

```
1 false XOR (false AND false)
2 false XOR false
3 false
```

Section 2.1.1 contains an example of an exclusivity constraint in the Facebook API: exactly one of `message`, `link` and `place` should be provided for a status update. In Section 2.1.2, we have seen that the Facebook API puts a dependency constraint on several properties that provide details for the `link` property. These two kinds of constraints can be safely expressed by simply combining them. Note that it is incorrect to translate the exclusivity constraint between the three kinds of status updates as `message XOR link XOR place` as this is also valid when all three arguments are true. The second part of the first formula ensures that the case with three present status properties is not accepted. An alternative to the notation in the listing below is to explicitly state the three allowed combinations of the status updates.

```
(((message XOR link) XOR place) XOR (message AND link AND place))
picture -> link
name -> link
caption -> link
description -> link
```

This section shows that translating the inter-property constraints found in documentation to logical formulas needs to be done carefully. There already exist tools that aid in verifying whether the logical formula correctly defines the desired constraint, such as a *truth table generator*[1], which generates a truth table for some logical formula. This facilitates reasoning about which combinations of properties are accepted or rejected by the logical formula. In light of inter-property constraints, these tools could be extended such that it generates a set of interfaces where each interface contains a valid combination of present and absent properties. We also envision a tool that generates inter-property constraints, given a list of properties and a set of accepted combinations of these properties.

2.3 Empirical Study of Inter-property Constraints in Web APIs

Many of the examples in the previous sections of this chapter contain excerpts taken from web API documentation. In this section, we first briefly explain how an application typically communicates with a web API. Next, we perform a small empirical study on the presence of inter-property constraints in web APIs. Section 2.3.2 shows the results.

[1]For example `http://turner.faculty.swau.edu/mathematics/materialslibrary/truth/`

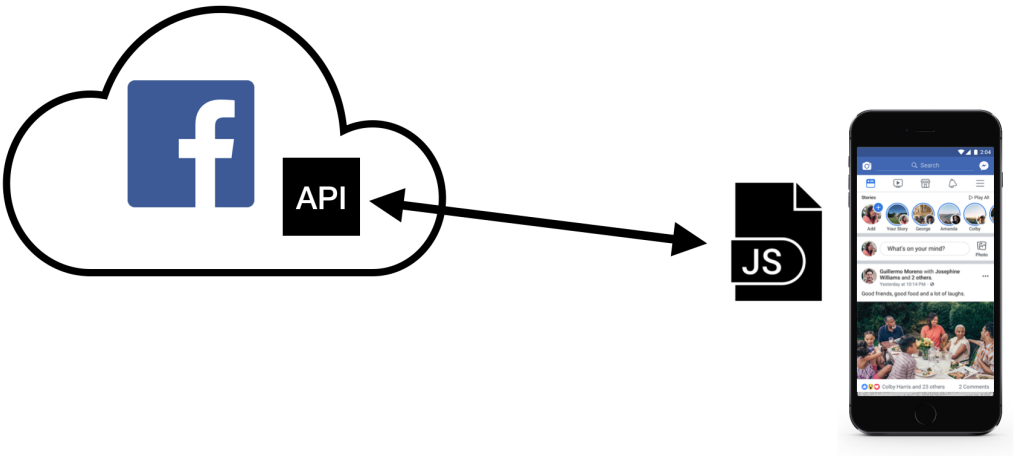


Figure 2.2: Diagram of an interaction between a web API and an application

2.3.1 A Primer on Web APIs

In order to integrate the functionality of a web service (such as Facebook, Twitter,...) in an application, the application needs to communicate with the web API exposed by the web service. Figure 2.2 shows a diagram of this process. We explain the aspects of the diagram in the following paragraphs.

The phone on the right-hand side of Figure 2.2 depicts the mobile Facebook application, but the application may be any desktop application, web application, or mobile application. In recent years, technologies such as Adobe PhoneGap, Electron and React Native have enabled developers to program all platforms in the JavaScript programming language. This language natively supports interaction with web services.

The left-hand side of Figure 2.2 depicts the Facebook web service alongside its API. The API provides functionality via a set of *entry points*. Every entry point has a location (a URL), requirements for the input properties and produces a result with a given shape. Typically, this information can be found in the web API documentation.

Finally, the communication between the application and the web service can be done by sending an HTTP request. This request identifies the user, the desired entry point and contains the input data. If all goes well, the web service accepts the data, performs the required operation and produces a result. Otherwise, an error message is produced.

2.3.2 Results of the Empirical Study

To investigate how frequently inter-property constraints occur in web APIs "in the wild", we manually analysed the documentation of web APIs. For this study, we selected the six most popular APIs of ProgrammableWeb[2]. This number is based on the usage of these APIs in mashups[3]. Other catalogs and metrics exist, such as API Harmony[4] and Mashape's PublicAPIs[5]. API Harmony lists the 5 most popular web APIs based on their usage in GitHub projects. PublicAPIs does not mention which metrics they use for sorting APIs by popularity.

The six most popular APIs of ProgrammableWeb (in July 2018) correspond to popular web applications and social media:

1. **Google Maps JavaScript API** (version 3): an API for viewing details on maps and rendering directions;

2. **Twitter REST API** (version 1.1): an API of a social network website to publish and search tweets, as well sending private messages;

3. **YouTube API** (version 3): an API for viewing, uploading and sharing videos, as well as interactive features such as commenting;

4. **Flickr** (no versioning available, data is from 2016-06-01): an API for an image hosting website, with functionality to upload photos and add extra information;

5. **Facebook Graph API** (version 2.8): an API of a social network website for publishing status updates on user feeds, as well as uploading and sharing content;

6. **Amazon Product Advertisement API** (version 2013-08-01): an API for looking up items on the Amazon web store, as well as managing shopping carts.

Table 2.4 on page 27 summarises our results. For every web API, the table lists the number of entry points that contain an exclusivity, dependency or double implication constraint. Note that the actual amount of inter-property constraints may be higher, because one entry point may contain several instances of a kind of inter-property constraints. For example, the Amazon Product Advertisement

[2] `http://www.programmableweb.com/apis/directory`
[3] Mashups are web applications that combine functionality of different web APIs.
[4] `https://apiharmony-open.mybluemix.net/`
[5] `https://market.mashape.com/explore`

API has 6 instances of value-present dependency constraints in the entry point for searching items. Some entry points in Twitter also contain several exclusivity constraints, for example when multiple users need to be identified.

Table 2.4 shows that the documentation of all six most popular web APIs contain exclusivity constraints, as well as double implication constraints. Except for Flickr, all APIs also have dependency constraints in their documentation. We summarise the rest of the results of our empirical study per type of inter-property constraint category:

- **Exclusivity constraints** are the most common kind of inter-property constraint in web API documentation, with a total of 77 occurrences. Especially Twitter uses exclusivity constraints extensively: one out of three entry points of the Twitter API contain one or more occurrences of exclusivity constraints. In the YouTube API, one out of five entry points have an exclusivity constraint imposed on their properties.

- Every API has **dependency constraints** in their API documentation, apart from Flickr. Occurrences of dependency constraints are subdivided into the three categories: present-present, value-present or present-value and value-value dependency constraints. Dependency constraints between the presence of two properties are the most common kind of dependency constraint, while only the Amazon API has instances of dependency constraints between the value of properties.

- **Double implication constraints** occur in all the APIs we investigated, but they are less often found in web API documentation compared to exclusivity and dependency constraints.

To conclude, inter-property constraints are commonly found in the documentation of web APIs. We have found multiple instances of exclusivity constraints, dependency constraints and double implication constraints in the documentation of the six popular web APIs. Exclusivity constraints were most commonly found.

2.4 Violations of Inter-property Constraints

So far, this chapter discussed the concepts behind inter-property constraints and how to identify them. In this section we discuss a different aspect, namely the recovery strategies employed by the largest API providers: how do they react to unsatisfied inter-property constraints.

The empirical study in the previous section shows that inter-property constraints are common in web APIs. Satisfying the constraints set by the API

providers is essential for a request to succeed. Developers have to rely on the API provider to respond with a meaningful error message in case of a malformed request, or they are forced to manually verify each request in the application. The problem with the former is that this means that bugs can only be identified after deployment of the application. Additionally, this approach requires full coverage of every API request by the application's test suite. Furthermore, every API provider responds differently — and not always with an error message — to requests that *do not* satisfy its constraints. In this section, we classify the responses to unsatisfied inter-property constraints in three categories:

1. **The API provider returns an *error message*:** in the best-case the API provider returns a meaningful error message whenever inter-property constraints are not satisfied. Unfortunately, this is not always the case. For example, when the exclusivity constraint from the YouTube API is not met by supplying more than one filter for a playlist, the following error message is returned: *"Incompatible parameters specified in the request"*. Twitter returns a more detailed error message when a dependency constraint is not satisfied: *"You must specify either a list ID or a slug and owner"*. For unsatisfied double implication constraints, Flickr returns as error message: *"Some co-ordinate parameters were blank"*.

2. **The API provider makes a *silent choice*:** API providers can opt to tolerate certain malformed requests in order to be compatible with a wider variety of clients. For example, Twitter does not complain when both the screen name and user ID are passed along when sending a direct message. However, when the screen name and the user ID belong to different users, Twitter chooses the screen name and silently ignores the user ID instead of raising an error. The same applies for double implication constraints present in the Twitter API: if not all double implication properties are present, all incomplete double implications are ignored. Similarly, Facebook just silently ignores all the dependency properties when the *base* property is not provided. These kinds of errors are very difficult to debug, because the developer does not receive any feedback about the incorrect requests. Moreover, these kinds of responses are closely linked to the particular implementation of the API, which can be changed without warning or API version update. This can cause code that previously ran as expected to suddenly break without further explanation.

3. **The API documentation is incorrect**: in the case of Facebook, where their API documentation mentions the exclusivity constraint *"either link,*

place or message must be supplied" for publishing a status update, supplying all properties *does* result in a sensible status update, where all provided values are combined.

2.5 Conclusion

In this chapter, we have surveyed the documentation of (web) APIs and libraries and identified the existence of *inter-property constraints*: constraints between a set of properties. A constraint on one property is combined with other constraints using the traditional operators from propositional logic. We have given several examples of exclusivity constraints (XOR), dependency constraints (implication), double implication constraints and NAND constraints. Correctly expressing constraints in documentation sometimes requires a combination of several logical connectives.

Next to the multiple examples of inter-property constraints, this chapter shows a small study that indicates that inter-property constraints are present in modern web APIs. Furthermore, the way web APIs respond to requests that do not satisfy constraints is not always well-defined. The service will either respond with an (often vague) error message or silently ignore part of the request. These diverse ways of responding to invalid requests stem from a divergence between the documentation of an API and its implementation.

Ideally, there would be support for inter-property constraints in all aspects of developing applications that use these libraries and (web) APIs. In the following chapters, we introduce a new statically typed programming language with support for inter-property constraints. By incorporating inter-property constraints into interface definitions, developers can rely on the type system to ensure that the inter-property constraints are satisfied whenever they call a function. In Chapter 9, we introduce a new machine-readable specification language for web APIs that also supports inter-property constraints. In combination with the programming language used to implement the client, the tools that accompany the API specification language (such as code generators) can translate constraints from the specification into interface definitions. This ensures continuity in the entire web application development process.

Table 2.4: Inter-property constraints in web APIs

	XOR	Dependency			Double Implication	# entry points
		PP	VP/PV	VV		
Google Maps JavaScript API	10	1	2	0	3	117
Twitter REST API	31	14	0	0	6	97
YouTube Data API	11	2	2	0	5	50
Flickr API	12	0	0	0	1	206
Facebook Graph API	11	4	0	0	1	209
Amazon Product Advertisement API	2	1	2	1	2	9

Notes

i. https://dev.twitter.com/rest/reference/post/direct_messages/new,
Twitter deprecated this entry point in August 2018;

ii. https://developers.facebook.com/docs/graph-api/reference/v2.8/user/feed,
Facebook Graph API version 2.8;

iii. https://stripe.com/docs/api/node#create_charge,
Stripe API version 2015-02-18;

iv. https://developers.google.com/youtube/v3/docs/playlists/list,
YouTube Data API version 3, 2017-11-16;

v. https://docs.microsoft.com/en-us/windows/desktop/api/netioapi/nf-netioapi-getipinterfaceentry,
Microsoft Desktop API version 2018-05-12;

vi. https://developers.facebook.com/docs/graph-api/reference/v2.8/user/feed,
Facebook Graph API version 2.8;

vii. https://dev.twitter.com/rest/reference/post/lists/members/create,
Twitter API version January 2019;

viii. https://developers.google.com/maps/documentation/javascript/reference#DirectionsRenderer,
Google Maps Platform API, version 3.35, 2018-12-19;

ix. https://developers.google.com/youtube/v3/docs/comments/setModerationStatus,
YouTube Data API version 3, 2017-11-16;

x. http://docs.aws.amazon.com/AWSECommerceService/latest/DG/ItemSearch.html,
Amazon Web Services Product Advertising API, version 2013-08-01;

xi. https://developers.google.com/places/web-service/search,
Google Maps Places API, version 2018-11-02;

xii. https://www.chartjs.org/docs/latest/charts/line.html#stepped-line,
chart.js API version 2;

xiii. https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update,
Twitter API version January 2019;

xiv. https://www.flickr.com/services/api/flickr.photos.people.add.html, Flickr API version January 2019;

xv. https://developers.google.com/youtube/v3/docs/playlists/insert, YouTube Data API version 3, 2017-11-16;

xvi. https://docs.python.org/3/library/os.html#os.utime, Python standard library version 3.7.2.

Chapter 3

Requirements for Inter-property Constraints in Programming Languages

The previous chapter introduced the concept of inter-property constraints, i.e. constraints between multiple properties. They are common in the documentation of (web) APIs as part of the prerequisites of API methods. As applications contain many calls to many different APIs, manually verifying these prerequisites becomes an error-prone task. Relying on error messages that result from incorrect API calls is not always possible either: Section 2.4 showed that API calls with unsatisfied inter-property constraints silently fail or return only a vague error message.

In order to help developers, we would like to automatically verify as many constraints as possible, as early as possible. There are several approaches to enable the automatic verification of inter-property constraints: static type systems and manifest contracts enable compile-time verification, while runtime assertion checkers and latent contracts enable runtime verification.

We base our approach on static type systems — and more specifically the type system of TypeScript, a statically typed variant of JavaScript— for four reasons. First, compile-time approaches meet the "as early as possible"-criterium. Second, TypeScript has enjoyed massive adoption for web- and server-side applications as a replacement for JavaScript. In 2018, GitHub marked TypeScript as its third fastest growing language [GitHub]. Third, the fields listed in the documentation of web APIs are often converted to data structure types, which embody a majority of the field constraints. Finally, as this chapter will show, incorporating inter-property constraints in a programming language will require new programming idioms.

```
interface PrivateMessage {
    text        : string;
    user_id?    : number;
    screen_name?: string;
}
```

Listing 3.1: TypeScript interface for the specification in Table 2.1

TypeScript's static type system already enables developers to automatically verify constraints from the documentation, by translating them to types for variables. The type system checks at compile time that constraints of types are satisfied for the variables. For object types in particular, these languages provide guarantees about structure: which properties have to be or may be present, and what their type is. By translating constraints on properties in the documentation of (web) APIs to object types, the type system will guarantee that these constraints will be satisfied *before* the application is executed. Unfortunately, state-of-the-art interfaces are limited to express constraints on only one property at the time. As a result, it is impossible to express inter-property constraints: constraints between a set of properties.

For example, in TypeScript (and also in other languages) it is impossible to express that *exactly one* of `user_id` and `screen_name` is required. As shown in Listing 3.1, the properties `user_id` and `screen_name` can only be denoted as *optional* properties, using question marks.

This means that the type system also accepts objects containing none or both of the user properties! Similarly, the double implication constraint with latitude and longitude properties for the location of a Tweet cannot be expressed: one can mark both properties as optional, but the type system will not reject the program when only one property is provided.

In this chapter, we introduce a statically typed programming language with support for object types with inter-property constraints on their properties. More specifically, this chapter shows how inter-property constraints can be incorporated into TypeScript[1], yielding the programming language TIPC. While the examples in this chapter are all written in TIPC, the concepts can be generalised to other programming languages as well: this chapter compiles a list of requirements that need to be fulfilled in order to so.

There is an impact on how objects are created and updated, as well as how properties are accessed and updated. For every part of the object life-cycle, this chapter explains which guarantees we expect to get from the type system. The

[1]We refer readers that are not familiar with TypeScript to Chapter 5.

overall goal is that the type system makes optimal use of the information provided by the program about the structure of objects. We aim to have a minimal impact on the existing expressivity of the TypeScript programming language, as well as minimal changes to its syntax. This way, existing programs can be easily extended with inter-property constraints.

The text in this chapter is published in Oostvogels et al. [2018b] (Section 1 and 2). Chapter 8 elaborates on type system research that enables the expression of inter-property constraints. It shows that they have a big impact on language expressivity or the simplicity of the type annotations in order to ensure type soundness for objects with inter-property constraints.

3.1 Interface Definition

Introducing inter-property constraints into common object-oriented programming languages has an impact on the way interfaces are declared. Syntax-wise, this is the most significant difference from TypeScript and other OO-languages. Interfaces in TIPC consist of two parts:

- **Property list:** The first part of the interface declaration contains the list of properties, together with the type for each property. Contrary to how interfaces are defined in regular OO-languages, this part does not impose any restrictions on the presence of these properties. As a consequence, all properties are optional by default rather than required by default.

- **Constraint list:** the second part of the interface declaration contains the constraints on properties. Objects are valid implementations of interfaces if all its constraints are satisfied. The kind of constraints that are supported by TIPC are limited to *presence constraints*.

  Presence constraints in the interface definition are not limited to constraints on a single property. Indeed, separating presence constraints from the property list allows developers to express constraints over multiple properties as well: constraints on the presence of a property can be combined with logical connectives.

This allows us to express the presence constraints imposed on the interface `PrivateMessage` correctly. In Listing 3.1 we could only indicate that the user properties `user_id` and `screen_name` as optional. Listing 3.2 shows an example of an interface declaration in TIPC, revisiting the Twitter specification for sending private messages.

```
1 interface PrivateMessage {
2     text        : string;
3     user_id     : number;
4     screen_name: string;
5 } constraining {
6     present(text);
7     present(user_id) xor present(screen_name);
8 }
```

Listing 3.2: Twitter private messaging API properties expressed as interface with constraints

$$\texttt{c} \in \text{Constraints} ::= \texttt{present(n)} \mid (\texttt{c}) \mid \texttt{c} \wedge \texttt{c} \mid \texttt{c} \vee \texttt{c} \mid \neg\texttt{c} \mid \texttt{c} \rightarrow \texttt{c} \mid \texttt{c} \leftrightarrow \texttt{c} \mid \texttt{c xor c}$$

Figure 3.1: Syntax for expressing constraints

Lines 2–4 list the three properties for `PrivateMessage`. Lines 6 and 7 denote the constraints on the presence of those three properties. The `PrivateMessage` interface lists two presence constraints: line 6 requires the presence of the `text` property and line 7 is the inter-property constraint between `screen_name` and `user_id`.

The exact syntax for defining constraints in interfaces is defined in Figure 3.1. A required property `p` is indicated with `present(p)`, an absent property with `¬present(p)`. The presence and absence of properties can be combined using the following logical connectives: conjunctions, disjunctions, implications, double implications and exclusive disjunctions. The syntax is very close to propositional logic, which is concise, well-understood, and lends itself to exploring the space of possible object by means of truth tables. It allows developers to describe all presence constraints that were listed in Chapter 2, but also to create other kinds of constraints. Constraints may only refer to properties defined in the interface or any of its superinterfaces.

As opposed to TypeScript and many other languages — where properties are required by default but can be made optional with a `?` annotation — properties in TIPC are *optional by default*. This is a consequence of moving *all* constraints on the presence of properties to the second part of the interface definition. Note that the constraint definition language does not list optional properties as an explicit constraint operation, as this can be expressed by the following constraint: `present(n)`$\vee$`¬present(n)`. This constraint is a tautology, and can thus be omitted

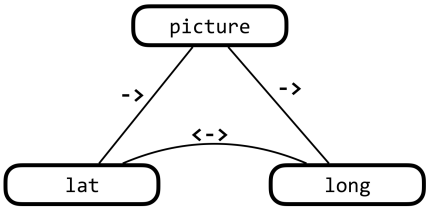


Figure 3.2: Visualisation of a combined constraint

from the constraints of an interface definition.

Although the syntax for defining constraints is fairly straightforward, writing correct constraints remains the responsibility of the developer. Section 2.2 showed that caution is advised when constructing inter-property constraints: especially combined constraints need to be composed carefully. Developers can always check whether the intended constraints correspond to the constraint definition, using truth table generators.

At the very least, the developers need to ensure that the set of constraints is *satisfiable*: there should be at least one combination of present and absent properties that satisfies the constraints.

Listing 3.3 shows another example of inter-property constraints. It describes an interface `Tweet` with four properties: the text for the tweet, a picture and `lat` and `long` to indicate the location. The text is a required property, which is indicated with the constraint on line 7. The picture and location properties are optional. The inter-property constraints on this interface are visualised in Figure 3.2. The properties `lat` and `long` are dependent on the `picture` property: if the picture itself is not provided, the location has to be omitted as well. In other words: the presence of the location properties implies that the picture must be present as well. These constraints are defined on lines 8 and 9. Moreover, the latitude and longitude properties are present or absent *together*, which is indicated by the constraint on line 10.

Before we explore the repercussions of the new interface definition, we briefly discuss why existing language features do not suffice:

Defining interfaces with inter-property constraints using union types This section shows a new kind of interface definition to enable the definition of inter-property constraints. An alternative to this approach is to use existing interface definitions and combine them using a union type. With this approach, each interface definition encodes one valid combination of present and absent

```
1  interface Tweet {
2      text    : string;
3      picture: string;
4      lat     : number;
5      long     : number;
6  } constraining {
7      present(text);
8      present(lat)  → present(picture);
9      present(long) → present(picture);
10     present(lat)  ↔ present(long);
11 }
```

Listing 3.3: Interface with dependency and double implication inter-property constraints

properties. For example, the presence constraints for a Twitter private message can be translated to a union of two interfaces: one with the properties `text` and `user_id`, and one with the properties `text` and `screen_name`.

At first sight, this looks like a feasible alternative to the interface definitions proposed in this section. However, this approach is only achievable for interfaces with a small number of constraints. As the number of constraints grow, the complexity increases as well. For example, adding a member to a list in Twitter imposes two inter-property constraints: the exclusivity constraint on the user ID and screen name (to identify the member) and the complex constraint to identify the list (as explained in Section 2.2). Combining these constraints results in 6 valid combinations of present and absent properties! Therefore, this approach would result in a set of interfaces in which developers themselves need to distil the constraints between properties. By defining constraints explicitly in the interface definition, it is clear which presence constraints are imposed on the properties.

Function definitions with inter-property constraints In this chapter, inter-property constraints are incorporated in *interface* definitions. Similarly, inter-property constraints can also be imposed on the arguments of *function* definitions. The requirements in this chapter – which discuss the creation and assignability of objects as well as the accessing and updating rules of their properties – can be applied to functions and their arguments as well. Note that function overloading suffers from the same disadvantages as the ones discussed in the previous paragraph.

Moreover, function overloading suffers from an additional problem, as overloading relies on the dispatching of *types*. For example, an exclusivity constraint

on two numbers would result into two functions where each function has one argument with type number. In this case, it is impossible to rely on function overloading.

3.2 Creating Interface Instances from Object Literals

When an object literal is assigned to an expression of an interface type, TIPC ensures that the object literal satisfies the interface constraints.

Listing 3.4 shows how three objects are created and assigned to three variables of type `PrivateMessage`. Note that, even though the interface contains inter-property constraints, the object creation does not change for the programmer on a syntactical level. To type check this code snippet properly, the type system has to verify that the interface constraints are satisfied for that object. In the example, the first object (`msg1`) satisfies all constraints: `text` is present, and the exclusivity constraint is satisfied as well: only `user_id` is passed along as identification for the user. However, the type system will generate errors for `msg2` and `msg3`, as they both violate the exclusivity constraint.

```
1  let msg1: PrivateMessage = {text: "Hello", user_id: 42}; // correct
2
3  let msg2: PrivateMessage = {text: "Hello"};
4   // error: the constraint (user_id xor screen_name) is not satisfied
5
6  let msg3: PrivateMessage = {text: "Hello",
7                              user_id: 42,
8                              screen_name: "Alice"};
9   // error: the constraint (user_id xor screen_name) is not satisfied
```

Listing 3.4: Creating objects with inter-property constraints

Requirement 1. Instantiating Interfaces with Object Literals

The type system allows the instantiation of an interface with an object literal only when that object literal satisfies the interface constraints.

It is straightforward to statically verify which properties are present and absent when there is a *fresh* object literal on the right-hand side of the assignment. However, it becomes more difficult to verify when the right-hand side of the assignment contains an object literal that is created earlier in the program (i.e. not fresh). Some object-oriented programming languages (such as TypeScript) employ

width subtyping[2], which allows objects to contain more properties than listed in their type. Because of width subtyping, the type of the object on the right-hand side of the assignment might not exactly match with the actual properties of the object at runtime.

The following listing shows an example. First, an object literal with all three private message properties is assigned to a variable whose type is an object literal type containing only `text` and `user_id`. In object-oriented languages with width subtyping (such as TIPC), this is a valid assignment. As a consequence of supporting width subtyping, TIPC rejects the second assignment. On first sight, an object of type {`text: string, user_id: number`} satisfies the constraints of `PrivateMessage`, but at runtime this instance of `PrivateMessage` would contain all three properties.

```
1  let obj: {text: string, user_id: number}
2              = {text: "Hello", user_id: 42, screen_name: "Alice"};
3    // allowed by type systems with width subtyping
4
5  let msg4: PrivateMessage = obj;
6    // rejected by the type system even tough the type
7    // of 'obj' only contains one of the user properties
```

Listing 3.5: Width subtyping complicates verifying inter-property constraints

In order to remain compatible with existing applications written in TypeScript, TIPC has to support width subtyping. However, in the light of inter-property constraints, TIPC also needs to ensure that the runtime value of any object with inter-property constraints imposed om them will never contain more properties than allowed by the constraints.

3.3 Accessing Object Properties

When inter-property constraints are involved, accessing object properties requires extra caution. When a type system allows a property access, a developer can assign new values to this property, given that their types are compatible. Therefore, it is crucial that the type system assigns a type to the property access that corresponds to the actual value of the property.

Required properties (i.e. a property `p` in an interface with a `present(p)` constraint) are guaranteed to be present in the object. When they receive a new value of the same type, none of the constraints will be affected. Absent properties (i.e. a property in an interface with a ¬`present(p)` constraint) are guaranteed to be

[2] A detailed explanation of width subtyping can be found in Section 5.4.

absent in the object. In TypeScript and TIPC, a property `p` is treated as absent when it is not a part of the object or when its runtime value is equal to `undefined`. Thus, an absent property may only receive `undefined` as a new value.

In some cases, it is impossible to infer whether a property is present or absent. For example, when the only constraint of an interface is `present(a) -> present(b)`, there is no guarantee that `a` or `b` will be present or absent. In this case, it is impossible to assign an exact type that will correspond to the value of that property. Therefore, TIPC rejects accesses to those kinds of properties in order to preserve type safety.

Requirement 2. Property Access

The type system assigns a type to a property access expression that correctly reflects the value of that property. When it is impossible to predict the exact type of the property, the type system rejects the property access.

Listing 3.6 shows a couple of examples of accessing properties of a `PrivateMessage` object. The property `text` in the `PrivateMessage` interface is a required property and thus it is certain this property is always present in objects of type `PrivateMessage`. Thus, TIPC allows the access of `text` (line 2) and assigns the corresponding type. By contrast, TIPC rejects accessing the user properties of a `PrivateMessage` object. The exclusivity constraint guarantees that exactly one of `user_id` and `screen_name` will be present, but it is not known *which* property actually is present.

```
1 function getInfoPM(msg: PrivateMessage) {
2   msg.text;        // :: string
3   msg.user_id;     // error: user_id not guaranteed to be present
4   msg.screen_name; // error: screen_name not guaranteed to be present
5   //...
6 }
```

Listing 3.6: Accessing properties

When the presence or absence of a property cannot be inferred from the presence constraints, the developers cannot access or update that property. However, a common pattern in dynamic languages such as JavaScript and languages with optional properties such as TypeScript is to perform tests that verify whether a property is present or absent.

```
1  function getUser(msg:PrivateMessage) {
2   if (msg.user_id !== undefined) {
3     msg.user_id;       // :: number    (present due to if statement)
4     msg.screen_name;   // :: undefined (not present due to xor constr.)
5   } else {
6     msg.user_id;       // :: undefined (not present due to if stmnt.)
7     msg.screen_name;   // :: string    (present due to xor constraint)
8   }
9   //...
10 }
```

Listing 3.7: Accessing properties

Requirement 3. Presence Test

The type system extracts run-time type information from `if` statements and takes it into account when verifying property accesses.

When an `if` statement tests the presence of a property, this property can be safely accessed inside the consequent of that `if` statement. Moreover, the inverse (property absence) has to be taken into account in the `else` statement of that `if` statement. On top of that, the extra knowledge from the `if` statement may trigger extra knowledge on the presence of other properties. The type system of TIPC infers these implicit presence or absence of properties and takes them into account when properties are accessed.

For example, the function `getUser` in Listing 3.7 first performs a test to check whether `user_id` is present. Inside the true branch, access to the user ID (line 3) is allowed by TIPC. In the false branch, the `user_id` is absent, which is indicated by the TIPC by assigning `undefined` to `msg.user_id`. Additionally, because there is an exclusivity inter-property constraint between `user_id` and `screen_name`, the `screen_name` property is guaranteed to be absent in the true branch, even though the programmer did not explicitly test for it. Therefore, TIPC assigns `undefined` to `msg.screen_name`, instead of disallowing that property access. The inverse holds in the false branch: given the absence of `user_id`, `screen_name` will certainly be present and is thus typed as `string`.

Testing the presence of one property may indirectly give information on the presence of *other* properties. Listing 3.8 shows a function `getLocation`, which retrieves the longitude and latitude of a picture. Inside the function, there is one `if` statement that verifies the presence of `long`. TIPC allows the access of `long` (line 3), which follows directly from the `if` statement. On top of that, TIPC also

```
1 function getLocation(picture: Picture) {
2  if (picture.long !== undefined) {
3    picture.long;     // :: number (present due to if statement)
4    picture.lat;      // :: number (present due to group constr.)
5    picture.picture;  // :: string (present due to dependency constr.)
6  }
7  //...
8 }
```

Listing 3.8: Accessing properties

accepts accessing the properties `lat` and `picture`, which are both guaranteed to be present if `long` is present.

3.4 Assigning Instances of Interfaces to Others

Earlier in this chapter, we showed how objects are assigned to variables of certain interface types. Next to objects, variables of interface types can also be assigned to other variables of interface types. There are two strategies an object-oriented language can chose from for verifying assignments: based on the name of the object type (nominal typing) or based on the structure of the object type (structural typing). The most interesting strategy is structural typing. When interfaces are structurally typed, this means that two interfaces variables can be assigned to each other as long as they are structurally compatible[3].

In TIPC, interfaces have complex constraints which express the presence of properties. These constraints have to be taken into account when comparing the structure of two interfaces. On the constraints level, the type system only allows objects with stricter presence constraints to be assigned to variables with less strict presence constraints.

Requirement 4. Assigning Instances of Interfaces to Variables of Interface Types

The type system ensures that no constraints are violated when an interface instance is assigned to a variable of an interface type.

For example, Listing 3.9 contains an alternate version of the `PrivateMessage`

[3] A more detailed explanation on structural and nominal typing in TypeScript can be found in Section 5.4.

```
1 interface PrivateMessageId {
2     text    : string;
3     user_id: number;
4 } constraining {
5     present(text);
6     present(user_id);
7 }
```

Listing 3.9: Stricter version of the PrivateMessage interface

interface: `PrivateMessageId` is similar to `PrivateMessage` but requires the property `user_id` to identify the receiver of the private message. TIPC allows an assignment of a `PrivateMessageId` object to variables of type `PrivateMessage`, because every `PrivateMessageId` object will also be a valid `PrivateMessage` object. However, TIPC rejects assignments in the other way: not every valid `PrivateMessage` object will use the `user_id` to identify the receiver.

Interface instances can also be assigned to anonymous object types. Given the advanced rules on the presence of interface properties, this must also happen with extra caution.

Requirement 5. Assigning Interface Instances to a Variable of an Object Literal Type

The type system ensures that all properties listed in the type of the object literal are present in the interface instance.

Listing 3.10 shows two assignments from interface instances to variables of object literal types (line 3 and 5). On line 3, a `PrivateMessage` interface instance is assigned to a variable of type {`text: string`}. TIPC accepts this assignment, as `text` is a required property in `PrivateMessage` and thus certainly present. On the other hand, TIPC rejects the assignment on line 5: the object type at the left-hand side of the assignment expects a `text` and a `user_id`, but it is not guaranteed that the `user_id` will be present in a `PrivateMessage` object.

3.5 Updating Object Properties

As with every object-oriented type system, the assignment of a new value to a property of an object should only be allowed when the value is of the "correct"

```
1 let pm: PrivateMessage = ...;
2
3 let o1: {text: string} = pm; // correct
4
5 let o2: {text: string, user_id: number} = pm;
6   // error: unknown whether user_id is present in pm
```

Listing 3.10: Assigning interface instances to variables of object literal types

type. Inter-property constraints add an extra complication: assigning to a property might invalidate an inter-property constraint.

Requirement 6. Property Update Requirement

The type system ensures that an updated property does not affect the constraints (partly) imposed on that property.

In general, a type system accepts an assignment expression when the type of the right-hand side is *assignable* to the type of the left-hand side. In the face of inter-property constraints, there are three cases to consider:

Updating a present or absent property. Given the property accessing rules of Section 3.4, assignment remains fairly straight-forward, even with the existence of inter-property constraints. As TIPC assigns its intended type to required properties, these can safely receive another value of its intended type without invalidating a constraint: the constraint was certainly present beforehand and remains present after the assignment. However, caution is required when defining the *assignment compatibility* relationship between two types: it is crucial for inter-property constraints that present properties cannot become absent during an assignment! Therefore, TIPC ensures that the value `undefined` cannot be assigned to types other than type `undefined`, as this would allow present properties to go absent. This is known as the *strict null-checks mode* in TypeScript, which is explained in more detail in Section 5.5. In the same vein, TIPC allows the value `undefined` only to be assigned to type `undefined` to ensure that absent properties do not become present. Given the strict null-checks rules, absent properties can only be updated with the `undefined` value which ensures they stay absent.

Listing 3.11 illustrates these rules with several examples. Line 2 updates the required property `msg.text`, which is thus guaranteed to be present: according to the property accessing rules, TIPC assigns the type `string` to this property

```
1  function setMsg(msg:PrivateMessage, text:string, user_id:number) {
2    msg.text = text;        // ok
3    msg.text = undefined;   // error: assigning undefined to
4                            //          present property
5    msg.user_id = user_id; // error: property with unknown
6                            //          presence status
7
8    if (msg.user_id !== undefined) {
9      msg.user_id     = user_id;   // ok
10     msg.screen_name = undefined; // ok
11   }
12   //...
13 }
```

Listing 3.11: Updating properties

access. Therefore, we can safely assign a new string to this property. As already explained, TIPC rejects assignments of `undefined` to types other than `undefined`, such as on line 3. The update of the `user_id` property on line 5 will also fail: TIPC disallows the property access, as explained in the previous section.

The `if` statement on line 8 verifies the presence of `msg.user_id`. As a consequence, the ID is known to be present inside the true branch, and can be safely updated with a new number (line 9). Moreover, the screen name will certainly be absent inside the true branch: the type system may only allow the assignment of `undefined` to `msg.screen_name` (line 10).

Adding an absent property or removing a present property Removing a present property or adding a previously absent property needs to be done with care in TIPC. Only truly optional properties that are not part of any constraint, can safely change their presence status. However, when a property is part of an inter-property constraint, changing the presence status of a property could affect other properties. This is discussed in the following paragraph.

Updating a property that is part of an inter-property constraints Updating an inter-property constraint often requires the modification of several properties at once, as the object could be in a type-incorrect state in-between several assignments. Let us consider the case in Listing 3.12 where a programmer wants to switch from identifying the receiver by user ID to screen name. This code snippet first verifies whether the receiver is identified using its ID. If that is the case, the code snippet first removes the user ID (by assigning the value `undefined`, making it an absent property) and adds a screen name instead. TIPC rejects

```
1 let msg: PrivateMessage = {text: "Hello", user_id: 42};
2 if (msg.user_id !== undefined) {
3   msg.user_id = undefined;
4   msg.screen_name = "Alice";
5 }
```

Listing 3.12: Changing an inter-property constraint is not possible with separate assignments

this program: both assignments break the rules imposed by the strict-null checking mode. This behaviour is desirable: in-between lines 3 and 4, the exclusivity inter-property constraint on `msg` is violated: it contains neither a user ID nor a screen name. The next section elaborates on simultaneous updating of properties to solve this problem.

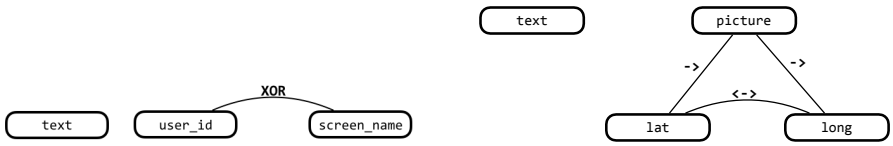
3.5.1 Updating Multiple Properties Simultaneously

Updating a property part of an inter-property constraint requires a language construct that updates multiple properties simultaneously. This ensures that the object is never in an invalid intermediate state between consecutive assignment statements. To enable this, TIPC has a function `assign`(i, o) that returns a *copy* of object i, in which the properties from the object o are added or updated. `assign` resembles the `Object.assign` function in JavaScript, but does not modify its input object: instead of modifying its first arguments, it returns a new object.

Listing 3.13 shows two examples of a multi-update, using `assign`. Line 3-4 shows an `assign` call that switches from user ID to screen name to identify the receiver of a private message. While programmers can update any subset of the properties of an object, not all combinations are correct. The second call to `assign` (line 6-7) shows an example of an invalid multi-update: only the user ID is updated (becoming absent). This kind of multi-update is rejected by TIPC, as it would invalidate the exclusivity constraint of `PrivateMessage`.

```
1 let msg:  PrivateMessage = {text: "Hello", user_id: 42};
2
3 let msg2: PrivateMessage =                // correct
4       assign(msg, {user_id: undefined, screen_name: "Alice"});
5
6 let msg3: PrivateMessage =
7       assign(msg, {user_id: undefined}); // incorrect
```

Listing 3.13: Using multi-assign to switch from user ID to screen name



(a) `PrivateMessage` (b) `Tweet`

Figure 3.3: Visualisation of constraints

Requirement 7. Simultaneous Update of Properties

The type system provides a means to safely update properties that are part of an inter-property constraint. A multi-update call is only accepted by the type system when *all relevant* properties are part of the update.

Intuitively, if an inter-property constraint exists between two or more properties, they have to all appear together in the call to `assign`. The properties of an object can thus be divided into one or more "clusters" that need to be updated together.

When constraints are visualised such as in Figure 3.3, clusters are easy to spot: all properties that are linked together with one or more constraint form a cluster.

For example, there are two clusters in the `PrivateMessage` interface (visualised in Figure 3.3a): the `text` property can be updated by itself, and the two properties of the exclusivity constraint also form a cluster. The `Tweet` interface in Listing 3.3 (constraints are visualised in Figure 3.3b) also has two clusters: there is a trivial cluster for `text`, and a separate cluster for the `long`, `lat` and `picture` properties.

3.6 Interface Inheritance

As other programming languages, TIPC also supports interface inheritance. When an interface extends another interface, all properties and all constraints of the superinterface are inherited. As a consequence, the constraints of the interface `I` and all its superinterfaces need to be satisfied for an object to be valid instance of `I`.

Requirement 8. Interface Inheritance

Interface definitions with inter-property constraints must be interoperable with interface inheritance.

Let us consider the example where we want a stricter version of the interface `PrivateMessage` in which only the screen name is allowed. Instead of creating a new interface, we can extend the existing interface with extra constraints. Listing 3.14 shows an interface in which all properties and constraints of `PrivateMessage` are inherited, with an additional `present(screen_name)` constraint. An object of type `PrivateMessageStrict` needs to satisfy the constraints of `PrivateMessageStrict` as well as the constraints of `PrivateMessage`. As the `xor` constraint from `PrivateMessage` is still applicable, this interface implicitly forbids the presence of a `user_id` property.

```
1 interface PrivateMessageStrict extends PrivateMessage {
2     // reuse properties from PrivateMessage
3 } constraining {
4     present(screen_name);
5 }
```

Listing 3.14: Extending PrivateMessage to require the screen name property

Listing 3.15 shows a code snippet that uses the `PrivateMessageStrict` interface. On lines 2 and 3 in Listing 3.15, the user properties of a `PrivateMessageStrict` object are accessed. Because of the constraint listed in the interface definition of `PrivateMessageStrict`, the type system allows the access of `screen_name` (line 2). Moreover, the type system also allows the access of `user_id` (line 3): by combining the `present(screen_name)` constraint (from `PrivateMessageStrict`) with the exclusivity constraint on both user properties (from the superinterface `PrivateMessage`), it is certain that `user_id` will be absent. To reflect this, the type system assigns the `undefined` type to `msg.user_id`.

```
1 function getInfoPMS(msg: PrivateMessageStrict) {
2   msg.screen_name; // :: string
3   msg.user_id;     // :: undefined
4   //...
5 }
```

Listing 3.15: Accessing properties of a `PrivateMessageStrict` instance

With the inheritance of interfaces, developers again have to ensure that the set of constraints of all interfaces and superinterfaces is satisfiable. In the case of `PrivateMessageStrict` there is exactly one combination of present and absent

properties that satisfy all constraints: `text` and `screen_name` are present, and `user_id` is absent.

3.7 Conclusion

In this chapter we described how the typing rules for language expressions need to be adapted in order to take inter-property constraints into account. We assumed a simple interface definition that separates presence constraints from type information. Constraints between different properties can be combined using the traditional logical connectives from propositional logic. We investigated how this extended interface type affected object creation, property access, object assignment, and property update. This has resulted in the following requirements for programming languages that want to support inter-property constraints. The requirements form the basis of the typing rules of TIPC.

Requirement 1. Instantiating Interfaces with Object Literals

The type system allows the instantiation of an interface with an object literal only when that object literal satisfies the interface constraints.

Requirement 2. Property Access

The type system assigns a type to a property access expression that correctly reflects the value of that property. When it is impossible to predict the exact type of the property, the type system rejects the property access.

Requirement 3. Presence Test

The type system extracts run-time type information from `if` statements and takes it into account when verifying property accesses.

Requirement 4. Assigning Instances of Interfaces to Variables of Interface Types

The type system ensures that no constraints are violated when an interface instance is assigned to a variable of an interface type.

Requirement 5. Assigning Interface Instances to a Variable of an Object Literal Type

The type system ensures that all properties listed in the type of the object literal are present in the interface instance.

Requirement 6. Property Update Requirement

The type system ensures that an updated property does not affect the constraints (partly) imposed on that property.

Requirement 7. Simultaneous Update of Properties

The type system provides a means to safely update properties that are part of an inter-property constraint. A multi-update call is only accepted by the type system when *all relevant* properties are part of the update.

In the next chapter we will elaborate on how the type system of TIPC meets these requirements. As the presence constraints on properties are linked using connectives from propositional logic, the type system will also use concepts from propositional logic to verify the type correctness of expressions.

Chapter 4

Statically Checking Inter-property Constraints

As shown in Chapter 2, inter-property constraints are commonly found in the documentation of web APIs. Moreover, bugs that stem from unsatisfied inter-property constraints are hard to catch. In the previous chapter, we have outlined how constraints between properties can be incorporated into TypeScript, giving rise to TIPC: a novel imperative object-oriented programming language. This language design translated into a set of requirements that specify how TypeScript's interface definitions are modified, and how interface instances need to be type checked. Together, these requirements statically ensure that inter-property constraints remain satisfied throughout the program.

In this chapter, we explain how the type system of TIPC fulfils the requirements listed in Chapter 3. Because the constraint language expresses constraints with logical connectives, the type system uses several concepts from propositional logic to guarantee correctness.[1]

The contents of this chapter are published in Oostvogels et al. [2018b].

[1]The constraint-centric interfaces introduced in this chapter should not be confused with constraint-based programming [Rossi et al., 2006]. Constraint-based programming is a discipline that finds solutions for a number of variables given constraints over these variables. By contrast, TIPC uses constraints and flow information to determine the most specific presence information for properties of objects.

4.1 Object Literals Have To Satisfy Constraints

In the previous chapter, we have seen that interfaces can be initialised with object literals or with interface instances. For the former, we have stated the following requirement:

Requirement 1. Instantiating Interfaces with Object Literals

The type system allows the instantiation of an interface with an object literal only when that object literal satisfies the interface constraints.

In this section, we discuss how TIPC meets this requirement when an interface is instantiated with an object literal. Using terminology from propositional logic, the type system uses the concept of a *valuation* to meet this requirement. A valuation in propositional logic is a mapping from proposition letters to truth values. For every valuation v there exists a unique function extension $\hat{v}$ which takes an entire proposition formula and returns true or false [Gallier, 2015]. In the context of inter-property constraints, proposition letters correspond to interface property names, and proposition formulas correspond to a logical conjunction of interface constraints. The type system requires that the properties in an object literal form a *valuation* that satisfies the presence constraints of the interface.

Solution for Requirement 1: Valuation

Given an object literal, the valuation v assigns `true` to a property `n` of the interface if and only if that property is present and not `undefined` in the literal. The domain of the valuation is the set of properties of the interface and its superinterfaces. All properties that are absent from the object literal or have the value `undefined` are `false`. To test whether an object literal satisfies the constraints, this valuation is applied to the function $\hat{v}$ formed by the interface constraints.

For example, the type system uses valuations for verifying that the following assignment is type safe:

```
1 let msg: PrivateMessage = {text: "Hello", user_id: 42};
```

The corresponding valuation will map the properties `text` and `user_id` from the right-hand side object literal onto `true`. There is only one property that is part of the property list of `PrivateMessage` but not part of the object literal: `screen_name`. This property is thus set to `false`.

$$v = \{text \mapsto true, user\_id \mapsto true, screen\_name \mapsto false\}$$

Given this valuation, the type system can use the valuation function $\hat{v}$ to verify whether this valuation satisfies the constraints of `PrivateMessage`, by applying this function to a propositional formula which is a conjunction of the `PrivateMessage` constraints:

$$\hat{v}(text \wedge (user\_id \text{ XOR } screen\_name)) = true \wedge (true \text{ XOR } false) = true$$

In the following example, an object literal with all three properties is assigned to a variable of the `PrivateMessage` interface:

```
1 let msg: PrivateMessage = {text: "Hello",
2                            user_id: 42,
3                            screen_name: "Alice"};
```

This results in a valuation in which all three properties are mapped onto true:

$$v = \{text \mapsto true, user\_id \mapsto true, screen\_name \mapsto true\}$$

When this valuation is applied to the `PrivateMessage` constraints, this results in `false`:

$$\hat{v}(text \wedge (user\_id \text{ XOR } screen\_name)) = true \wedge (true \text{ XOR } true) = false$$

If the valuation function fails, the type system rejects the assignment. In that case, the resulting error message can indicate which constraint was not satisfied. A more *human-readable* error message could clarify the constraint for developers who did not write the interface definitions themselves. For example, an error message for the failing assignment could look as follows:

```
1 ERROR: The assignment of an object with type {text: string,
2 user_id: number, screen_name: string} to a variable with type
3 PrivateMessage failed: the constraint "user_id XOR screen_name"
4 was not satisfied.
5 Please provide exactly one of: user_id and screen_name.
```

When the type system constructs the valuation for an initialisation check, it needs to have an exact representation of which properties are present and which are not. As we explain in Section 5.4, TypeScript supports width subtyping. As a consequence, the type system cannot guarantee that an object literal type contains *only* the properties listed in that type.

This leads to the following restriction:

Restriction: Object Literals

When an interface instance is initialised with an object literal, the right-hand side of that assignment needs to be a fresh object literal instead of any expression of an object literal type.

By only allowing fresh object literals (instead of also allowing object literals created earlier in the program), the type system has an exact view of the properties that are present and can thus guarantee that the interface constraints are satisfied. An alternative to this restriction could be to disallow width subtyping altogether. In that case, any variable of an object literal type could be used to initialise an interface instance. Both approaches have advantages and disadvantages. We have chosen for the object literal restriction, to minimise the impact of interfaces with inter-property constraints in existing TypeScript programs.

Appendix A shows the details of a small study on web APIs. This study indicates that the object literal restriction is not a severe restriction. The study explored a list of GitHub projects that use an SDK to send requests to the Twitter and YouTube API. In 163 of the 180 studied API calls, the data was provided as an object literal. In 14 out of the 17 cases where the data argument was not an object literal, the object was defined directly above the API call.

In the future, we plan to draw inspiration from other research to weaken this restriction. For example, Heidegger and Thiemann [2010] propose a *recency type system* to support the *initialisation phase* of object literals. Their type system allows the type to change during the construction of the object. Afterwards, the type system assigns a *summary type* to the object literal. Incorporating a form of recency types into TIPC would enable us to allow recently created object literals to be assigned to interfaces variables. However, further research is needed to investigate the full impact of recency types (more specifically its flow-sensitivity, heap types and strong updates) in the TIPC type system.

There are two ways the initialise an interface with an object literal: via assignment or casting. To limit the complexity in the formalisations of TIPC, interface instances can only be created using type casts. This does not limit the expressivity of TIPC: object literals can still be assigned to interface variables as long as they are first casted to the interface. The following code snippet shows an example:

```
1 let msg: PrivateMessage = <PrivateMessage>{text: "Hello",
2                                            user_id: 42};
```

Note that the examples in this dissertation omit this type cast for brevity.

4.2 Constraints Dictate Property Presence

As with other type systems, interface declarations contain a list of properties with their types. However, accessing a property of an interface may only be allowed when that property is present or absent. This information can be found in constraints of the interface.

Requirement 2. Property Access

The type system assigns a type to a property access expression that correctly reflects the value of that property. When it is impossible to predict the exact type of the property, the type system rejects the property access.

Of course, with complex inter-property constraints, these constraints may not be *directly* present in the constraint set. For example, the following interface definition is a variant of the `PrivateMessage` interface with an extra constraint indicating the absence of `screen_name`. In this case, `user_id` will always be present in an object with this type, but the interface does not have an explicit constraint indicating the presence of `user_id`.

```
1 interface PrivateMessageExplId {
2     text        : string;
3     user_id     : number;
4     screen_name: string;
5 } constraining {
6     present(text);
7     present(user_id) xor present(screen_name);
8     ¬present(screen_name);
9 }
```

To know whether a property is present or absent given a set of constraints, the type system uses the concept of *logical entailment* from propositional logic. A logical entailment (denoted $\vDash_\ell$) verifies whether a constraint logically follows from a set of constraints.

Accessing a property of an interface instance may only be allowed when that property is certain to be present or absent. In other words, when accessing a property, the presence (or absence) of a property has to follow from the interface constraints.

Solution for Requirement 2: Logical Entailment

- The type system assigns the intended type to a property, if that property is certainly present, i.e.:
  interface constraints $\vDash_\ell$ `present(property)`
- The type system assigns `undefined` to that property, if that property is certainly absent, i.e.:
  interface constraints $\vDash_\ell \neg$`present(property)`
- The type system rejects the property access, if it is not certain whether the property is present or absent.

Calculating logical entailments can be efficiently automated using deductive systems such as the Gentzen system [Gallier, 2015].

Returning to the `PrivateMessage` example, the type system verifies the following logical entailment for accessing the `text` property. The constraints of `PrivateMessage` are on the left of the logical entailment, and the constraint indicating the presence of `text` is on the right. This logical entailment is obviously true, as the presence of `text` is also part of the interface constraints.

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\end{array}\right\} \vDash_\ell \texttt{present(text)}$$

Similarly, the absence of a property can also follow from a set of constraints. In that case, the type system can assign the `undefined` type to that property to indicate its absence. This is not very useful in a normal interface with inter-property constraints: when a property always has to be absent, it can also be omitted from the property list. However, proving the absence of a property will prove to be useful when taking runtime type information into account (Section 4.3) and when comparing interface structures (Section 4.4).

In the case where neither the presence nor absence of a property can be derived from the constraints, the type system should conservatively reject the access of that property. This also follows from the logical entailment. For example, the type checker rejects the function `getInfoPM` of Listing 3.6, because neither the presence nor the absence of `user_id` is a logical consequence of the interface constraints:

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\end{array}\right\} \nvDash_\ell \texttt{present(user\_id)}$$

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\end{array}\right\} \not\models_\ell \neg\texttt{present(user\_id)}$$

In this case, the type system throws an error message which should indicate that the property access failed. A possible solution is to encourage the developer to first verify the presence of the property via an if test. This could look as follows:

```
1 ERROR: the property access of user_id failed: neither the
2 presence nor the absence can be guaranteed by the PrivateMessage
3 interface constraints.
4 Consider verifying the presence of user_id via an if test before
5 accessing it.
```

Note on Union Types: At first, one might think that assigning the union type `number | undefined` to `user_id` is a better solution than rejecting the property access, as `user_id` is either `number` (when present) or `undefined` (when absent). However, this would lead to type-unsafe programs, as the assignment of any number or `undefined` to `user_id` would be accepted by the type system. This could change the presence status of `user_id` without guaranteeing that inter-property constraint on both user properties remains satisfied, and should thus be disallowed by the type system.

Note on Gradual Types: TIPC rejects accessing properties that are not certainly present or absent. Another approach can be found in the research on gradual type systems, which insert run-time checks between typed and untyped code. Instead of rejecting accessing a property that is not certainly present or absent, a gradual type system inserts a run-time check before the property access to ensure type safety. This is discussed in detail in Section 10.3. TIPC follows the TypeScript philosophy of leaving no type trace in the compiled JavaScript code, and thus requires that developers write these run-time presence checks themselves.

4.3 Explicit Property Presence Tests

It is common in dynamic languages to perform tests to verify the presence of properties at runtime, before accessing a property. However, even in a statically typed programming language, these runtime property presence tests can provide the type system with more information about the object being tested.

Requirement 3. Presence Test

The type system extracts run-time type information from `if` statements and takes it into account when verifying property accesses.

When an `if` statement verifies the presence of a property, then it is certain that in one branch the property is present, while it is guaranteed to be absent in the other. Moreover, the extra knowledge on the presence of one property can also add certainty on the presence or absence of other properties. This idea is also known as *occurrence typing* [Tobin-Hochstadt and Felleisen, 2008, 2010].

Solution for Requirement 3: Extra Constraints on the Premises of the Logical Entailment

Inside the true and false branch of an `if` statement, the premises on the left-hand side of a logical entailment are extended with the information from the `if` statement whenever a property of the object being tested is accessed.

In Listing 3.7 (page 40) there is an `if` statement that verifies the presence of `user_id` in a `PrivateMessage` object. For every property access of that object inside the true branch of that `if` statement, the type system has to add the extra information on the presence of the user ID (`present(user_id)`) to the premises of the logical entailment. With the extra constraint on the left-hand side, the logical entailment will now succeed to prove the presence of `user_id`. This way, the type system can safely assign the intended type (`number`) to `user_id`, and thus allow the access of `user_id` inside the true branch.

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\\ \texttt{present(user\_id)}\end{array}\right\} \vDash_{\ell} \texttt{present(user\_id)}$$

Similarly, the absence of the user ID in the false branch is added to the premises of logical entailments for property accesses of that `PrivateMessage` instance.

Given this extended set of constraints in the premise of the logical entailment, the type system can now also assign the type `undefined` to accesses of `screen_name` in the true branch: the absence of `screen_name` follows from combining the extra constraint with the exclusivity constraint of `PrivateMessage`.

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\\ \texttt{present(user\_id)}\end{array}\right\} \models_\ell \neg\texttt{present(screen\_name)}$$

Likewise, the presence of `screen_name` will follow from the premises in the false branch.

In Listing 3.8 (page 41), the presence check on longitude of a `Picture` instance guarantees that the longitude is present, but also suffices to safely access latitude (by combining the constraint `present(long)` $\leftrightarrow$ `present(lat)` with `present(long)`) and the picture itself (combining constraints `present(long)` $\rightarrow$ `present(picture)` and `present(long)`).

4.4 Interface-Interface Compatibility

When an expression of a certain interface type is assigned to a variable of another interface type, the type system has to ensure that the assignment is type safe. Next to assignment compatibility for literal types, programming languages with regular interfaces only need to verify whether all required properties of the left-hand side of the assignment are certainly present in the right-hand side interface. In TIPC, on the other hand, there can be very complex constraints on the presence of properties, and between the presence of properties.

Requirement 4. Assigning Instances of Interfaces to Variables of Interface Types

The type system ensures that no constraints are violated when an interface instance is assigned to a variable of an interface type.

Normally, an instance of interface `Source` is considered assignable to a variable of another interface type `Target` if `Source` contains at least every property and method in `Target`. However, with the addition of constraints we must also take care that there cannot exist a valid instance of `Source` that violates the constraints in `Target`.

Solution for Requirement 4: Logical Entailment

To verify the assignment compatibility of interfaces on the level of constraints, TIPC uses logical entailment checking. More specifically, the constraints of the target of the assignment need to follow from the constraints of the source.

However, this does not suffice. The type system also needs to take the structural differences between the property lists of the two interfaces into account. This is achieved by adding extra absence constraints to both sides of the logical entailment, based on the differences between the properties of both interfaces.

$$\left\{\begin{array}{l} \textit{source constraints} \\ \textit{structural differences} \end{array}\right\} \vDash_{\ell} \left\{\begin{array}{l} \textit{target constraints} \\ \textit{structural differences} \end{array}\right\}$$

Section 4.4.1 shows an example of the assignment compatibility between two interfaces with identical property lists. The second and third section explain the need for taking the differences in property lists into account in the left-hand side (premises, Section 4.4.2) and right-hand side (consequent, Section 4.4.3) of the logical entailment.

4.4.1 Target Constraints Follow From Source Constraints

To guarantee that all constraints of `Target` are satisfied, every constraint from `Target` must be a *logical entailment* of the constraints in `Source`. When the property lists of the source and target interfaces are identical in an assignment, no extra constraints need to be generated.

The following code snippet shows an example: an instance of the interface `PrivateMessage` is assigned to a variable of type `PrivateMessageAll`. This interface is defined in Listing 4.1 and has three properties, which are all required: `text`, `user_id` and `screen_name`.

```
1 let msg1: PrivateMessage    = { text: "Hello", user_id: 42 };
2 let msg2: PrivateMessageAll = msg1;
```

The assignment on line 2 results in the following logical entailment, which is invalid: the presence of `user_id` *as well as* the presence of `screen_name` do not follow from `PrivateMessage` constraints, which require *either* the presence of `user_id` or `screen_name`.

```
interface PrivateMessageAll{
    text        : string;
    user_id     : number;
    screen_name: string;
} constraining {
    present(text);
    present(user_id);
    present(screen_name);
}
```

Listing 4.1: Variant of the `PrivateMessage` interface

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id) xor present(screen\_name)}\end{array}\right\} \not\models_\ell \begin{array}{l}\texttt{present(text)} \wedge\\ \texttt{present(user\_id)} \wedge\\ \texttt{present(screen\_name)}\end{array}$$

As the logical entailment fails, the type system will reject the assignment. The following code snippets shows an example of what the corresponding error message could look like.

```
ERROR: the assignment of a PrivateMessageAll expression to a
 PrivateMessage expression is invalid.
The presence of user_id and the presence of screen_name do
 not follow from the PrivateMessage interface constraints.
```

4.4.2 Structural Differences: Premises

When an interface instance is assigned to an expression with an interface type, the constraints of the target interface have to follow from the constraints of the source interface. When the `Target` interface has properties that are not part of the property list in `Source`, these properties are certainly *absent* in the `Source`. It is useful to add this knowledge to the premises of the logical entailment, as it allows more conclusions to be deduced from the premises.

The previous chapter showed an example of when this is useful. It introduced a variant of the `PrivateMessage` interface called `PrivateMessageStrict` (defined in Listing 3.9), in which only the `user_id` may be used to identify the receiver. Assigning a variable of the stricter interface type `PrivateMessageStrict` to a variable of type `PrivateMessage`, gives rise to the following logical entailment. The premises contain the constraints of the source of the assignment (`PrivateMessage`) and the consequent of the logical entailment contains the constraints of the target (`PrivateMessage`).

Next to the constraints of `PrivateMessage`, the premises of the logical entailment contain an extra constraint. This constraint denotes the absence of the screen name in the `PrivateMessageStrict` interface. Without the third constraint, the logical entailment would not be valid.

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id)}\\ \neg\texttt{present(screen\_name)}\end{array}\right\} \models_\ell \begin{array}{l}\texttt{present(text)} \wedge\\ \quad \texttt{present(user\_id) xor present(screen\_name)}\end{array}$$

4.4.3 Structural Differences: Consequent

In Section 4.1, we have discussed the need for restricting the initialisation of interfaces to object literals. This restriction is necessary to prevent the existence of hidden properties in objects. The same kind of restriction is necessary when interface instances are assigned to each other: the type system has to ensure that —at runtime— there are no properties in the object that are not a part of the property list of the target interface.

To ensure that there are no hidden properties, the consequent of the logical entailment for an assignment needs to be extended. For every property that is listed in the source interface but that is not part of the target interface, the absence of that property should follow from the source constraints.

Note that this kind of restriction is more generous than requiring the property lists to be identical. Especially with objects inside of `if` statements that verify the presence of properties, it is likely that an interface lists a property that inside the `if` statement is known to be absent.

Restriction: Interfaces Do Not Allow Width Subtyping

As a consequence of this compatibility strategy, TIPC does not support width subtyping for its interfaces. Evidently, width subtyping is irreconcilable with a type system that requires the absence of properties. Therefore, the type system has to (counter-intuitively) require that the source interface only contains properties other than those in the target interface when those properties are guaranteed to be absent.

The following code snippets shows an example of the need for disallowing interface width subtyping. It contains three interfaces: the previously defined interface `PrivateMessage`, a new interface `PrivateMessageId` (see Listing 3.9 on page 42, a variant of the private message interface where the receiver has to be identified using the user ID) and the interface `PrivateMessageAll` (see Listing 4.1

on page 61, a variant where the receiver is identified with both the user ID and the screen name).

```
1 let msg1: PrivateMessageAll = { text: "Hello", user_id: 42,
2                                 screen_name: "Alice"};
3 let msg2: PrivateMessageId  = msg1;
4 let msg3: PrivateMessage    = msg2;
```

The first line contains the initialisation of a `PrivateMessageAll` object. Next, it is assigned to a variable of type `PrivateMessage`. On the last line of the code snippet, the `PrivateMessageId` instance is assigned to a variable of the `PrivateMessage` interface type. In order to guarantee type safety in TIPC, this code snippets needs to be rejected: the object `msg3` has type `PrivateMessage`, but contains both `user_id` and `screen_name`, violating its constraints. The type system of TIPC prevents this scenario by disallowing width subtyping on interfaces: this way, the type system considers the second assignment (line 3) invalid, ensuring the inter-property constraints remain satisfied.

The following logical entailment is performed by the type system when verifying the second assignment (line 3). The third constraints in the consequent of the logical entailment is generated because of the structural differences between `PrivateMessageAll` and `PrivateMessageId`, and is the cause for the invalid logical entailment, and thus the cause for rejection of that assignment by the type system.

$$\left\{\begin{array}{l}\texttt{present(text)}\\ \texttt{present(user\_id)}\\ \texttt{present(screen\_name)}\end{array}\right\} \not\models_\ell \begin{array}{l}\texttt{present(text)} \wedge\\ \texttt{present(user\_id)} \wedge\\ \neg\texttt{present(screen\_name)}\end{array}$$

As this logical entailment fails due to *generated* constraints (unbeknownst to developers), it is important that the corresponding error message clearly indicates why the assignment fails. An example of such an error message could be:

```
1 ERROR: The assignment of a PrivateMessageAll expression to a
2 PrivateMessageId expression is invalid.
3 Because screen_name is not part of the PrivateMessageId
4 interface, its absence needs to follow from the
5 PrivateMessageAll constraints.
```

4.5 Interface-Object Compatibility

The previous chapter listed the following requirement for the type system for assigning an expression of an interface type to a variable of a regular object type.

Requirement 5. Assigning Interface Instances to a Variable of an Object Literal Type

The type system ensures that all properties listed in the type of the object literal are present in the interface instance.

In TIPC, all properties in an object literal type are required properties. As a consequence, the properties of the object literal type all have to be present in the interface type as well.

Solution for Requirement 5: Logical Entailment

For every property `p` in the object literal type (target), the following logical entailment needs to be true:

$$\{\textit{interface (source) constraints}\} \vDash_\ell \texttt{present(p)}$$

In the example in Listing 3.10 of Chapter 3 (page 43), an expression of interface type `PrivateMessage` is assigned to a variable of the object literal type `{text: string, user_id: number}`. This assignment is rejected by the type system, as the corresponding logical entailment is not valid:

$$\left\{\begin{array}{l}\texttt{present(text);}\\ \texttt{present(user\_id) xor present(screen\_name);}\end{array}\right\} \nvDash_\ell \begin{array}{l}\texttt{present(text)} \wedge\\ \texttt{present(user\_id)}\end{array}$$

The corresponding error message has to clearly explain why this assignment failed:

```
1 The assignment of a PrivateMessage expression to an expression
2  with type {text: string, user_id: number} is invalid.
3 The interface constraints of PrivateMessage have to guarantee
4  the presence of the properties text and user_id.
```

Note that for object literal types, width subtyping is still allowed in TIPC. For example, the type system will accept an assignment where an instance of `PrivateMessage` is assigned to a variable of the object literal type `{text: string}`, as the corresponding logical entailment is valid.

$$\left\{\begin{array}{l}\texttt{present(text);}\\ \texttt{present(user\_id) xor present(screen\_name);}\end{array}\right\} \vDash_\ell \texttt{present(text)}$$

4.6 Updated Objects Have To Satisfy Constraints

As TIPC is a programming language with complex presence constraints on interface properties, the type system has to take constraints into account when updating properties:

Requirement 6. Property Update Requirement

The type system ensures that an updated property does not affect the constraints (partly) imposed on that property.

Given the property accessing rules imposed by the type system, updating a single property of an object is quite straightforward.

Solution for Requirement 6: Assignment Compatibility

The updating of one property does not need any constructs of propositional logic. The type of the right-hand side expression of the assignment simply needs to be assignable to the type of the left-hand side. As already explained in Section 3.5, the strict null-checks are crucial to ensure present properties do not become absent (and the other way around).

In the previous chapter, we have also introduced a new language construct in TIPC: `assign`, which allows updating multiple properties at once. It expects two objects as parameters and returns a copy of the first parameter in which properties of the second argument are updated or added. This is especially useful in the context of inter-property constraints, where switching between one valid combination of properties to another using single-property updates results in intermediate invalid objects.

Requirement 7. Simultaneous Update of Properties

The type system provides a means to safely update properties that are part of an inter-property constraint. A multi-update call is only accepted by the type system when *all relevant* properties are part of the update.

To verify that all constraints are still satisfied after a simultaneous update of multiple properties, the type system again uses valuations. Valuations were already used to verify the initialisation of an object (Section 4.1). However, the

```
interface PrivateMessage1 {
    text          : string;
    r_user_id     : number;
    r_screen_name: string;
    s_user_id     : number
    s_screen_name: string;
} constraining {
    present(text);
    present(r_user_id) xor present(r_screen_name);
    present(s_user_id) xor present(s_screen_name);
}
```

Listing 4.2: Variant of the `PrivateMessage` interface with sender and receiver

use of valuations for `assign` calls is slightly different:

Solution for Requirement 7: Valuations

The multi-update in TIPC only affects a subset of the properties of an object. Therefore, the second argument of `assign` must only serve as a valid valuation of a *subset* of properties and constraints of the interface.

This subset is the smallest possible subset of properties and closed constraints that contains all properties that are part of the update and all constraints involving those properties. This transitive closure can be calculated as follows: start with a subinterface which contains only the properties being updated. Next, repeatedly add all constraints which contain any of the properties in the subinterface, and any properties mentioned by these constraints, until no more properties or constraints can be added.

Evidently, the types of properties in the object literal must conform to those defined in the interface (with the exception of undefined properties). Note that an update is only valid when all properties of the relevant subset (henceforth called cluster) are updated.

Consider the variant of the `PrivateMessage` interface defined in Listing 4.2 which indicates both the sender (with either `s_user_id` or `s_screen_name`) and the receiver (either `r_user_id` or `r_screen_name`) of a private message. This interface contains three constraints: text is a required property, and there is an exclusivity constraint for both the sender and receiver properties.

Logically, these properties form separate clusters that do not affect each other:

1. `text` can be updated separately;
2. `s_user_id` and `s_screen_name` have to be updated together;
3. `r_user_id` and `r_screen_name` have to be updated together.

The following code snippet creates private message with a sender and a receiver. Afterwards, the receiver of the private message is updated using `assign` on line 3. The second invocation of `assign` is an invalid one: only a part of the cluster is being updated.

```
1 let msg1: PrivateMessage1 = { text: "Hello",
2                               r_user_id: 42, s_user_id: 43};
3 let msg2    = assign(msg, { r_user_id: undefined,
4                               r_screen_name: "Alice"}); //OK
5 let msg3    = assign(msg, { r_screen_name: "Alice"}); //ERROR
```

The first `assign` call in this code snippet only updates the receiver of the private message. Therefore, the constraints for the sender side do not have to be taken into account. The `assign` operation type checks if the object literal (the second argument of `assign` is a valid valuation of the constraint on line 9. This is the case, as `undefined` is interpreted as an absent property.

The second `assign` call is rejected by the type system: it only updates the screen name property of the private message receiver, which is only a part of the cluster. As the clusters are not explicitly listed in the interface definition, it is important that the error message clearly indicates why this call is invalid:

```
1 ERROR: The call to assign is invalid.
2 The second argument of the call contains the property
3  r_screen_name.
4 In that case, the following properties also have to be
5  present in the object literal: r_user_id.
```

4.7 Conclusion

In this chapter, we have explained how the type system of TIPC ensures type safety in programs where interfaces have complex presence constraints on their properties. The type system uses several concepts from propositional logic to ensure that the objects are created and updated in a type-safe way, i.e. while the constraints remain satisfied. Valuations are used to verify when an interface is initialised with an object literal, as well as when multiple properties of an interface

instance are updates simultaneously. Logical entailment is used in various ways to verify safe property accesses and the assignment of interfaces to others.

In the next chapter, we introduce TypeScript, the programming language which forms the basis of TIPC, and we discuss several features typical to TypeScript. Afterwards, we present the formalisations of TIPC. The solutions that were presented in this chapter are incorporated into the typing rules of TIPC's type system. Together with the operational semantics, Chapter 6 will prove that the type system is sound, even in the presence of inter-property constraints.

Chapter 5

TypeScript's Idiosyncrasies

The previous two chapters introduced TIPC, a statically typed programming language which supports constraints between properties of an object. As we have seen in Chapter 2, these kinds of constraints are commonly found in the documentation of (web) APIs. Incorporating support for inter-property constraints in the type system of TIPC alleviates the need for developers to check these constraints manually. Instead, the constraints are verified at compile-time.

TIPC is an extension of an existing programming language, to wit TypeScript. Before we introduce the formalisation of TIPC in Chapter 6, this chapter first presents the idiosyncratic features of TypeScript that are important with regards to inter-property constraints.

TypeScript is a superset of JavaScript that adds optional typing, classes and interfaces. JavaScript is a very dynamic programming language, originally intended as a small scripting language for adding interactivity to web pages. JavaScript is a very lenient programming language: for example, properties may be added to or removed from objects at any time, and the plus operator accepts nearly any combination of operands. As the size of JavaScript programs grows, the advantages of the dynamic features of JavaScript start to diminish. Large JavaScript programs are harder to develop and maintain as bugs are more difficult to find, especially because type errors only occur at runtime.

TypeScript has a type system to statically catch type errors, which results in safer programs. To achieve this, TypeScript extends JavaScript with type annotations, classes and inheritance. The compiler can also inform IDEs with type information for documentation. One of the main strengths of TypeScript are its *type definitions*: regular JavaScript libraries can be used in TypeScript programs by using type definitions that provide type information about JavaScript

libraries. Moreover, transforming existing JavaScript programs to TypeScript programs has a minimal impact: TypeScript compiles to regular JavaScript code (as does TIPC).

In the following sections, we explain the key features of TypeScript.

5.1 Optional Types

Types in TypeScript are entirely optional: the developer has no obligation to provide type annotations for its variables, parameters, functions, etc. If possible, TypeScript infers the type of a variable. For example, TypeScript infers that in the assignment `let user_id = 42;` the `user_id` is a number.

However, sometimes the type of a variable is unknown and cannot be inferred. In that case, TypeScript assigns the top type `any` to that variable. As the type system cannot infer any information from `any`, a variable of type `any` is equivalent to an untyped variable.

Contextual typing The type system of TypeScript uses *contextual typing* [Bierman et al., 2014] to improve type inference by using type information from the *inverse* direction. Informally, the context of a function definition informs the type checker on the type of the function arguments. Without contextual typing, these function arguments would receive type `any`.

The following code snippets shows an example. The function `getLength` on line 1 has type `any -> any`: TypeScript is unable to infer any information about the untyped parameter `x`. However, as soon as `getLength` is assigned to the variable `f` of type `string -> any`, the type system imposes the contextual type `string` on the untyped argument `x`. In turn, the expression `x.length` receives type `number` instead of `any`. This process is known as contextual typing.

```
1 function getLength(x) { return x.length; } //;any -> any
2
3 let f: string -> any;
4 f = function getLength(x) { return x.length; }; //;string -> number
```

Contextual typing is especially convenient for callback arguments. The following code snippet shows an example. The function `waitForResult` takes a callback argument and provides the type for that callback function. When `waitForResult` is called, its callback argument is contextually typed with the type from the variable declaration. The type system will reject the call to `waitForResult` because its contextual typing will infer that `x` will actually be a `string`.

```
1 let waitForResult: (cb: (x: string) => number) => void;
2 //...
3 waitForResult((x) => (x * x)); //error
```

Optional Typing versus Gradual typing TypeScript is *optionally* typed, which should not be confused with *gradual* type systems [Siek and Taha, 2006, 2007]. Both kinds of type systems have in common that they type check code that is partly typed and partly untyped. The difference between both kinds of type systems is in its soundness. Gradual type systems ensure soundness between the typed and untyped parts of the code by inserting run-time checks between the typed and untyped parts of the code. Optional type systems, on the other hand, do not insert type checks between the typed and untyped code. Contrary to gradual type systems, optional type systems will compile the code regardless of type errors.

Next to the optional type system, TypeScript also has extra unsound language features. These will be discussed in the next section.

5.2 Unsoundness

Typically, statically typed programming languages are *sound*, which guarantees that every program that is accepted by the type checker, will not result in type errors at runtime. TypeScript, on the other hand, takes a different approach. Contrary to most statically typed programming languages, TypeScript is deliberately *unsound*. Next to its optional type system, in which the untyped parts may be type unsafe, TypeScript also has type unsafe features in the typed parts of the program in order to support features that are found in typical JavaScript programs. In the rest of this section, we elaborate on the unsound language features in TypeScript.

Indexing Accessing a field of an object using the dot notation (for example `obj.foo = "bar"`) in JavaScript is syntactic sugar for indexing with the bracket notation (i.e. `obj["foo"] = "bar"`). When the index is a string literal, the type system of TypeScript is able to look up the type of that property. In the case when the string representing the index is not known at compile time, or when the string literal points to an unknown member, the type system cannot guarantee that the field access will be safe. However, in order to support as many JavaScript

programs as possible, TypeScript does not disallow these kinds of field accesses. Instead, it assigns the top type `any` to the field access expression.

On the other hand, TypeScript provides extra type safety regarding indices when the object has a type. Using `keyof`, an *index type query*, developers can retrieve a union type of all key strings of a certain object type. For example, `keyof PrivateMessage` (without advanced constraints) results in the type `"text" | "user_id" | "screen_name"`. TypeScript uses its *string literal types*: the only valid value for a string literal type is the string literal itself. The result of a `keyof` can later be used in the program to ensure that a valid index is used to access a property.

Downcasting The casting of a variable to a type can be divided into two categories: upcasting and downcasting. Upcasting happens when a variable is casted to a more general type (than the type of that variable). An upcast is always sound, as one moves *up* in the inheritance hierarchy. Downcasting, on the other hand, is when a type is casted to a more specific type. This cast can possibly be unsound as it is not guaranteed that the target type of the cast is of that type. Most of the object-oriented languages that allow downcasting (such as Java) ensure soundness by verifying whether the downcast is safe *at runtime* using runtime type information. TypeScript also allows downcasting, but does not generate runtime type tests to verify whether the cast is safe at runtime. However, TypeScript does take information from explicit runtime type tests (provided by the developer) into account while type checking the program. We elaborate on this in Section 5.6.

Covariance Type checking function parameters in TypeScript is *bivariant*, which is a combination of covariance and contra-variance. This means that the type of a function parameter should be assignable to the type of provided argument, or the other way around. While contra-variance is sound, covariance is not: Listing 5.1 shows an example. As TypeScript allows covariance in function parameter types, this program is accepted by the type checker (because `Dog` is assignable to `Animal`). The assignment inside the function `f` is also accepted by the type checker: `animals` is an array of animals, which thus may contain cats. But this results in an unsound program where the array of `dogs` contains a cat.

```
1 function f (animals: Animal[]) {
2    animals[0] = new Cat();
3  }
4  let dogs: Dog[] = ...;
5  f(dogs);
```

Listing 5.1: Bivariance in TypeScript

```
1 function foo() {
2
3    x = 5;
4    if (true) {
5       var x;
6    }
7    return x;
8 }
```

Listing 5.2: Function scoping in JavaScript

```
1 function foo() {
2    var x;
3    x = 5;
4    if (true) {
5
6    }
7    return x;
8 }
```

Listing 5.3: Function scoping (rewritten)

In the common case where the function argument is not mutated inside the function, the covariance does not pose a risk for unsoundness. Disallowing covariance would impose a huge restriction on which JavaScript programs are supported by TypeScript.[1]

5.3 Block Scoping

Variable declarations in JavaScript are different from most programming languages. For example, the code snippet in Listing 5.2 defines a valid JavaScript function. The function `foo` first assigns a number to the variable `x`. Only afterwards, the variable `x` is defined, in an `if` statement. At runtime, a call to `foo` will return the number 5. The type system of TypeScript supports this scoping behaviour.

In JavaScript, variables can be referred to *before* they are assigned. Even more, all variable declarations inside a function are hoisted to the beginning of the function declaration (which is called *function scoping*). The example in Listing 5.2 is treated as if it was written like Listing 5.3, where the variable definition inside the `if` statement is hoisted to the beginning of the body of `foo`.

Function scoping is quite permissive: every variable declared inside a function can be used throughout the entire function body. The leniency of function scoping can lead to bugs that are hard to catch. A more restrictive way of scoping is *block scoping*, where variables are only visible in the block in which they are declared. Most statically typed programming languages have block scoping, while scripting languages sometimes have more permissive variable declarations.

TypeScript supports both function scoping (with `var`) and block scoping (with

[1] However, developers can disallow the bivariance of function type parameters using the `strictFunctionTypes` flag.

`let` and `const`). The latter two kinds of variable declarations are translated to `var` declarations when compiled to JavaScript. Note that block-scoped variable declarations in TypeScript cannot be used before they are declared and cannot be redeclared in the same block.

5.4 Interfaces

Interfaces in TypeScript are used to describe the structure of an object: they contain fields (methods or properties) which can be required or optional (indicated using a question mark). TypeScript also supports inheritance, such that interfaces can inherit fields from other interfaces.

Excess properties Interfaces define which required and optional fields an object may describe. However, it is not certain that an object of an interface type *only* contains the fields defined in the interface. TypeScript only verifies that there are no excess properties when an *object literal* is assigned to a variable of a certain interface type. In all other cases (when the right-hand side of an assignment does not contain an object literal), TypeScript employs *width subtyping*: the object on the right-hand side has to contain *at least* all properties defined in the type of the left-hand side. This is shown in Listing 5.4. While line 5 (the assignment of an object literal to `XY`) results in a typing error, this is not the case when another variable is assigned to `XY` (line 7).

```
1 interface XY {
2     x: number
3     y: number
4 }
5 let xy1: XY = { x: 5, y: 6, z: 7 }; //ERROR
6 let xyz: { x: number, y: number, z: number } = { x: 5, y: 6, z: 7 };
7 let xy2: XY = xyz; //OK
```

Listing 5.4: Excess properties

Structural type system Type systems can be divided into nominal type systems and structural type systems, which differ in when types are considered equal. In nominal type systems, the equality of types is based on the name declarations of those types. As the name already suggests, structural type systems base the equality of types on the *structure* of a type.

TypeScript is structurally typed. This means that the equality of interfaces in TypeScript is based on their structure: interfaces are equal when they have the same properties and when the types of those properties are equal as well.

Listing 5.5 shows an example of two interfaces `X` and `Y`. In a structural type system, both interfaces are considered equal, as they both have one property (`a`) with the same type. Thus, assigning an `X` object to a variable of type `Y` will be accepted by the type system of TypeScript. In nominally-typed programming languages such as Java and C#, `X` and `Y` would not be equal because they originate from two different definitions.

```
1 interface X {
2     a: number
3 }
4 interface Y {
5     a: number
6 }
7 let x: X = {a: 5};
8 let y: Y = x; //OK in TypeScript
```

Listing 5.5: Structurally-typed interfaces in TypeScript

As interfaces simply map names onto an object type, and interface names are not taken into account when comparing types, the type system of TypeScript can translate interface types to object literal types without losing information. The object literal equivalent for the two interfaces in Listing 5.5 is {`a: number`}.

Callable objects In JavaScript, it is common to store additional data on function objects. In order to model this, TypeScript supports the definition of function types in interfaces. To define one, the interface may only contain one *bare* property, which is an anonymous function. Listing 5.6 shows an example: the interface contains one property which is an anonymous function that takes a number and returns a number. On line 4, a new variable of type `Double` is declared with as value an anonymous function. This variable can then be used as a procedure, such as on line 5.

```
1 interface Double {
2     (n: number): number;
3 }
4 let double: Double = (n: number) => { return n  };
5 double(5);
```

Listing 5.6: Callable object

Classes TypeScript also supports classes, which are an alternative to interfaces to describe the type of objects. In order to support existing (undefined) inter-property constraints in web applications, inter-property constraints are incorporated in TIPC as an extension of interfaces instead of classes such that they can

be used to type object literals. We will revisit classes in combination with inter-property constraints in the future work section of this dissertation (Section 10.3).

5.5 Null-checking Mode

When a variable or object property that does not exist gets accessed, JavaScript returns the value `undefined`. The value `undefined` can be assigned to a variable of any type. The same applies to the value `null`, which is used to indicate the empty object. However, this behaviour can be undesirable when developers want to ensure that a variable definitely contains a value (other than `undefined` or `null`).

To change this behaviour, TypeScript provides the `strictNullChecks` mode. When this flag is enabled, this means that the underlying assignment rules of TypeScript change. More specifically, it is not allowed to assign `null` and `undefined` to variables of any other type. However, developers can explicitly allow the assignment of `null` and `undefined` by assigning a union type: for example:
`let x: string | undefined = undefined`.

The type system of TypeScript assigns a union type to optional properties and parameters, which combines the type originally assigned to the property/parameter combined with the `undefined` type. The union type is sufficient to express the optionality: an optional type is either present (of the original type) or absent (undefined).

The strict null-checking mode is essential when dealing with inter-property constraints, which requires that the type system is able to ensure the presence or absence of properties.

5.6 Occurrence Typing

In dynamically typed programming languages, developers cannot rely on a type system to know more about the type of a variable. Instead, they have to rely on conditional tests on a variable when they want to have guarantees about its structure.

When a type system is retrofitted on a dynamically typed programming language, it can take this runtime type information into account when determining the type of a variable. Listing 5.7 shows a function definition that takes one argument `x` of type `string | undefined`. Inside the function definition, there is an `if` statement that serves as a type guard to check whether `x` is defined. Because of the type guard, TypeScript can safely assign the type `string` to `x`, instead of the union type. Similarly, `x` will certainly be undefined in the false branch.

```
interface PMId {
    text: string;
    user_id: number;
}
interface PMName {
    text: string;
    screen_name: string;
}

function isId(pm: any): pm is PMId {
    return pm.user_id !== undefined;
}

function foo(pm: PMId | PMName) {
    pm.user_id; //ERROR
    if (isId(pm)) {
        pm.user_id; //OK
    }
}
```

Listing 5.8: Type guards with type predicate functions

```
function foo(x: string | undefined) {
  if(x !== undefined) {
    x = "foo";      // :: string
  } else {
    x = undefined; // :: undefined
  }
}
```

Listing 5.7: Occurrence typing

It is also possible in TypeScript to define type guards for non-primitive types as well. Listing 5.8 shows an example. There are two default TypeScript interfaces, which define variants of the `PrivateMessage` interface: one identifies the user with an ID, the other with a name. The function `isId` is a type predicate: it verifies whether the property `user_id` is present. With the `is` keyword, a developer can indicate that a parameter is of a certain type whenever the predicate returns true. The function `foo` shows an example: it receives one parameter which is either a `PMId` or a `PMName`. Only inside the `if` statement, which calls the type predicate `isId`, the type system will allow accessing the `user_id` property.

Chapter 3 has already shown how occurrence typing can also be beneficial while programming with inter-property constraints. However, it is not possible to simply reuse TypeScript's occurrence typing for inter-property constraints in

TIPC. While TypeScript is able to narrow basic types, we have seen in Chapter 4 that the type system needs to perform extra operations to narrow the type of interfaces with inter-property constraints.

5.7 Type Declaration Files

Providing a typed variant of JavaScript is especially useful in large JavaScript projects such as web applications. These web applications often use a variety of JavaScript libraries. Instead of only allowing TypeScript libraries for web applications written in the TypeScript programming language, TypeScript has a mechanism that allows TypeScript files to include JavaScript libraries, to wit *typed declaration files*.

A type declaration file defines a set of interfaces for all methods and variables in the API of a JavaScript library. A web application written in TypeScript has to include these type declaration files for every JavaScript library they use, using the triple-slash notation:

```
1 /// <reference path='typed_definitions/library.d.ts' />
```

As a community effort, there already exist typings for about 5000 JavaScript libraries[2]. Together with the optional typing, this facilitates the transition from JavaScript projects to TypeScript projects.

5.8 Conclusion

Because TypeScript is designed to support JavaScript programs, it has characteristics that are not typically found in statically typed programming languages. In this chapter, we have covered the idiosyncratic features of TypeScript, as well as features that are key for inter-property constraints. In the next chapter, we present the formalisations of TIPC, a variant of TypeScript with interfaces with inter-property constraints. For clarity, the atypical features that are unrelated to inter-property constraints are omitted from the formalisations. As TypeScript is unsound, the formalisations are part of a subset of TypeScript that only contains sound features.

[2] https://github.com/DefinitelyTyped/DefinitelyTyped

Chapter 6

TIPC: Formalisation

This dissertation started with the introduction of inter-property constraints: constraints between properties. We showed how inter-property constraints can be checked at compile-time by incorporating them into interface definitions. Interfaces with constraints between properties require a different way of type checking object creations, property accesses and property updates. Moreover, assignments to interface instances have to ensure that constraints remain satisfied. In Chapter 4, we informally showed how a type system can guarantee type safety in the light of inter-property constraints. Even though the examples were all written in TIPC, the requirements and solutions presented in Chapters 3 and 4 are applicable to any statically typed object-oriented programming language. In this chapter, we present the formalisations of this programming language.

In Section 6.1, we discuss the formalism upon which TIPC is based and discuss which changes are made to that basis before adding support for inter-property constraints. Next, we discuss the syntax (Section 6.2), typing rules (Section 6.3) and semantics (Section 6.4) of TIPC, including the extensions needed for incorporating inter-property constraints. Finally, we prove the soundness of TIPC in Section 6.5.

The formalisation presented in this chapter is an extended version of the formalisation presented in Oostvogels et al. [2018b] (Sections 4, 5 and 6). This chapter includes a full set of rules for evaluating and typing sequences, which did not fit in the page limit of said paper, as well as more detailed proofs of soundness.

6.1 SafeFTS: a Formalisation of TypeScript

The formalisations presented in this chapter are based upon those presented by Bierman et al. [2014]. They present a formalisation of TypeScript (version

0.9.5), including all the features that are so characteristic to JavaScript. The formalisation is presented in two stages: the first stage is a safe calculus (called safeFTS) which contains the core features of TypeScript. This calculus describes a subset of TypeScript including features such as structural typing, contextual types and function scoping. In the second stage, the formalisation covers the fact that TypeScript is purposefully unsound by extending safeFTS to a production-ready calculus. This calculus (prodFTS) includes unsound features such as downcasting, covariance and indexing.

TIPC reuses most of safeFTS's features, but there are some differences between safeFTS and TIPC:

Null-checking mode SafeFTS allows assigning `undefined` to variables of any type; however, this thwarts TIPC's ability to guarantee the presence or absence of properties. Therefore, TIPC diverges from safeFTS and disallows such assignments. This coincides with the *strict null checking mode* which was added in TypeScript 2.0.

Contextual typing In JavaScript (and TypeScript), it is common to work with callbacks and event handlers. The types of these functions are influenced by the *context* in which they are used. To support this usage, safeFTS supports contextual typing which augments the regular type inference with context information. As contextual typing is orthogonal to addition of inter-property constraints to interfaces, the formalisations that deal with contextual typing are omitted for clarity. Adding contextual typing to TIPC would not be different from the contextual typing rules in safeFTS.

Block scoping Section 5.3 introduced the atypical scope mechanism in JavaScript: variable declarations are function scoped instead of block scoped. To accurately represent this, safeFTS uses bespoke typing rules. TypeScript introduced alternative variable declarations which are block scoped, using `let`. Scoping mechanisms are orthogonal to object creation and interfaces. TIPC only supports block-scoped variable declarations: safeFTS's function scoping is omitted, but can be trivially added to TIPC.

In Sections 6.2 to 6.4, we indicate the additions to the formalisation of safeFTS with a grey background.

6.2 Syntax

In this section, we present the syntax of TIPC. TIPC is a variant of TypeScript with support for interfaces with inter-property constraints.

Figure 6.1 presents the syntax of expressions and statements in TIPC. The meta variables `e` and `f` range over expressions, `x` ranges over identifiers, `l` ranges over literals, `a` ranges over property assignments, `n` ranges over property names, and `s` and `t` range over statements. In addition to the standard TypeScript syntax, TIPC only adds the language construct `assign`.

Next, Figure 6.2 on page 84 presents the types in TIPC. The meta variables `R`, `S` and `T` range over types, `P` ranges over primitive types, `O` ranges over object types, `I` ranges over interface types, `L` over literal types, and `M` and `N` range over type members. Except for the interface types, types in TIPC are equal to those in safeFTS.

6.2.1 Expressions

TIPC features basic language expressions such as identifiers `x`, literals `l`, assignment and binary operators. Literals can be numbers n, strings s, the boolean constants `true` and `false`, the empty object `null`, or `undefined` which is returned when accessing a property that is not present in an object.

Objects are defined using object literals, which map property names (`n`) to the result of expressions. Accessing a property of an object happens using the dot notation. In TypeScript, property accesses can also happen with the square bracket notation (see Section 5.2), This notation can be used to address properties which are not valid identifiers. It also enables *computed property accesses*, where the property is determined at runtime. The square bracket notation is not supported by TIPC, as type safety cannot always be guaranteed.

Multiple properties of an object can be updated at once using `assign`, introduced in Section 3.5. This language construct is a functional version of JavaScript's `Object.assign`. `assign` returns a new object, instead of updating it: the resulting object contains all property of the first argument and where properties from the second argument are either updated (when already present in the first argument) or added (otherwise).

Using the assignment operator $=$, expressions can be assigned to variables or object properties. Two expressions can be combined using several binary operators (such as `<`, `+`, `===`), which are abstracted in the operator $\otimes$.

Function expressions are similar to those in JavaScript, but with type annotations for the parameters and the return type. TIPC requires without loss of

$\mathtt{l} \in$ Literals	::=	n	(Number)
		s	(String)
		`true`	(Boolean value)
		`false`	(Boolean value)
		`null`	(Empty object)
		`Undefined`	(Undefined property)
$\mathtt{e}, \mathtt{f} \in$ Expressions	::=	`x`	(Identifier)
		`l`	(Literal)
		$\{\overline{\mathtt{a}}\}$	(Object literal)
		`e.n`	(Property access)
		`assign`$(\mathtt{e}, \{\overline{\mathtt{a}}\})$	(Assign operator)
		$\mathtt{x} = \mathtt{f}$	(Variable assignment)
		$\mathtt{e.n} = \mathtt{f}$	(Property assignment)
		$\mathtt{e} \otimes \mathtt{f}$	(Binary operator)
		`function` $(\overline{\mathtt{x}} : \overline{\mathtt{S}})$: `T` $\{\overline{\mathtt{s}}\}$	(Function expression)
		$\mathtt{e}(\overline{\mathtt{f}})$	(Function call)
		`<T>e`	(Type assertion)
$\mathtt{a} \in$ Property assignments	::=	$\mathtt{n} : \mathtt{e}$	(Property assignment)
$\mathtt{s}, \mathtt{t} \in$ Statements	::=	`e;`	(Expression statement)
		`if` (`e`) $\{\overline{\mathtt{s}}\}$ `else` $\{\overline{\mathtt{t}}\}$	(If statement)
		`return;`	(Return statement)
		`return e;`	(Return value statement)
		`let x:T` $= \mathtt{e}$	(Variable declaration)

Figure 6.1: Syntax of TIPC

generality that the body of a function contains a `return` statement per execution path.

Expressions can be cast to a type using angular brackets. Contrary to TypeScript, the type system of TIPC will only allow casts when the cast is known to be type safe. This is discussed in Section 6.3, page 91.

Several of the expressions use the sequencing notation (the line over a meta variable, such as $\overline{\mathtt{f}}$). The empty sequence is denoted with $\bullet$ and concatenation is denoted using a comma. A sequence of expressions is written as $\overline{\mathtt{e}}$ and is short for $\mathtt{e}_1, \ldots \mathtt{e}_\mathrm{n}$, with n the length of the sequence. A sequence of property assignments $\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}$ is an abbreviation for $\{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_\mathrm{n} : \mathtt{e}_\mathrm{n}\}$. Similarly, $(\overline{\mathtt{x}} : \overline{\mathtt{T}})$ is a sequence of function arguments $(\mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_\mathrm{n} : \mathtt{T}_\mathrm{n})$. Sequences of property names and

function parameter names should contain no duplicates.

6.2.2 Statements

The lower part of Figure 6.1 contains the statement syntax of TIPC. A statement can be an expression, an `if` statement, a `return` statement, or a variable declaration. A `return` statement can contain an optional return value. TIPC only features variable declarations where the type and the value for the variable are both provided. A sequence of statements $\overline{\texttt{s}}$ is short for $\texttt{s}_1, \ldots, \texttt{s}_n$.

6.2.3 Types

Figure 6.2 shows the types of TIPC. There are three kinds of types: the top type `any`, primitive types (`P`) and object types (`O`).

TIPC contains six primitive types: `number`, `string`, `boolean`, `void` (used whenever a `return` statement does not contain a value), `Null` (with a capital, to indicate the type of the literal `null`) and `Undefined` (also with a capital to indicate the difference with the corresponding literal).

An object type is represented by either a literal type or an interface type. An object literal type maps property names onto types. The declaration of interface types is covered further below.

Note that functions do not have a separate type in TIPC. Instead, they are represented as a callable object (as explained in Section 5.4) that contains one anonymous field (a function) with its type of the form $(\overline{\texttt{x}} : \overline{\texttt{S}})\texttt{:T}$. For example, the type of a function that expects two parameters `a` and `b` of type number and returns a number is written as follows:

$$\{(\texttt{a} : \texttt{number}, \texttt{b} : \texttt{number}) : \texttt{number}\}$$

A sequence of types is denoted as $\overline{\texttt{T}}$, and the sequence of properties, parameters and call signatures is analogous to their corresponding value sequences.

Interfaces

Interfaces play a key role in incorporating inter-property constraints in TIPC. To include complex presence constraints in interfaces, the interface declaration in TIPC is different from other languages. Figure 6.3 shows the declarations in TIPC.

TIPC interfaces first list the property (field or method) names, together with their types as usual. By default, all properties are optional. Constraints on the presence of a property are specified in the `constraining` section, using the syntax

$$
\begin{array}{rcll}
\texttt{R}, \texttt{S}, \texttt{T} \in \text{Types} & ::= & \texttt{any} & \\
 & & \texttt{P} & \\
 & & \texttt{O} & \\
\texttt{P} \in \text{Primitive types} & ::= & \texttt{number} & \\
 & & \texttt{string} & \\
 & & \texttt{boolean} & \\
 & & \texttt{void} & \\
 & & \texttt{Null} & \\
 & & \texttt{Undefined} & \\
\texttt{O} \in \text{Object types} & ::= & \texttt{I} & \text{(Interface type)} \\
 & & \texttt{L} & \text{(Object literal type)} \\
\texttt{L} \in \text{Object literal types} & ::= & \{\overline{\texttt{M}}\} & \\
\texttt{M}, \texttt{N} \in \text{Type members} & ::= & \texttt{n:T} & \text{(Property)} \\
 & & (\overline{\texttt{x}} : \overline{\texttt{S}})\texttt{:T} & \text{(Call signature)}
\end{array}
$$

Figure 6.2: Types of TIPC

$$
\texttt{D} \in \text{Declarations} ::= \begin{cases}
\texttt{interface I } \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\} & \\
\texttt{interface I extends } \overline{\texttt{I}} \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\} & \\
\qquad\qquad (\overline{\texttt{I}} \text{ non-empty})
\end{cases}
$$

Figure 6.3: Declarations in TIPC

$$\text{Property lookup (1)}\ \frac{\Sigma_i(\texttt{I}) = \texttt{interface I } \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}}{\mathit{properties}(\texttt{I}) = \{\overline{\texttt{n}} : \overline{\texttt{T}}\}}$$

$$\text{Property lookup (2)}\ \frac{\Sigma_i(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{I}} \ \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}}{\mathit{properties}(\texttt{I}) = \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \cup \mathit{properties}(\overline{\texttt{I}})}$$

$$\text{Constraint lookup (1)}\ \frac{\Sigma_i(\texttt{I}) = \texttt{interface I } \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}}{\mathit{constraints}(\texttt{I}) = \{\overline{\texttt{c}}\}}$$

$$\text{Constraint lookup (2)}\ \frac{\Sigma_i(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{I}} \ \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}}{\mathit{constraints}(\texttt{I}) = \{\overline{\texttt{c}}\} \cup \mathit{constraints}(\overline{\texttt{I}})}$$

Figure 6.4: Definition of *properties* and *constraints*

presented in Section 3.1. Constraints *between* the presence of properties can be listed in this section as well. Interfaces can inherit properties and constraints from other interfaces.

To retrieve the properties and constraints from a given interface, we define two auxiliary functions *properties* and *constraints* in Figure 6.4. *properties* returns all properties of the interface, and its superinterfaces. Retrieving properties of interfaces in TIPC is unaffected by the (inter-property) constraints. Analogous to the inheritance of properties, constraints from the superinterfaces are simply accumulated by the *constraints* function.

Before the analysis starts, all interface declarations are gathered and stored in a mapping Σ_i[1] of interface names `I` to their respective declaration `D`. As in safeFTS, a program is a pair $(\Sigma_i, \overline{\texttt{s}})$ containing an interface table and a sequence of statements. TIPC requires every interface to satisfy a set of sanity conditions:

1. For every $\texttt{I} \in \mathrm{dom}(\Sigma_i)$, $\Sigma_i(\texttt{I}) = \texttt{interface I } \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}$ or $\Sigma_i(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{I}} \ \{\overline{\texttt{n}} : \overline{\texttt{T}}\} \texttt{ constraining } \{\overline{\texttt{c}}\}$;

[1] The interface mapping Σ_i is not to be confused with the heap type Σ, introduced in Section 6.5.

2. for every interface name `I` appearing anywhere in Σ_i and $\overline{\mathtt{s}}$, it is the case that $\mathtt{I} \in \mathrm{dom}(\Sigma_i)$;

3. there are no cycles in the dependency graph induced by the `extends` clauses of the interface declarations defined in Σ_i;

4. interfaces may not override properties that are already defined in a super-interface;

5. for every interface name `I` in $\mathrm{dom}(\Sigma_i)$, there exists at least one valuation that satisfies the constraints $\mathit{constraints}(\mathtt{I})$. The valuation assigns truth values to proposition symbols, where the proposition symbols map onto property names and the truth values indicate the presence or absence of those properties;

6. for every interface name `I` in $\mathrm{dom}(\Sigma_i)$, none of the properties of `I` is allowed to be of type `any`, `void` or `Undefined`.

The first three sanity conditions are identical to those of safeFTS. The first sanity condition requires that every interface in the interface table corresponds to a valid interface declaration. The second sanity condition ensures that every interface in the TIPC program has a corresponding interface declaration in the interface table. The third sanity check prevents infinite interface definitions when an interface that extends itself (directly or indirectly). The fourth sanity check prevents shadowing a property.

While the first four sanity conditions are fairly standard for state-of-the-art interfaces, the latter two sanity conditions are specifically for interfaces with inter-property constraints. The fifth condition prevents the declaration of interfaces with inherent contradictions: it requires that the constraints of an interface are *satisfiable*. This ensures that there is at least one object with a combination of present and absent properties that is a valid instance of that interface. The sixth condition disallows types `any`, `void` and `Undefined` for all properties of an interface. This prevents the assignment of `undefined` to an object property (as the value `undefined` can only be assigned to variables of these three types), which — at runtime — is equal to an absent property. This way, we avoid that the presence constraints in the interface are circumvented: a property may only become absent when the constraints indicate this is safe.

By default, Σ_i contains four predefined interfaces: Object, String, Number and Boolean. The latter three form the interface equivalent of the corresponding primitive type.

6.3 Typing Rules

In this section we present the type system of TIPC. Each section contains the relevant typing rules. At the end of this chapter, Figures 6.9 to 6.11 (pages 120 to 122) list all typing rules for the expressions and statements.

The typing judgement is written as follows: $\Gamma \vdash \texttt{e} : \texttt{T}$, where given an environment Γ the expression `e` is of type `T`. An environment Γ maps variables to types ($\overline{\texttt{x}} : \overline{\texttt{T}}$) and is extended as follows: $\Gamma, \texttt{x} : \texttt{T}$. For sequences, we write $\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}}$ as shorthand for $\Gamma \vdash \texttt{e}_1 : \texttt{T}_1, \ldots, \Gamma \vdash \texttt{e}_\text{n} : \texttt{T}_\text{n}$, with n the length of the sequence.

The rules that do not (directly) deal with interfaces are explained in the following paragraphs. They are identical to those in safeFTS.

I-Id The type of a variable `x` is looked up in the environment.

$$\text{I-Id}\frac{}{\Gamma, \texttt{x:T} \vdash \texttt{x:T}}$$

I-Number, I-String, I-Bool, I-Null and I-Undefined These typing rules cover the typing of literals in TIPC. To indicate the difference between the literal and its type, the type of the literals `null` and `undefined` start with a capital: `Null` and `Undefined`.

$$\text{I-Number}\frac{}{\Gamma \vdash n : \texttt{number}} \qquad \text{I-String}\frac{}{\Gamma \vdash s : \texttt{string}}$$

$$\text{I-Bool}\frac{}{\Gamma \vdash \texttt{true}, \texttt{false} : \texttt{boolean}} \qquad \text{I-Null}\frac{}{\Gamma \vdash \texttt{null} : \texttt{Null}}$$

$$\text{I-Undefined}\frac{}{\Gamma \vdash \texttt{undefined} : \texttt{Undefined}}$$

I-ObLit The type of an object literal is a mapping of all property names $\overline{\texttt{n}}$ onto the type of their expressions $\overline{\texttt{e}}$.

$$\text{I-ObLit}\frac{\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}}}{\Gamma \vdash \{\overline{\texttt{n}} : \overline{\texttt{e}}\} : \{\overline{\texttt{n}} : \overline{\texttt{T}}\}}$$

I-Op A binary function call only receives a certain type when the parameters have the expected type. Here, $\otimes$ is a stand-in for commonly used binary operators on literals, such as `+`, `*` and `-`. We assume a table of $(\texttt{S}_0 \otimes \texttt{S}_1) = \texttt{T}$ exists.

$$\text{I-Op}\frac{\Gamma \vdash \texttt{e} : \texttt{S}_0 \quad \Gamma \vdash \texttt{f} : \texttt{S}_1 \quad \texttt{S}_0 \otimes \texttt{S}_1 = \texttt{T}}{\Gamma \vdash \texttt{e} \otimes \texttt{f:T}}$$

$$
\mathit{lookup}(\mathtt{S},\mathtt{n}) = \begin{cases}
\mathit{lookup}(\text{Number},\mathtt{n}) & \text{if } \mathtt{S} = \mathtt{number} & (1)\\
\mathit{lookup}(\text{Boolean},\mathtt{n}) & \text{if } \mathtt{S} = \mathtt{boolean} & (2)\\
\mathit{lookup}(\text{String},\mathtt{n}) & \text{if } \mathtt{S} = \mathtt{string} & (3)\\
\mathtt{T} & \text{if } \mathtt{S} = \{\overline{\mathtt{M}}_0, \mathtt{n}\!:\!\mathtt{T}, \overline{\mathtt{M}}_1\} & (4)\\
\mathit{lookup}(\text{Object},\mathtt{n}) & \text{if } \mathtt{S} = \{\overline{\mathtt{M}}\} \text{ and } \mathtt{n} \notin \{\overline{\mathtt{M}}\} & (5)\\
\mathtt{T} & \text{if } \mathtt{S} = \mathtt{I} \text{ and } \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}) & (6)\\
 & \quad \text{and } \mathit{constraints}(\mathtt{I}) \vDash_\ell \mathtt{present}(\mathtt{n}) & \\
\mathtt{Undefined} & \text{if } \mathtt{S} = \mathtt{I} \text{ and } \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}) & (7)\\
 & \quad \text{and } \mathit{constraints}(\mathtt{I}) \vDash_\ell \neg\mathtt{present}(\mathtt{n}) &
\end{cases}
$$

Figure 6.5: Definition of lookup

6.3.1 Property Lookup

The rule **I-Prop** covers the typing rule for looking up a property of an object in TIPC. Just as in safeFTS, it first retrieves the type of the object of which a property is looked up. Next, the type of the object serves as a parameter of a call to *lookup*, together with the name of the property being looked up.

$$\text{I-Prop}\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \mathit{lookup}(\mathtt{S},\mathtt{n}) = \mathtt{T}}{\Gamma \vdash \mathtt{e.n}\!:\!\mathtt{T}}$$

The *lookup* function is defined in Figure 6.5, and results in the type of property `n` in the object type `S`. The behaviour of *lookup* depends on the kind of object type that was provided as first argument. The following three paragraphs cover the different cases. The first two cases do not deal with interface definitions and are thus identical to those in safeFTS.

`S` is a primitive type The first three cases cover the *lookup* of a property type when the object type is a primitive type. In that case, TIPC will not raise an error (in order to model the behaviour in JavaScript and TypeScript). Instead, the property is looked up in the interface type that is associated with the primitive type.

`S` is an object literal type Cases 4 and 5 cover *lookup* when the object type is an object literal type. When the property is found in the object literal type

(case 4), *lookup* returns the type for that property according to the object literal type. When the property is not found in the object literal type (case 5), the *lookup* function searches the property in the supertype of all object types: Object. This interface contains functionality that is common for all objects, such as the functions `assign`, `getOwnPropertyNames` and `values`.

`S` is an interface type Cases 6 and 7 in the definition of *lookup* show how a property is looked up in a TIPC interface. First, the property is looked up in the properties of the interface, including properties of superinterfaces. However, only looking up the property — as we did for properties of object literal types — does not suffice. Next to being part of the property list, *lookup* also has to take the interface constraints on the presence of its properties into account. Only when the property is guaranteed to be present (such as `text` in `PrivateMessage`), it is safe for *lookup* to return the intended type for that property. In the case that the property is certainly absent, the *lookup* function can indicate this by returning the `Undefined` type. As we already explained in Chapter 4, the presence and absence of properties is verified using a logical entailment.

Note that *lookup* is a partial function: it is not defined for all possible arguments. More specifically, looking up the type of a property in a type is undefined when the first argument of a call to *lookup* is `any`, `void`, `Null`, `Undefined` or a call signature. Furthermore, the *lookup* function is also undefined when looking up a property of an interface type for which neither the presence nor the absence can be guaranteed (such as for `user_id` and `screen_name` in `PrivateMessage`).

A note on the `unknown` type. The *lookup* function is very restrictive regarding property accesses of interfaces: the function is undefined when the presence or absence of a property cannot be guaranteed. TypeScript 3.0 introduced the `unknown` type, a type-safe alternative to the `any` type. This means that anything is assignable to a variable of type `unknown`, but a variable of type `unknown` can only be assigned to other variables of type `unknown` or `any`. Only when a developer performs the necessary type tests, the type of an `unknown` variable can be narrowed down to a more specific type.

On first sight, *lookup* could assign the `unknown` type to properties that are not certainly present or absent. This way, the type system would enforce those properties can only be accessed after their presence or absence is confirmed using an `if` statement. However, the `unknown` type is not as restrictive as necessary, as `unknown` instances can still be assigned to each other. This would result in an unsound type system. For the `PrivateMessage` example, this means that the

type system would accept the following assignment: both property accesses would receive the `unknown` type and could – following the assignment compatibility rules for `unknown` – be safely assigned to each other. However, this would result in a `PrivateMessage` object that contains both a user ID and a screen name.

```
1 function foo(pm: PrivateMessage) {
2   pm.user_id = pm.screen_name;
3 }
```

6.3.2 Assignment Compatibility

A key part of the type system is verifying whether an expression may be assigned to a variable or object property. For a type system, this boils down to verifying whether the type of that expression is assignable to the type of the variable or object property. There are several situations for which the type system needs to perform such an *assignment compatibility* check: an assignment, a function definition, a function call and a cast expression.

The assignment compatibility relation is defined in Figure 6.10 and is written down as $\mathtt{S} \leqq \mathtt{T}$. This means that any valid value for `S` will also be a valid value for `T`. $\overline{\mathtt{S}} \leqq \mathtt{T}$ is an abbreviation for $\mathtt{S}_1 \leqq \mathtt{T}, \ldots, \mathtt{S}_n \leqq \mathtt{T}$ and we write $\overline{\mathtt{S}} \leqq \overline{\mathtt{T}}$ as shorthand for $\mathtt{S}_1 \leqq \mathtt{T}_1, \ldots, \mathtt{S}_\mathtt{n} \leqq \mathtt{T}_\mathtt{n}$.

In this section, we first discuss the typing rules for the four expressions that use the assignment compatibility. Afterwards, we discuss the details of the assignment compatibility rules themselves. Although the four rules discussed below look similar to those in safeFTS, their behaviour will be different because of the assignment compatibility rules in TIPC.

I-Assign This rule covers the assignment of an expression to variables and object properties, which are the only two allowed expressions on the left-hand side of an assignment (as defined in Section 6.2). In TIPC, an expression `f` may only be assigned to an expression `e` if: 1) both expressions are type safe, and 2) the source expression (`f`) has a type that is *assignable to* ($\leqq$) the type of the target expression `e`. While the type system assigns the type of `f` to the assignment expression, the type of `e` remains unaltered. Note that the syntax of TIPC only allows identifiers and object properties as left-hand side expressions.

$$\text{I-Assign}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \mathtt{f} : \mathtt{T} \qquad \mathtt{T} \leqq \mathtt{S}}{\Gamma \vdash \mathtt{e} = \mathtt{f} : \mathtt{T}}$$

I-Func For a function definition, the type system type-checks the function body $\overline{\texttt{s}}$ under an environment that is extended with the type declarations for the parameters $\overline{\texttt{x}}$. The environment is also extended with the `this` variable, which gets type `any`: it is impossible to correctly model the type of `this` at compile time, as the value of `this` depends on the calling context. When the return types of all branches of the function body $\overline{\texttt{R}}$ are all *assignable to* the declared return type `T`, the function definition receives as type a callable object type. This type models an object that contains one anonymous property with as type the call signature as expressed in the function definition.

$$\text{I-Func}\frac{\Gamma, \texttt{this} : \texttt{any}, \overline{\texttt{x}} : \overline{\texttt{S}} \vdash \overline{\texttt{s}} : \overline{\texttt{R}} \qquad \overline{\texttt{R}} \leqq \texttt{T}}{\Gamma \vdash \texttt{function}(\overline{\texttt{x}} : \overline{\texttt{S}}) : \texttt{T}\ \{\overline{\texttt{s}}\} : \{(\overline{\texttt{x}} : \overline{\texttt{S}}) : \texttt{T}\}}$$

I-Call For a function call, the type system requires the function to have a callable object type. Moreover, it also verifies that the types of the parameters of the function call are *assignable to* the declared types in the function type.

$$\text{I-Call}\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \{(\overline{\texttt{x}} : \overline{\texttt{S}}) : \texttt{R}\} \\ \Gamma \vdash \overline{\texttt{f}} : \overline{\texttt{T}} \quad \overline{\texttt{T}} \leqq \overline{\texttt{S}}\end{array}}{\Gamma \vdash \texttt{e}(\overline{\texttt{f}}) : \texttt{R}}$$

I-Assert As TIPC only allows *safe* casts, casting an expression `e` to a type `T` is only allowed when the type of the expression is *assignable to* `T`.

$$\text{I-Assert}\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \texttt{S} \\ \texttt{S} \leqq \texttt{T}\end{array}}{\Gamma \vdash \texttt{<T>e} : \texttt{T}}$$

The following paragraphs discuss the assignment compatibility rules (defined in Figure 6.10) in detail. They are based on the assignment compatibility rules of safeFTS, but differ in several ways. First, the assignability of the types `Null` and `Undefined` in TIPC is defined according to the strict null checking rules of TypeScript. Second, the assignment compatibility rules of TIPC cover the compatibility of interface types with others. We use the following notation to indicate the well-formedness of a type: $\texttt{T} \vdash \diamond$.

A-Trans Assignment compatibility is transitive: when a type `R` is assignable to a type `S` and `S` is assignable to another type `T`, then `R` is assignable to `T`.

$$\text{A-Trans}\ \frac{\mathtt{R} \leqq \mathtt{S} \quad \mathtt{S} \leqq \mathtt{T}}{\mathtt{R} \leqq \mathtt{T}}$$

A-Refl Assignment compatibility is also reflexive: a well-formed type is assignable to itself.

$$\text{A-Refl}\ \frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{S}}$$

A-AnyR Any type can be assigned to `any`, as long as that type is well-formed.

$$\text{A-AnyR}\ \frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{any}}$$

A-Undefined In order to be able to reason about present and absent properties, TIPC employs the strict null-checking mode. As a consequence, the values `null` and `undefined` can only be assigned to variables of types `Null` resp. `Undefined`, and `any`. This is already covered by the reflexivity rule and the assignment compatibility rule for `any`. The only exception is that the value `undefined` may also be assigned to variables of type `void`. This is covered in the rule A-Undefined.

$$\text{A-Undefined}\ \frac{}{\mathtt{Undefined} \leqq \mathtt{void}}$$

A-Prim When a primitive type is assigned to another type, the assignment compatibility rules use a helper function $\mathcal{I}$. This function takes a primitive type and returns the equivalent interface type. For example, $\mathcal{I}(\mathtt{boolean})$ returns an object which contains the methods `toString` and `valueOf`. A primitive type is assignable to another type when its interface type is assignable to that type.

$$\text{A-Prim}\ \frac{\mathcal{I}(\mathtt{P}) \leqq \mathtt{T}}{\mathtt{P} \leqq \mathtt{T}}$$

A-Object The rule A-Object covers the assignment compatibility of object literal types. An object literal type can be assigned to another object literal type when all the properties of the target object are also present in the source object. As TIPC supports width subtyping, it is possible that the source object contains more properties than the target object. The common properties of both object literal types have to be pairwise assignable.

$$\text{A-Object}\ \frac{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \vdash \diamond \quad \overline{\mathtt{M}}_1 \leqq \overline{\mathtt{M}}_2}{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \leqq \{\overline{\mathtt{M}}_2\}}$$

A-Prop The assignment compatibility of object properties is invariant: when the property names are identical, their types have to be identical too.

$$\text{A-Prop}\ \frac{\mathtt{T} \vdash \diamond}{\mathtt{n} : \mathtt{T} \leqq \mathtt{n} : \mathtt{T}}$$

A-CS and A-CS-Void The assignment compatibility rules for call signatures are divided into two rules, depending on the type of the return value of the target. When the return value of the target function signature is `void`, there is no need for an assignment compatibility check on the return values. Of course, the source return type needs to be well-formed. This is covered in A-CS-Void. Otherwise, the assignment of the function return types is covariant: the return type of the source needs to be assignable to the return type of the target. This is covered in A-CS. In both cases, the function argument types are contravariant: the parameter types of the target need to be assignable to the parameter types of the source.

$$\text{A-CS}\ \frac{\overline{\mathtt{T}} \leqq \overline{\mathtt{S}} \quad \mathtt{R}_1 \neq \mathtt{void} \qquad \mathtt{R}_0 \leqq \mathtt{R}_1}{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{R}_0 \leqq (\overline{\mathtt{y}} : \overline{\mathtt{T}}) : \mathtt{R}_1} \qquad \text{A-CS-Void}\ \frac{\overline{\mathtt{T}} \leqq \overline{\mathtt{S}} \quad \mathtt{R} \vdash \diamond}{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{R} \leqq (\overline{\mathtt{y}} : \overline{\mathtt{T}}) : \mathtt{void}}$$

A-Interface The rule A-Interface covers the assignment of interfaces to each other. These rules are more involved than assigning object literal types to each other, as the inter-property constraints on the presence of properties need to be taken into account as well. In general, common properties should have the same type, and interfaces must be at least as strict as the target interface to be considered assignment-compatible. Translating this using concepts from propositional logic, the rule is that an interface is assignable to another interface when the constraints of the target interface logically follow (or entail) from the constraints of the source interface.

Recall that Section 4.4 already showed that the structural differences between both property lists needs to be taken into account as well. For every property that is part of the source property list but absent in the target property list, a constraint indicating its absence is added to the conclusion of the logical entailment (c_0). By adding these constraints, the type system prevents width subtyping for interfaces. The other way around, properties that are part of the target interface but not part of the source interface result in extra absence constraints in the premise of the logical entailment (c_1). As Section 4.4.2 showed, generating extra absence constraints by comparing the property lists leads to more valid assignment compatibilities. $\bigwedge \overline{c}$ is used to denote $c_1 \wedge \cdots \wedge c_n$.

$$\text{A-Interface}\ \frac{\begin{array}{c}\forall \mathtt{n} : \mathtt{S} \in \mathit{properties}(\mathtt{I}_0) : (\mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_1) \implies \mathtt{S} = \mathtt{T}) \\ c_0 = \{\neg\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_0) \setminus \mathit{properties}(\mathtt{I}_1)\} \\ c_1 = \{\neg\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_1) \setminus \mathit{properties}(\mathtt{I}_0)\} \\ \mathit{constraints}(\mathtt{I}_0) \cup c_1 \vDash_\ell \bigwedge \mathit{constraints}(\mathtt{I}_1) \wedge \bigwedge c_0\end{array}}{\mathtt{I}_0 \leqq \mathtt{I}_1}$$

A-IntObj An interface type is only assignable to an object literal type when the interfaces property list is assignable to the object literal, and when the interface constraints guarantee the presence of all common properties.

$$\text{A-IntObj}\ \frac{\mathit{properties}(\mathtt{I}) \leqq \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} \qquad \mathit{constraints}(\mathtt{I}) \vDash_\ell \mathtt{present}(\overline{\mathtt{n}})}{\mathtt{I} \leqq \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

Due to width subtyping, the type of an object does not guarantee that *only* those properties are present at runtime (as can be seen in A-Object). This conflicts with interfaces which may require properties to be absent: the assignment of an object to an interface could possibly invalidate the interface constraints at runtime. As already discussed in Chapter 4, TIPC only allows the casting of a *literal* object to an interface. As a consequence, there is no assignment compatibility rule for assigning an object to an interface. This is covered by the rule I-AssertInf, which is covered in the next section.

Readers familiar with the work of safeFTS [Bierman et al., 2014] (the basis for the formalisation of TIPC) might notice that — contrary to safeFTS — the assignment compatibility relationship in TIPC is not coinductive. In safeFTS, interfaces are replaced by corresponding object literals. When an interface (indirectly) references itself in its field declarations, this can lead to an infinite type expansion. To

deal with this, safeFTS defines assignment compatibility as a coinductive relation, which guarantees termination. In TIPC, on the other hand, interfaces cannot be replaced by object literals, as interfaces may also contain constraints. Thus, assignment compatibility for interface fields with interface types in TIPC must be checked against the interface definition instead of using a coinductive relation.

6.3.3 Creating Interface Instances

The rule **I-AssertInf** covers the case where an object literal is cast to an interface. The type system only accepts type-safe casts, i.e. tests for which the properties of the object have the correct type *and* the presence and absence of properties form a valid valuation[2] of the interface constraints.

$$\text{I-AssertInf}\ \frac{\begin{array}{cc}\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{M}}\} & \{\overline{\mathtt{M}_p}\} = \{\mathtt{n} : \mathtt{T} \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}}\} \wedge \mathtt{T} \neq \mathtt{Undefined}\} \\ \{\overline{\mathtt{M}_p}\} \subseteq \mathit{properties}(\mathtt{I}) & c_p = \{\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_p}\}\} \\ \{\overline{\mathtt{M}_{np}}\} = \mathit{properties}(\mathtt{I}) \setminus \{\overline{\mathtt{M}_p}\} & c_{np} = \{\neg\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_{np}}\}\} \\ v = c_p \cup c_{np} & \hat{v}(\mathit{constraints}(\mathtt{I}))\end{array}}{\Gamma \vdash \texttt{<I>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}}$$

To generate the valuation function, the rule has to create presence constraints for properties that are present in the object literal, and create absence constraints for the properties that are a part of the interface property list but not part of the object literal type. Note that a property is considered absent when it is not in the object literal, or when its type is `Undefined`.

6.3.4 Updating Multiple Properties

This section covers the typing rules for updating multiple properties simultaneously, using the functional `assign` function. `assign` expects two arguments: the first argument is the object that needs to be updated and the second argument is an object that contains the new values for (some of) the properties. The type system has two rules for `assign`, depending on whether the type of its first argument is an object literal type (I-UpdateObj) or an interface type (I-UpdateInf).

I-UpdateObj When the type of the first argument of `assign` is an object literal type, I-UpdateObj simply combines the properties of the second argument with the first. The combination of two types is achieved using $\Subset$, which takes two object literal types and returns a new object literal type. This type initially contains the

[2] We refer to Section 4.1 for the definition of a valuation.

properties of the left-hand side object literal type. For every property of the right-hand side object literal type, the type of the property is either updated (when the property is already present in the left-hand side object literal type) or added (when the property was not present in the left-hand side object literal type).

$$\text{I-UpdateObj}\frac{\Gamma \vdash \texttt{e} : \{\overline{\texttt{M}}\} \qquad \Gamma \vdash \{\overline{\texttt{n}} : \overline{\texttt{e}}\} : \{\overline{\texttt{N}}\}}{\Gamma \vdash \texttt{assign}(\texttt{e}, \{\overline{\texttt{n}} : \overline{\texttt{e}}\}) : \{\overline{\texttt{M}}\} \Subset \{\overline{\texttt{N}}\}}$$

I-UpdateInf On the surface, I-UpdateInf is similar to I-UpdateObj: it combines the properties of the source interface instance `e` with new properties and produces an object of the same interface type `I`. However, care must be taken to verify that this does not invalidate `I`'s constraints: the type system has to check that all constraints imposed on the object remain satisfied after the update. As the second argument does not necessarily contain every property of the interface, it does not suffice to check whether the new properties satisfy all the constraints.

$$\text{I-UpdateInf}\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \texttt{I} \qquad \texttt{I}' = \mathit{slice}(\texttt{I}, \overline{\texttt{n}}, \mathit{constraints}(\texttt{I})) \qquad \Gamma \vdash \texttt{<I'>}\{\overline{\texttt{n}} : \overline{\texttt{e}}\} : \texttt{I}' \\ \overline{\texttt{n}} \in \mathit{dom}(\mathit{properties}(\texttt{I})) \qquad\qquad \overline{\texttt{n}} = \mathit{dom}(\mathit{properties}(\texttt{I}'))\end{array}}{\Gamma \vdash \texttt{assign}(\texttt{e}, \{\overline{\texttt{n}} : \overline{\texttt{e}}\}) : \texttt{I}}$$

To solve this, I-UpdateInf uses the *slice* function (defined below) to generate a sub-interface of `I` that contains a superset of the properties in $\overline{\texttt{n}}$ and a closed set of constraints on these properties. Given this generated interface, rule I-AssertInf is reused to verify whether the updated properties satisfy the applicable subset of constraints. An `assign` call fails if the second argument is not a correct valuation for the generated interface, if any of the updated properties are not declared in the interface `I`, or if not all properties of the generated interface are part of the second argument of `assign`.

To preserve soundness, `assign` does not actually modify its first argument; instead it returns a fresh object. Allowing `assign` to mutate the object would impose severe usage restrictions (such as in Flow [Chaudhuri et al., 2017] and RSC [Vekris et al., 2016]), or requires tracking aliases (such as in DJS [Chugh et al., 2012a]). This will be extensively discussed in Chapters 8 and 10.

slice returns the transitive closure of all properties and constraints of the given interface which are affected by the properties being updated. Formally, *slice* is defined as follows:

Definition *slice* expects three parameters: the interface which needs to be "sliced", a set of properties, and a set of constraints. The function uses an auxiliary function *fv* which takes a constraint and returns all referenced properties. We omit its trivial definition.

$$slice(\mathtt{I}, \overline{\mathtt{p}}, \overline{\mathtt{c}}) = \begin{cases} \mathtt{interface\ I'\ \{}\overline{\mathtt{p}}\mathtt{\}\ constraining\ \{}\overline{\mathtt{c}}\mathtt{\}} & \text{if } (\overline{\mathtt{p}}, \overline{\mathtt{c}}) \equiv (\overline{\mathtt{p}}', \overline{\mathtt{c}}') \\ slice(\mathtt{I}, \overline{\mathtt{p}}', \overline{\mathtt{c}}') & \text{otherwise} \end{cases}$$

$$\text{where } \overline{\mathtt{c}}' = \overline{\mathtt{c}} \cup \{\mathtt{c} \mid \mathtt{c} \in constraints(\mathtt{I}) \wedge fv(\mathtt{c}) \cap \overline{\mathtt{p}} \neq \emptyset\}$$
$$\overline{\mathtt{p}}' = \overline{\mathtt{p}} \cup \{fv(\mathtt{c}) | \mathtt{c} \in \overline{\mathtt{c}}'\}$$

slice first computes two sets: the set of constraints in which at least one of the properties (second argument of *slice*) occur, and the set of properties that occur in the new set of constraints. Given these two sets, there are two options:

- Either the properties and constraints are identical to the second and third argument of *slice*: in this case *slice* has resulted in a fixed point. *slice* returns a new subinterface of `I` containing the fixed point properties and constraints;
- When there is no fixed point, *slice* recursively calls itself with the new set of properties and constraints. This process is guaranteed to terminate as there is only a finite amount of properties and constraints in `I` from which *slice* can choose, and the set of properties constraints cannot shrink.

Note that in the case of dependency constraints, the definition of *slice* is sometimes more restrictive than necessary. More specifically, the presence or absence of the consequent `B` of the logical implication `A -> B` is irrelevant when the antecedent `A` is false. For example, properties of a dependency constraint such as `present(description) -> present(picture)` are part of one cluster, which means that both properties always have to be updated together. However, when `description` is set to `undefined`, the dependency constraint will be satisfied regardless of the presence or absence of `picture`. In the future, we plan on adding support for this improvement to the type system of TIPC.

6.3.5 Statement Typing

Finally, Figure 6.11 shows the typing rules for sequences, which are of the form $\Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}$, where given an environment Γ the sequence of statements $\overline{\mathtt{s}}$ has a set of return types $\overline{\mathtt{R}}$. A sequence of return types $\overline{\mathtt{R}}$ is short for $\mathtt{R}_1, \ldots, \mathtt{R}_n$ These return types are collected from all `return` statements in the sequence. When analysing a function definition, this is used by the type system to verify whether the types of all `return` statements are assignable to the declared return type.

Most of the sequence typing rules in TIPC are very similar to those in safeFTS. In TIPC, information from `if` statements is taken into account for property access checks, which is covered by an extra `if` statement sequence rule.

I-EmpSeq The type of an empty sequence is also denoted by $\bullet$.

$$\text{I-EmpSeq}\frac{}{\Gamma \vdash \bullet : \bullet}$$

I-ReturnVal The type of a sequence of statements is the set of types that are returned in these statements. In I-ReturnVal, the type system returns the type of a sequence of statements of which the first expression is a `return` statement. The type of the expression that is returned is added to the type of the rest of the statements $\overline{\mathtt{s}}$. Although these statements are unreachable because of the `return` statement, they are still type checked.

$$\text{I-ReturnVal}\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{return\ e}; \overline{\mathtt{s}} : \mathtt{T}, \overline{\mathtt{R}}}$$

I-Return The rule I-Return types a sequence of statements that starts with an empty `return` statement. In that case, the type `void` is added to the type of the rest of the statements.

$$\text{I-Return}\frac{\Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{return}; \overline{\mathtt{s}} : \mathtt{void}, \overline{\mathtt{R}}}$$

I-ExpSt This rule covers the case where the first statement in the sequence is an expression. Although the type system requires that the expression is well-typed, the type is not taken into account for the type of the entire sequence.

$$\text{I-ExpSt}\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{e}; \overline{\mathtt{s}} : \overline{\mathtt{R}}}$$

I-IfGeneral The type system considers two cases where type checking an `if` statement. When the condition verifies the presence of a property of an object with an interface type, the type system has to take special actions. This is covered in I-IfPresenceInterface. I-IfGeneral covers `if` statements on other expressions. This rule type checks the condition, the statements in the true branch and the

statements in the false branch. The resulting type is the set of the type of the true branch, false branch and the rest of the sequence.

$$\text{I-IfGeneral}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{boolean} \quad \Gamma \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1 \\ \Gamma \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2 \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}} : \overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}}}$$

I-IfPresenceInterface As already seen in Chapter 3 and Chapter 4, special action is required when a condition of an `if` statement contains a property presence test for a property of an object with an interface type. Analogous to the latent predicates in occurrence typing [Tobin-Hochstadt and Felleisen, 2010], the type system uses the presence tests inside conditions of `if` statements to refine interface types in the branches. Although Figure 6.11 only defines rules for a single pattern of conditional expressions, the typing rule can be generalised to inequalities and combined logical expressions, like in Tobin-Hochstadt and Felleisen [2010].

$$\text{I-IfPresenceInterface}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{x} : \mathtt{I} \quad \mathtt{n} : \mathtt{S} \in \mathit{properties}(\mathtt{I}) \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}} \\ \mathtt{I}^- = \mathit{addConstraint}(\mathtt{I}, \neg\mathtt{present}(\mathtt{n})) \quad \Gamma \uplus \mathtt{x} : \mathtt{I}^- \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1 \\ \mathtt{I}^+ = \mathit{addConstraint}(\mathtt{I}, \mathtt{present}(\mathtt{n})) \quad \Gamma \uplus \mathtt{x} : \mathtt{I}^+ \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathtt{x.n} \equiv \mathtt{undefined})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}} : \overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}}}$$

The typing rule first verifies whether the object has an interface type and that the accessed property is part of the property list of that interface. If that is the case, two variations of the original interface are created. One interface is extended with a constraint that indicates the absence of the tested property, the other interface is extended with a constraint indicating the presence. The expressions in the true (resp. false branch) are type checked against the environment in which the original interface is replaced with the first (resp. second) interface variant.

Generating the interface variants is done using the function *addConstraint*, which adds the constraints to the interface and performs a satisfiability check to verify that there are no inconsistent constraints in the extended constraint set. Its definition is trivial and omitted. In the case of inconsistencies (ie. when the formula $\mathtt{present}(\mathtt{n}) \wedge \neg\mathtt{present}(\mathtt{n})$ can be proven for any `n`), *addConstraint* will return the bottom type `Undefined`, preventing access to an invalid object.

Note that the type assignment for `x` is *overwritten* in both branches using $\uplus$, leaving type assignments for other variables as-is.

I-ITVarDec This rule type checks a sequence of expressions with an initialisation of a variable at the front. It uses the assignable compatibility check to verify whether the type of the right-hand side of the assignment is assignable to the declared type of the variable. The function *noDup* is used to avoid declaring a variable again. The remainder of the expressions in the sequence are type checked against an extended environment: using the $\uplus$ operator, the environment is extended with the new variable.

$$\text{I-ITVarDec}\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \mathtt{T} \leqq \mathtt{S} \quad \mathit{noDup}(\Gamma, \mathtt{x} : \mathtt{S}) \quad \Gamma \uplus \mathtt{x} : \mathtt{S} \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} : \mathtt{S} = \mathtt{e}; \overline{\mathtt{s}} : \overline{\mathtt{R}}}$$

6.4 Operational Semantics

TypeScript is a superset of JavaScript that adds static typing. However, after compilation, TypeScript emits JavaScript code in which all types are erased, which means that the semantics of TypeScript (and safeFTS and TIPC) are the same as those of JavaScript. However, we provide the operational semantics of TIPC, which will be used in Section 6.5 to prove its soundness. The evaluation rules of TIPC are almost identical to those of safeFTS, but TIPC extends the heap and evaluation rules with tags for interface objects and defines the evaluation of `assign` calls. Evaluation of an expression `e` requires a heap H and a scope chain L. We define these now.

Figure 6.6 shows the definition of a heap. A heap H is a partial function from locations (l) to heap objects (o) . A heap object is either a closure or an object map. A closure represents a function, and is a pair containing a lambda expression (where a function $\mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\}$ is represented by $\lambda\overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}$) and a scope chain L.[3] An object map represents an object literal, and is a partial function from variables (x) to values (v). A variable is either a program variable `x`, a property name `n`, or the internal properties `@this` or `@interface`. A value is a location l or a literal `l`. A result r is a value or a reference, and a reference is a pair containing a location and a variable.

An empty heap is indicated by `emp`, a heap cell by $l \mapsto o$, a heap lookup by $H(l, x)$, a heap update by $H[l \mapsto o]$ and the union of two disjoint heaps is indicated by $H_1 * H_2$. $H[(l, x) \mapsto v]$ updates or extends an object map l with the value v assigned to the variable x. $H(l, x)\downarrow$ is true iff $H(l, x)$ is defined. We define a helper function $\gamma(H, r)$ that returns r if r is a value, otherwise (i.e. r is a reference (l, x)) it returns $H(l, x)$ if defined and `undefined` otherwise. `null` is a distinguished location, and is not in the domain of the heap.

[3] The scope chain L is not to be confused with the object literal type `L`.

$H \in$ Heaps	::=	`emp`	(Empty Heap)
		$l \rightharpoonup o$	(Heap)
$o \in$ Heap Objects	::=	$\langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L \rangle$	(Closure)
		$x \rightharpoonup v$	(Object Map)
$x \in$ Variables	::=	`x`	(Program Variable)
		`n`	(Property Name)
		`@this`	(Internal Property)
		`@interface`	(Internal Property)
$v \in$ Values	::=	l	(Location)
		`l`	(Literal)
$r \in$ Results	::=	v	(Result Value)
		(l, x)	(Reference)
$L \in$ Scope Chains	::=	l_g	(Global JavaScript Object)
		$l : L$	(Scope Chain)

Figure 6.6: Heap in TIPC

The evaluation rules use a *scope chain* to model the treatment of variables in JavaScript: JavaScript resolves variables dynamically against a scope object. A scope chain is a list of locations of the scope objects, and $l : L$ is a concatenation of a location l to a scope chain L. Crucially, scope objects also reside on the heap. This simplifies our proofs. For each function call, a new scope object is created and prepended to the beginning of the scope chain. After evaluating the function call, that scope object is removed from the scope chain.

The initial configuration used to evaluate TIPC programs consists of a scope chain containing only the global JavaScript object l_g. This object resides on the initial heap and contains locations of JavaScript prototype objects (`Number`, `String`, etc) and functions (such as `parseInt`) which themselves reside on the heap.

The variable lookup function σ is defined as follows. It expects three arguments: a heap, a scope chain and a variable. When the variable x is defined in the location l of the heap H, σ returns that location. Otherwise, the function σ is recursively called with the next location in the scope chain.

$$\sigma(H, l : L, x) = \begin{cases} l & \text{if } H(l, x)\downarrow \\ \sigma(H, L, x) & \text{otherwise} \end{cases}$$

6.4.1 Evaluating Expressions

The evaluation of an expression `e` is written as follows: $\langle H_1, L, \texttt{e}\rangle \Downarrow \langle H_2, r\rangle$, with H_1 as initial heap and L as scope chain, evaluating to heap H_2 with result r. As we often need to evaluate expressions to values instead of references, we define $\langle H_1, L, \texttt{e}\rangle \Downarrow_v \langle H_2, v\rangle$ as the combination $\langle H_1, L, \texttt{e}\rangle \Downarrow \langle H_2, r\rangle$ and $\gamma(H_2, r) = v$.

The following paragraphs discuss the semantics for evaluating expressions in TIPC. Figure 6.12 lists all evaluation rules at the end of this chapter. The evaluation rules of TIPC are almost identical to those in safeFTS, but ignore block scoping (as discussed in Section 6.1).

E-Id This rule evaluates a variable expression. After looking up the location of the variable in the scope chain (using σ), this rule returns a reference (a combination of a location and the variable itself).

$$\text{E-Id}\ \frac{\sigma(H, L, \texttt{x}) = l}{\langle H, L, \texttt{x}\rangle \Downarrow \langle H, (l, \texttt{x})\rangle}$$

E-Lit Evaluating a literal does not depend on the heap or scope chain: it simply results in that literal.

$$\text{E-Lit}\ \frac{}{\langle H, L, \texttt{l}\rangle \Downarrow \langle H, \texttt{l}\rangle}$$

E-this When encountering a `this` keyword, the evaluation rule E-this first looks up the scope object of the internal keyword `@this`. E-this then dereferences `@this`.

$$\text{E-this}\ \frac{\begin{array}{c}\sigma(H, L, @\texttt{this}) = l_1 \\ H(l_1, @\texttt{this}) = l\end{array}}{\langle H, L, \texttt{this}\rangle \Downarrow \langle H, l\rangle}$$

E-Op For the evaluation of a binary operator call, first the two parameters are evaluated. Both parameters have to evaluate to literals, which are then combined using the binary operator.

$$\text{E-Op}\ \frac{\begin{array}{c}\langle H_0, L, \texttt{e}_1\rangle \Downarrow_v \langle H_1, \texttt{l}_1\rangle \\ \langle H_1, L, \texttt{e}_2\rangle \Downarrow_v \langle H_2, \texttt{l}_2\rangle\end{array}}{\langle H_0, L, \texttt{e}_1 \otimes \texttt{e}_2\rangle \Downarrow \langle H_2, \texttt{l}_1 \otimes \texttt{l}_2\rangle}$$

E-Oblit Evaluating an object literal expression first requires a new location in the heap. E-Oblit uses an auxiliary function *new* to create a new object map that contains one internal property `@this` that points to itself. Next, every property is evaluated and its resulting value is added to the freshly created object map. Finally, E-Oblit returns the location of the object map.

$$\text{E-ObLit}\frac{\begin{array}{c} H_1 = H_0 * [l \mapsto new(l)] \\ \langle H_1, L, \mathtt{e}_1\rangle \Downarrow_v \langle H_1', v_1\rangle \qquad H_2 = H_1'[(l, \mathtt{n}_1) \mapsto v_1] \\ \cdots \\ \langle H_m, L, \mathtt{e}_m\rangle \Downarrow_v \langle H_m', v_m\rangle \qquad H = H_m'[(l, \mathtt{n}_m) \mapsto v_m] \end{array}}{\langle H_0, L, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}\rangle \Downarrow \langle H, l\rangle}$$

$$new(l) = (\mathtt{@this} \mapsto l)$$

E-Assign The rule E-Assign evaluates both sides of the assignment. The left-hand side has to evaluate to a reference, while the right-hand side is evaluated to a value. Next, the heap is extended or updated accordingly. The result of the evaluation is the value of the right-hand side.

$$\text{E-Assign}\frac{\langle H_0, L, \mathtt{e}_1\rangle \Downarrow \langle H_1, (l, x)\rangle \qquad \langle H_1, L, \mathtt{e}_2\rangle \Downarrow_v \langle H_2, v\rangle}{\langle H_0, L, \mathtt{e}_1 = \mathtt{e}_2\rangle \Downarrow \langle H_2[(l, x) \mapsto v], v\rangle}$$

E-Update In E-Update, we evaluate a call to `assign`, which is used to update or add multiple properties. Recall that `assign` is a functional update: its first argument does not get modified. Therefore, E-Update first duplicates the first argument of `assign`, using the auxiliary function *clone*. The internal property `@this` of the clone refers to the clone instead of the original object. Next, every property of the second argument is evaluated to its value. The cloned object then replaces or adds every evaluated property to the cloned object. Finally, E-Update returns the location of the cloned object.

$$\text{E-Update}\frac{\begin{array}{c} \langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H_0', l\rangle \qquad H_1 = H_0' * [l_r \mapsto clone(H_0'(l), l_r)] \\ \langle H_1, L, \mathtt{e}_1\rangle \Downarrow_v \langle H_1', v_1\rangle \qquad H_2 = H_1'[(l_r, \mathtt{n}_1) \mapsto v_1] \\ \cdots \\ \langle H_m, L, \mathtt{e}_m\rangle \Downarrow_v \langle H_m', v_m\rangle \qquad H = H_m'[(l_r, \mathtt{n}_m) \mapsto v_m] \end{array}}{\langle H_0, L, \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\})\rangle \Downarrow \langle H, l_r\rangle}$$

The definition of *clone* is as follows. The $\uplus$ operator takes the union of two object maps, preferring fields from the right operand where the domains of the two inputs overlap.

$$clone(o, l_r) = o \uplus (\texttt{@this} \mapsto l_r)$$

E-Prop Evaluating a property access is less complicated than its type checking counterpart: as types (and thus inter-property constraints) are omitted from the evaluation rules, E-Prop verifies that the object does not dereference null. It returns a pair containing the location of the object and the accessed property. Note that this evaluation rule also covers property accesses of undefined properties: in this case, γ will return `undefined`.

$$\text{E-Prop}\frac{\langle H_0, L, \texttt{e}\rangle \Downarrow_v \langle H_1, l\rangle \quad l \neq \texttt{null}}{\langle H_0, L, \texttt{e.n}\rangle \Downarrow \langle H_1, (l, \texttt{n})\rangle}$$

E-Prop' The rule E-Prop' covers the accessing of properties of literals. As literals do not have properties themselves, E-Prop' uses the auxiliary function *box* to construct an object that corresponds to the literal. E-Prop' then returns the location of the boxed literal, together with the property that is being accessed.

$$\text{E-Prop'}\frac{\langle H_0, L, \texttt{e}\rangle \Downarrow_v \langle H_1, \texttt{l}\rangle \quad H_2 = H_1 * [l_{boxed} \mapsto box(\texttt{l}, l_{boxed})]}{\langle H_0, L, \texttt{e.n}\rangle \Downarrow \langle H_2, (l_{boxed}, \texttt{n})\rangle}$$

We assume that for every kind of literal (string, number, boolean,...) there exists a corresponding prototype object o_{proto} that contains its prototype functions (i.e. `toString` and `valueOf`). Furthermore, we assume that each prototype function expects a literal value in `this.value`. We now define the *box* function as follows. *box* constructs an object that contains both the prototype functions alongside the literal value in `this.value`.

$$box(\texttt{l}, l_{boxed}) = clone(o_{proto}, l_{boxed}) \uplus (value \mapsto \texttt{l})$$

E-Call and E-CallUndef Evaluating a function or method call first requires the evaluation of the callee. The location of the callee must contain a lambda expression. Next, the auxiliary function *This* is used to retrieve the location of the internal variable `@this`. Its definition is as follows. If the scope object of the reference has an `@this` property, the scope object itself is returned. Otherwise, *This* returns the global scope object.

$$\text{E-Call}\ \frac{\begin{array}{cc}\langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & \gamma(H_1, r) = l_1 \\ H_1(l_1) = \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L_1\rangle & \mathit{This}(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_v \langle H_2, v_1\rangle \quad \ldots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_v \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * \mathit{act}(l, \overline{\mathtt{x}}, \overline{v}, l_2) & \langle H', l : L_1, \overline{\mathtt{s}}\rangle \Downarrow \langle H'', \mathtt{return}\ v;\rangle\end{array}}{\langle H_0, L_0, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\rangle \Downarrow \langle H'', v\rangle}$$

$$\text{E-CallUndef}\ \frac{\begin{array}{cc}\langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & \gamma(H_1, r) = l_1 \\ H(l_1) = \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L_1\rangle & \mathit{This}(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_v \langle H_2, v_1\rangle \quad \ldots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_v \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * \mathit{act}(l, \overline{\mathtt{x}}, \overline{v}, l_2) & \langle H', l : L_1, \overline{\mathtt{s}}\rangle \Downarrow \langle H'', \mathtt{return};\rangle\end{array}}{\langle H_0, L_0, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\rangle \Downarrow \langle H'', \mathtt{undefined}\rangle}$$

$$\begin{array}{rl} \mathit{This}(H, (l, x)) = l & \text{if } H(l, \mathtt{@this})\downarrow \\ \mathit{This}(H, v) = l_g & \text{otherwise}\end{array}$$

In JavaScript, the evaluation of `o.m(...)` is done in two phases: first `o.m` is evaluated, and this function is subsequently applied with `o` bound to `@this`, and the arguments bound to their values. To model this, E-Call inspects the object from which the function was selected, and if it contains `@this`, the object is bound to `@this` for the duration of the function call.

Next, all arguments of the call are evaluated to a value[4]. Using the auxiliary function *act* (defined below), the heap is extended with a new location that points to a new scope object. This scope object contains the arguments of the function, mapped onto their values. The scope object also contains a `@this` variable that points to the result of the call to *This*.

$$\mathit{act}(l, \overline{\mathtt{x}}, \overline{v}, l') = l \mapsto (\{\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{@this} \mapsto l'\})$$

Finally, the body of the lambda expression is evaluated against a scope chain that is extended with the new scope chain object. This must result in either a value `return` statement or an empty `return` statement. In the former case, the function or method call is evaluated to that value. In the latter case, the call evaluates to `undefined`.

Note that we do not create bindings for all local variables in the body of the lambda expression up front: they are added to the local scope as they are declared and initialised. This correctly emulates the concept of block-scoped variable declarations (using `let`) in TIPC.

[4]Recall that a value can either be a literal `l` or a location l. The latter ensures pass-by-reference in case of object parameters.

E-Func The evaluation rule for function definitions first creates the corresponding lambda expression. Next, the heap is extended with a new location that points to that lambda expression. Finally, E-Func returns the new location.

$$\text{E-Func}\frac{H_1 = H_0 * [l \mapsto \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L\rangle]}{\langle H_0, L, \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\}\rangle \Downarrow \langle H_1, l\rangle}$$

E-TypeAssert The evaluation rule for a type cast discards the cast itself and just evaluates the expression. This is safe thanks to the checks performed by the type system. Note that the casting of object literals to interface types have to evaluated differently. This is covered in the following evaluation rule.

$$\text{E-TypeAssert}\frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow \langle H_1, r_1\rangle}{\langle H_0, L, \mathtt{<T>e}\rangle \Downarrow \langle H_1, r_1\rangle}$$

E-TypeAssertInf This rule evaluates the cast of an object literal to an interface type. Similar to E-TypeAssert, this rule first evaluates the object literal expression. Then, the internal property `@interface` is added to the evaluated object literal. This internal property indicates that the expression is of interface type `I`. In the next section, this property is used for linking the run-time interface in a location to the declared type in the program; this serves to simplify the soundness proofs.

$$\text{E-TypeAssertInf}\frac{\begin{array}{c}\langle H_0, L, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}\rangle \Downarrow_v \langle H_1, l\rangle \\ H = H_1[(l, \mathtt{@interface}) \mapsto \mathtt{I}]\end{array}}{\langle H_0, L, \mathtt{<I>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}\rangle \Downarrow \langle H, l\rangle}$$

6.4.2 Evaluating Statement Sequences

The evaluation relation for statement sequences is written as $\langle H_1, L, \overline{\mathtt{s}}_1\rangle \Downarrow \langle H_2, s\rangle$, where s is a statement result (i.e. either a return without value `return;`, a return with value `return` v`;` or no return `;`). The latter can occur at the top level of the program, or when evaluating the consequent and alternative of an `if` statement.

The following paragraphs show the evaluation rules for sequences[5]. Figure 6.13 lists all sequence evaluation rules at the end of this chapter.

[5]The evaluation rules for sequences are omitted in the definition of safeFTS [Bierman et al., 2014], but we defined no rules specific to TIPC interface definitions.

E-EmptySeq Evaluating the empty sequence results in the terminal semicolon.

$$\text{E-EmptySeq}\ \frac{}{\langle H, L, \bullet \rangle \Downarrow \langle H, ; \rangle}$$

E-Return The evaluation rule for a sequence that starts with an empty `return` statement discards the statements after the return.

$$\text{E-Return}\ \frac{}{\langle H, L, \texttt{return}; \overline{\texttt{s}} \rangle \Downarrow \langle H, \texttt{return}; \rangle}$$

E-ReturnVal The rule E-ReturnVal evaluates a sequence that starts with a value `return` statement. It evaluates the expression to a value, and discards the remaining statements after the return.

$$\text{E-ReturnVal}\ \frac{\langle H, L, \texttt{e} \rangle \Downarrow_v \langle H_1, v \rangle}{\langle H, L, \texttt{return e}; \overline{\texttt{s}} \rangle \Downarrow \langle H_1, \texttt{return } v; \rangle}$$

E-ExpSt Rule E-ExpSt evaluates the sequence of statements that starts with an expression that does not contain a `return` statement. In this case, this expression is evaluated, but its result is not taken into account for the result of this evaluation rule. Instead, this rule evaluates to the statement result of the evaluation of the rest of the sequence.

$$\text{E-ExpSt}\ \frac{\langle H, L, \texttt{e} \rangle \Downarrow \langle H_1, r \rangle \quad \langle H_1, L, \overline{\texttt{s}} \rangle \Downarrow \langle H_2, s \rangle}{\langle H, L, \texttt{e}; \overline{\texttt{s}} \rangle \Downarrow \langle H_2, s \rangle}$$

E-IfTrue and E-IfFalse There are two rules that cover the evaluation of `if` statements: rule E-IfTrue covers the case that the condition evaluates to `true`, E-IfFalse covers the case that the condition evaluates to `false`. After evaluating the condition, E-IfTrue evaluates the true branch while E-IfFalse evaluates the false branch. Note that, because of block scoping in TIPC, the branches of `if` statements introduce a new scope, so variables declared there are not visible outside. The result of that evaluation is concatenated with the rest of the statements ($s; \overline{\texttt{s}}$). That concatenation is evaluated against the original scope chain. This short-circuits evaluation in case the branch contained a `return` statement.

Contrary to the typing rules for `if` statements, there is no separate evaluation rules for `if` statements that verify the presence of properties, as they require no different evaluation strategy.

$$\text{E-IfTrue}\ \frac{\begin{array}{c}\langle H, L, \mathtt{e}\rangle \Downarrow_v \langle H_1, \mathtt{true}\rangle \\ H_2 = H_1 * [l \mapsto ()] \\ \langle H_2, l : L, \overline{\mathtt{t}}_1\rangle \Downarrow \langle H_3, s\rangle \\ \langle H_3, L, s\,;\overline{\mathtt{s}}\rangle \Downarrow \langle H_4, s_r\rangle\end{array}}{\langle H, L, \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}}\rangle \Downarrow \langle H_4, s_r\rangle}$$

$$\text{E-IfFalse}\ \frac{\begin{array}{c}\langle H, L, \mathtt{e}\rangle \Downarrow_v \langle H_1, \mathtt{false}\rangle \\ H_2 = H_1 * [l \mapsto ()] \\ \langle H_2, l : L, \overline{\mathtt{t}}_2\rangle \Downarrow \langle H_3, s\rangle \\ \langle H_3, L, s\,;\overline{\mathtt{s}}\rangle \Downarrow \langle H_4, s_r\rangle\end{array}}{\langle H, L, \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}}\rangle \Downarrow \langle H_4, s_r\rangle}$$

E-ITVarDec This rule first evaluates a sequence that starts with a variable declaration. First, the rule evaluates the value for the expression. Next, the scope chain is extended with a new scope object in which the new variable is mapped onto the evaluated expression. The rest of the sequence is evaluated against the extended scope chain.

$$\text{E-ITVarDec}\ \frac{\begin{array}{c}\langle H, L, \mathtt{e}\rangle \Downarrow_v \langle H_1, v\rangle \\ H_2 = H_1 * [l \mapsto (\{\mathtt{x} \mapsto v\})] \\ \langle H_2, l : L, \overline{\mathtt{s}}\rangle \Downarrow \langle H_3, s\rangle\end{array}}{\langle H, L, \mathtt{let}\ \mathtt{x} : \mathtt{S} = \mathtt{e}; \overline{\mathtt{s}}\rangle \Downarrow \langle H_3, s\rangle}$$

6.5 Soundness

So far, this chapter described the formalisation of the TIPC programming language. The novelty of the type system in TIPC lies in its guarantee that *all* constraints imposed on objects are guaranteed to be satisfied throughout the execution of the program, including those over multiple properties.

In this section, we prove the soundness of the type system of TIPC. The key element of this proof is the type safe usage of objects with inter-property constraints.

The key element to prove for the preservation of TIPC is that the evaluation rules and typing rules of TIPC ensure that objects on the heap tagged with an

interface property satisfy the interface constraints. This property is captured in Lemma 1:

Lemma 1 (Correctness of interface types at runtime)**.** For heap locations tagged as interface types, i.e. those where $\Sigma(l) = \mathtt{I}$ and $\Sigma \models H$, the following is required:

1. Every interface instance is tagged as such:
   $H(l, \mathtt{@interface}) = \mathtt{I}' \wedge \mathtt{I}' \leqq \mathtt{I}$;

2. All properties are correctly typed:
   $\forall \mathtt{n} \in \mathit{fields}(l) : \mathtt{n{:}T} \in \mathit{properties}(\mathtt{I}') \wedge \Sigma \models \langle H, (l, \mathtt{n})\rangle : \mathtt{T}$;

3. The constraints are satisfied by a valuation over the presence or absence of properties: $v = c_p \cup c_{np}$ and $\hat{v}(\mathit{constraints}(\mathtt{I}'))$

$$\begin{aligned} \text{where } c_p &= \{\mathtt{present(n)} \mid \mathtt{n} \in \mathit{fields}(l)\} \\ \text{where } c_{np} &= \{\neg\mathtt{present(n)} \,|\, \mathtt{n} \in \mathit{properties}(\mathtt{I}') \\ &\qquad \wedge (\neg H(l,\mathtt{n})\downarrow \vee\ H(l,\mathtt{n}) = \mathtt{undefined})\} \\ \text{where } \mathit{fields}(l) &= \{\mathtt{n} \mid H(l,\mathtt{n}) \downarrow \wedge H(l,\mathtt{n}) \neq \mathtt{undefined}\}. \end{aligned}$$

The theorems for subject reduction and the corresponding judgments are based on those of safeFTS. The structure for our proof is in the style of Abadi and Cardelli [1996] and Bierman et al. [2003]. We introduce some new judgments in Section 6.5.1 and we provide the proof for the key property (Lemma 1. Section 6.5.3 contains the full proofs for the preservation of types in expressions and statements (Section 6.5.3):

Theorem 1 (Type Preservation for Expressions)**.** If $\Sigma \models \langle H, L, \mathtt{e}\rangle : \mathtt{T}$ and $\langle H, L, \mathtt{e}\rangle \Downarrow \langle H', r\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', r\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \mathtt{T}$.

Theorem 2 (Type Preservation for Statements)**.** If $\Sigma \models \langle H, L, \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}}$ and $\langle H, L, \overline{\mathtt{s}}\rangle \Downarrow \langle H', s\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', s\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$.

Because the operational semantics presented in Section 6.4 is written down in big-step notation, we cannot prove progress [Pierce, 2002, page 505]. The common alternative is to provide explicit stuck states [Abadi and Cardelli, 1996] and prove that these cannot be entered. We follow this reasoning and inspect each evaluation rule for potential stuck states. For each stuck state, we show how the type system prevents those (Section 6.5.4).

6.5.1 Judgments

Before introducing the proofs, we first introduce a relationship between types and the operational semantics. We define a heap type Σ as a partial function from heap locations to types (either function types, object literal types, or interface types). A heap type is a subset of another heap type ($\Sigma \subseteq \Sigma'$) when $\operatorname{dom}\Sigma \subseteq \operatorname{dom}\Sigma'$ and $\forall l \in \operatorname{dom}\Sigma : \Sigma(l) = \Sigma'(l)$.

We also define several judgments.

Definition 1 (Well-formed Heap)**.** A heap is well-formed when:

- for all closures in the codomain of the heap: the scope chain of the closure must be compatible with the heap: $H, L \models \diamond$;
- for all object maps in the codomain of the heap: every location l in the codomain of the object map has to be in the domain of the heap ($l \in \operatorname{dom} H$).

It is written down as $H \models \diamond$.

Definition 2 (Heap–Scope Chain Compatibility)**.** A judgment that a heap H and scope chain L are compatible is written as $H, L \models \diamond$. This judgment requires that all scope objects l in the scope chain L exist on the heap:

$$\text{HSC}\frac{H \models \diamond \quad \forall l \in L : l \in \operatorname{dom} H \wedge H(l) \text{ is an object map}}{H, L \models \diamond}$$

Definition 3 (Heap–Heap Type Compatibility)**.** We use a judgment $\Sigma \models H$ to denote that the heap H is compatible with the heap type Σ. Intuitively, a heap type is compatible with a heap when every value in the heap has the type predicted by the heap typing. This compatibility also requires that the constraints of interface types are satisfied.

$$\text{HHC}\frac{\operatorname{dom}\Sigma = \operatorname{dom} H \quad H \models \diamond \quad \forall l \in \operatorname{dom}\Sigma : \Sigma, H \models l \texttt{ ok}}{\Sigma \models H}$$

The third part of the HHC judgment ($\Sigma, H \models l$ `ok`) actually verifies the compatibility between the heap and the heap type, and is defined in Figure 6.7. A location in the heap points to a closure or an object map. For a closure, the compatibility verifies that the parameter list of the closure (in the heap) and the type (in the heap type) match. Moreover, the closure body must have a return type that is assignable to the defined return type (Figure 6.7, Closure). This is verified using *context*, which is defined later in this section (Definition 4).

$$\textsc{Closure}\ \frac{\Sigma(l) = \{(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{S}\} \quad H(l) = \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L\rangle \quad \mathit{context}(\Sigma, L), \mathtt{this} : \mathtt{any}, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{s}} : \overline{\mathtt{S}}' \quad \overline{\mathtt{S}}' \leqq \mathtt{S}}{\Sigma, H \models l\ \mathtt{ok}}$$

$$\textsc{Object Map (Literal)}\ \frac{\Sigma(l) = \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} \quad H(l) = \text{one of} \begin{cases} \{\overline{\mathtt{n}} \mapsto \overline{v}\} \\ \{\mathtt{@this} \mapsto l', \overline{\mathtt{n}} \mapsto \overline{v}\} \end{cases} \quad \Sigma, H \models \overline{v} : \overline{\mathtt{T}}\ \mathtt{ok}}{\Sigma, H \models l\ \mathtt{ok}}$$

$$\textsc{Object Map (Intf.)}\ \frac{\begin{array}{c} \Sigma(l) = \mathtt{I} \quad H(l) = \{\mathtt{@this} \mapsto l, \mathtt{@interface} \mapsto \mathtt{I}', \overline{\mathtt{n}} \mapsto \overline{v}\} \\ \mathtt{I}' \leqq \mathtt{I} \quad \mathit{properties}(\mathtt{I}') = \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} \\ \Sigma, H \models \overline{v} : \overline{\mathtt{T}}\ \mathtt{ok} \\ \{\overline{\mathtt{M}_p}\} = \{\mathtt{n} : \mathtt{T} \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} \wedge H(l, \mathtt{n})\downarrow\} \\ \{\overline{\mathtt{M}_p}\} \subseteq \mathit{properties}(\mathtt{I}') \\ \{\overline{\mathtt{M}_{np}}\} = \mathit{properties}(\mathtt{I}') \setminus \{\overline{\mathtt{M}_p}\} \\ c_p = \{\mathtt{present(n)} \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_p}\}\} \\ c_{np} = \{\neg\mathtt{present(n)} \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_{np}}\}\} \\ v = c_p \cup c_{np} \qquad \hat{v}(\mathit{constraints}(\mathtt{I}')) \end{array}}{\Sigma, H \models l\ \mathtt{ok}}$$

Figure 6.7: Definition of $\Sigma, H \models l\ \mathtt{ok}$ for heap objects

When a location points to an object map, the object map can either represent an object literal type or an interface type. This distinction is made by the presence or absence of the internal property `@interface`. For object literals, the rule verifies that the type of each property is compatible with its value (Figure 6.7, Object Map (Literal)). Finally, for interfaces the values of the properties have to have a valid typing given the heap and heap typing. Next, the rule verifies whether the properties in the interface instance satisfy the interface constraints (Figure 6.7, Object Map (Interface)).

Figure 6.8 defines heap-heap type compatibility for values (which is either a literal or a location), given a type. Verifying a literal is straightforward reusing the typing judgment, and verifying the location reuses the heap compatibility defined in Figure 6.7.

Note that we do not elaborate further on circular references, i.e. when the object map of one location refers to another location that in turn refers to the first location. As in Bierman et al. [2014], circular references can be covered by

$$\text{Literal}\frac{\vdash \mathtt{l} : \mathtt{T}' \quad \mathtt{T}' \leqq \mathtt{T}}{\Sigma, H \models \mathtt{l} : \mathtt{T}\ \mathtt{ok}} \qquad \text{Location}\frac{\Sigma, H \models l\ \mathtt{ok} \quad \Sigma(l) = \mathtt{T}' \quad \mathtt{T}' \leqq \mathtt{T}}{\Sigma, H \models l : \mathtt{T}\ \mathtt{ok}}$$

Figure 6.8: Definition of $\Sigma, H \models v : \mathtt{T}\ \mathtt{ok}$ for values

coinductive proof techniques.

Definition 4 (Context)**.** The function $\mathit{context}(\Sigma, L)$ builds a typing judgment describing the variables in the scope chain L, using the types in Σ. The $\uplus$ operator ensures that only the inner-most type for a variable is used: if a variable is present on both sides, the right-side instance is returned. Because E-TypeAssertInf attaches an `@interface` label to all interface variables in the heap, Σ predicts interface types as well as function types and object literal types.

$$\begin{aligned} \mathit{context}(\Sigma, []) &= \{\} \\ \mathit{context}(\Sigma, l : L) &= \mathit{context}(\Sigma, L) \uplus \{\Theta(x : \mathtt{T}) \mid x : \mathtt{T} \in \Sigma(l)\} \end{aligned}$$

The function Θ transforms the internal property `@this` to a program variable `this`. This is necessary to correctly construct an equivalent to the original typing environment.

$$\Theta(x : \mathtt{T}) = \begin{cases} \mathtt{this} : \mathtt{T} & \text{if } x = \mathtt{@this} \\ x : \mathtt{T} & \text{if } x \neq \mathtt{@this} \end{cases}$$

Next, we define several judgments that combine the previous definitions:

$$\frac{\Sigma \models H \quad H, L \models \diamond \quad \mathit{context}(\Sigma, L) \vdash \mathtt{e} : \mathtt{T}}{\Sigma \models \langle H, L, \mathtt{e} \rangle : \mathtt{T}}$$

We define an analogous judgment for statements:

$$\frac{\Sigma \models H \quad H, L \models \diamond \quad \mathit{context}(\Sigma, L) \vdash \overline{\mathtt{s}} : \overline{\mathtt{T}}}{\Sigma \models \langle H, L, \overline{\mathtt{s}} \rangle : \overline{\mathtt{T}}}$$

Finally, we add a judgment on the result of the evaluation of expressions:
$\Sigma \models \langle H, r \rangle : \mathtt{T}$. A result r is either a reference (a combination of a location l and a variable x), or a value (which is either a literal `l` or a location l). The following three judgments cover these three cases. The type of a location is looked up in the

heap typing (Σ), while the type of a literal is known using a type judgment. For a type of a reference, the location forms the environment for the type judgment of the variable.

$$\frac{\Sigma \models H \quad \Sigma(l) = \mathtt{T}}{\Sigma \models \langle H, l\rangle : \mathtt{T}} \qquad \frac{\Sigma \models H \quad \vdash \mathtt{l} : \mathtt{T}}{\Sigma \models \langle H, \mathtt{l}\rangle : \mathtt{T}} \qquad \frac{\Sigma \models H \quad \Sigma(l) \vdash x : \mathtt{T}}{\Sigma \models \langle H, (l, x)\rangle : \mathtt{T}}$$

Correspondingly, we also introduce judgments for statement results. The empty sequence produces an empty set of types ($\bullet$), a default return has type `void` and the statement form `return e;` has the type of `e`.

$$\frac{\Sigma \models H}{\Sigma \models \langle H, \mathtt{;}\rangle : \bullet} \qquad \frac{\Sigma \models H}{\Sigma \models \langle H, \mathtt{return;}\rangle : \mathtt{void}}$$

$$\frac{\Sigma \models H \quad \Sigma \models \langle H, v\rangle : \mathtt{T}}{\Sigma \models \langle H, \mathtt{return}\ v\mathtt{;}\rangle : \mathtt{T}}$$

6.5.2 Key Properties

In this section, we summarise the parts of the preservation proofs that cover inter-property constraints. In Section 6.5.1, we have extended the definition of heap–heap type compatibility to take interfaces into account as well (Figure 6.7). Lemma 1 shows how the evaluation and typing rules ensure that object on the heap tagged with an interface property satisfy the interface constraints. Corollary 1 gives an informal overview of how the type system accurately predicts the presence or absence of interface instance properties at runtime.

Lemma 1 (Correctness of interface types at runtime)**.** For heap locations tagged as interface types, i.e. those where $\Sigma(l) = \mathtt{I}$ and $\Sigma \models H$, the following is required:

1. Every interface instance is tagged as such:
   $H(l, \mathtt{@interface}) = \mathtt{I}' \wedge \mathtt{I}' \leqq \mathtt{I}$;
2. All properties are correctly typed:
   $\forall \mathtt{n} \in \mathit{fields}(l) : \mathtt{n{:}T} \in \mathit{properties}(\mathtt{I}') \wedge \Sigma \models \langle H, (l, \mathtt{n})\rangle : \mathtt{T}$;
3. The constraints are satisfied by a valuation over the presence or absence of properties: $v = c_p \cup c_{np}$ and $\hat{v}(\mathit{constraints}(\mathtt{I}'))$

$$\text{where } c_p = \{\mathtt{present(n)} \mid \mathtt{n} \in \mathit{fields}(l)\}$$
$$\text{where } c_{np} = \{\neg\mathtt{present(n)} \mid \mathtt{n} \in \mathit{properties}(\mathtt{I}') \wedge (\neg H(l, \mathtt{n})\downarrow \vee\ H(l, \mathtt{n}) = \mathtt{undefined})\}$$
$$\text{where } \mathit{fields}(l) = \{\mathtt{n} \mid H(l, \mathtt{n}) \downarrow \wedge H(l, \mathtt{n}) \neq \mathtt{undefined}\}.$$

Proof. By induction on the evaluation rules for expressions and statements. Most rules do not directly modify the heap, so we only focus on the rules that potentially invalidate this condition.

Note that this lemma is not only unaffected by explicit property presence tests, the lemma guarantees this because of requirement 3. Assuming that an object of interface type `I` is well-formed before the presence test, then the strengthened interface type `I′` in the taken branch represents the state of the runtime object more accurately.

E-TypeAssertInf This evaluation rule is responsible for instantiating interface types on the heap, given an object literal. Requirement 1 follows from the evaluation rule. Requirements 2 and 3 follow directly from the type system.

E-Assign There are three sub-cases: $\texttt{e}_1$ can either resolve to a variable reference, an object property, or an interface property:

- In case of a variable reference to an interface `I`, the three properties follow directly from assignment compatibility between `I` and the interface type `I'` assigned to $\texttt{e}_2$.
- In case of a property belonging to an object: the three requirements cannot be invalidated.
- In case of an interface property: it depends on whether this expression is trying to add a new property or update a present property. The type system assigns type `Undefined` to properties which are guaranteed to be absent, and rejects programs that access properties whose presence is unknown.

  For property update, the syntax prevents accesses to the `@interface` property (preserving requirement 1). Requirements 2 and 3 are guaranteed by assignment compatibility.

E-Update This rule first clones the source object (for which all properties are already satisfied) before assigning the new fields. Requirement 1 follows from the evaluation rule: the `@interface` tag is cloned along with other fields. We now consider the generated interface `I′` in I-UpdateInf. *slice* ensures that the interface contains the smallest possible subset of constraints and properties such that all constraints in `I` either do not mention any properties from `I′` or are part of the constraints in `I′`. For the fields in `I′`, requirements 2 and 3 are guaranteed by the I-UpdateInf rule. For fields *not* in `I′`, requirements 2 and 3 continue to hold, as they cannot be affected by the `assign` operation by definition.

E-ObLit This rule creates a new object on the heap. This cannot invalidate existing interface instances on the heap.

E-Prop', E-Func These rules create a heap location for respectively properties of literal objects and a closure, but neither alter existing interface instances on the heap.
E-Call, E-CallUndef The heap modifications made by these two rules are limited to evaluation of sub-expressions or the allocation of a new scope object to hold the new function's local variables. In the latter case, we rely on the fact that extension cannot affect existing interface instances on the heap.
E-ReturnVal, E-ExpSt The heap modifications made by these statement evaluation rules are limited to evaluating sub-expressions and sub-statements, hence they cannot affect existing interface instances on the heap.
E-IfTrue, E-IfFalse The modifications to the heap made by these rules are evaluating sub-expressions and sub-statements. These rules also extend the heap with a new empty object map, which does not affect existing objects on the heap.
E-ITVarDec Next to the heap modifications that stem from evaluating a sub-expression and sub-statements, this rule extends the heap with a new object map in which the declared variable points to the evaluated sub-expression. Thus, this evaluation rule does not affect existing interface instances. Moreover, the typing rule for variable declarations (I-ITVarDec) prohibits the creation of variables of an interface type from object literals. Therefore, E-ITVarDec does not introduce new interface instances on the heap. It is possible to assign an interface instance to a variable of an interface type. However, the assignment compatibility rules ensure that the source interface (to which the `@interface` tag points) is assignable to the target interface. This suffices to ensure all three requirements of this lemma. □

Corollary 1 (Constraint–presence correlation)**.** The type system of TIPC guarantees that if the constraints of an interface contain a constraint `present(n)`, it is certain that the property `n` is present at runtime in objects with that interface type. Similarly: if there is a constraint ¬`present(n)` , it is certain that the property `n` will not be present.

There are three cases to consider:

Case 1: *Construction* Interfaces can only be constructed in three ways, which all ensure that the correlation holds:

Case 1a: I-AssertInf. When an object literal is cast to an interface, the interface constraints are verified against the properties in the object literal. The correlation is thus informed by the exact properties of the runtime object (E-TypeAssertInf) and enforced by the type system.

Case 1b: I-Assign. When an instance of interface $\texttt{I}_0$ is assigned to a variable of type interface $\texttt{I}_1$, the type system requires that the constraints are satisfied via the assignment compatibility rule A-Interface. The correlation holds for the source object (of type $\texttt{I}_0$) and the compatibility rule asserts that the properties of $\texttt{I}_1$ must be respectively present or absent. Therefore, the correlation must hold after the cast as well. At runtime, nothing changes.

Case 1c: I-Assert. Analogous to Case 1b: assignment compatibility dictates the presence and absence of properties in the source object. Nothing changes at runtime.

Case 2: *Property assignment* The assignment of new values to object properties either happens on a per-property basis (Case 2a), or multiple properties at once using `assign` (Case 2b).

Case 2a: I-Assign. When a new value is assigned to a property `n` of an interface, two typing rules are relevant: I-Prop (including the *lookup* function) and I-Assign. At runtime, the E-Assign rule simply overwrites the object property, so it is up to the type system to enforce the correlation. We assume the correlation holds before the assignment, so the constraints of the interface must state one of the following:

`present(n)`: the *lookup* function of I-Prop returns the type of `n` and I-Assign then allows the assignment of another value (following the typing rules). As this will only update the value of a property that is already present, this does not change the presence of `n` in the object, thus the correlation holds.

¬`present(n)`: the *lookup* function of I-Prop returns type `Undefined`. The assignment compatibility required by I-Assign will fail as no type is assignable to `Undefined`, except for `undefined`, in which case the property will remain absent. Again, the correlation holds.

Neither: the *lookup* function of I-Prop is not defined in this case, so the program does not typecheck. Without this safety guard in place, the correlation would not hold.

Case 2b: I-Update. The `assign` function updates multiple properties of an object. Again, we assume that the correlation holds before the assignment. The `assign` function returns a new object, of the same type as the first argument, in which the properties of the second argument are updated. Properties can become absent or present (by resp. assigning `undefined` or a value different from `undefined`), or simply change

value. The assignment is only accepted by the type checker if the second argument of `assign` is assignable to the generated interface which covers its properties. Therefore, a change in presence for those properties is only allowed if the input interface did not already require their presence or absence. At runtime, rule E-Update first clones the object and then the properties are overwritten by those of the second argument. The correlation holds for both the generated interface (because of assignment compatibility and isolation) and the rest of the object.

Case 3: *After a presence test* In case of an `if` statement that tests the presence of an interface property, the newly gained information is added to the constraints of the type in both branches (function *addConstraint* in I-IfPresenceInterface). Here the property follows from the program flow: if the field presence test succeeds the type system can only conclude that the `present` constraint applies, and vice versa when the presence test fails. Outside of the `if` statement, the `present` constraint is discarded again. Even though the runtime value does not change, this is again an example of the properties of the runtime value informing the type system and thus the correlation.

6.5.3 Preservation

We recall the theorem for the preservation of types while evaluating expressions and statements:

Theorem 1 (Type Preservation for Expressions)**.** If $\Sigma \models \langle H, L, \mathtt{e}\rangle : \mathtt{T}$ and $\langle H, L, \mathtt{e}\rangle \Downarrow \langle H', r\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', r\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \mathtt{T}$.

Theorem 2 (Type Preservation for Statements)**.** If $\Sigma \models \langle H, L, \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}}$ and $\langle H, L, \overline{\mathtt{s}}\rangle \Downarrow \langle H', s\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', s\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$.

For the preservation of a sequence of statements, the type of the evaluated expression has to be assignable to the union type of a sequence of types (written $\cup(\overline{\mathtt{T}})$). As union types are not part of TIPC, the assignment compatibility rule needs to be extended:

$$\text{A-Union}\ \frac{\forall\ \mathtt{T}' \in \overline{\mathtt{R}} : \mathtt{T} \leqq \mathtt{T}'}{\mathtt{T} \leqq \cup(\overline{\mathtt{R}})}$$

The proofs for the preservation of types while evaluating expressions and statements can be found in Appendix B.1 resp. Appendix B.2. Preservation is proved by a case analysis on evaluation rules.

6.5.4 Progress

Section 6.4 presented the big-step operational semantics of TIPC. Because of the big-step notation, it is impossible to prove progress for TIPC: proving progress for TIPC would be equal to proving that every well-typed term evaluates to a value. However, this is not feasible: as TIPC supports general recursion, this would require us to prove that every program will terminate.

Instead, this section lists the possible "stuck" states in TIPC where one or more evaluation rules syntactically match the expression to be evaluated but none of the rules' preconditions apply. For each of these states, we show that the type system prevents these situations from occurring.

Looking up an unknown identifier (E-Id) E-Id will fail when a non-existent identifier is looked up in the environment. In a well-typed program, I-Id will prevent this.

Looking up `this` outside of method (E-this) E-this fails when `@this` is not defined in any object of the scope chain. Again, I-Id will prevent this.

Wrong operands or operator (E-Op) E-Op fails when one of the operands does not evaluate to a literal or when there is no operator for the combination of literals. However, this is prevented by I-Op, which will only succeed when the combination of the two literals and the binary operator is accepted by the type system.

The left-hand side of an assignment is not a reference (E-Assign) This is prevented by the syntax of TIPC, which requires that the left-hand side of an assignment is either a variable or a property. E-Id and E-Prop both evaluate to a reference. Moreover, I-Id guarantees that the variable is known and I-Prop guarantees that the property is part of the object.

First argument of `assign` call is not an object (E-Update) More specifically, the evaluation rule for `e` requires that `e` does not evaluate to a literal but to a location, and the evaluation rules for properties requires that the location points to an object. This is guaranteed by I-UpdateObj and I-UpdateInf.

Object of property access evaluates to `null` (E-Prop) This is prevented by *lookup* in I-Prop which will not succeed, as `null` does not have any properties.

Callee does not evaluate to a function location, number of arguments does not match declared number and function body does not evaluate to a `return` statement (E-Call and E-CallUndef) The first two stuck states are prevented by I-Call. The third stuck state is prevented by the syntax of TIPC, which requires that every execution path in a function body contains at least one `return` statement.

The if condition does not evaluate to `true` or `false` (E-IfTrue and E-IfFalse) This is guaranteed by I-IfGeneral and I-IfPresenceInterface.

6.6 Conclusion

In this chapter, we have introduced the formalisation of TIPC, a variant of the object-oriented programming language TypeScript that adds interfaces with inter-property constraints. More specifically, incorporating inter-property constraints into a regular programming language had an impact on every aspect of the formalisation:

Syntax On a syntactical level, changes were kept to a minimum. Interfaces are defined in an unconventional way where the constraints on the presence are moved to a separate section. This allows the definition of presence constraints between properties. A second addition is the functional object update called `assign` which enables the safe update of properties that are part of inter-property constraints.

Typing rules The type system of TIPC has to guarantee that inter-property constraints are satisfied at initialisation time and remain satisfied throughout the program. The biggest impact of inter-property constraints is on the functionality that checks property accesses, verifying whether assignments are type-safe and type checking the simultaneous update of properties.

Operational semantics As inter-property constraints are an extension of the TIPC's types, the impact on the operational semantics of TIPC was minimal. The only evaluation rule that was impacted was the evaluation of a type cast of an object literal. This rule adds an interface tag in the runtime object.

Finally, we proved that the formalisations are sound, including that objects with inter-property constraints imposed on them will never contain an invalid combination of constraints at runtime.

The next chapter presents the implementation of TIPC.

$$\text{I-Id}\ \frac{}{\Gamma, \mathtt{x{:}T} \vdash \mathtt{x{:}T}} \qquad \text{I-Number}\ \frac{}{\Gamma \vdash n : \mathtt{number}} \qquad \text{I-String}\ \frac{}{\Gamma \vdash s : \mathtt{string}}$$

$$\text{I-Bool}\ \frac{}{\Gamma \vdash \mathtt{true}, \mathtt{false} : \mathtt{boolean}} \qquad \text{I-Null}\ \frac{}{\Gamma \vdash \mathtt{null} : \mathtt{Null}}$$

$$\text{I-Undefined}\ \frac{}{\Gamma \vdash \mathtt{undefined} : \mathtt{Undefined}} \qquad \text{I-ObLit}\ \frac{\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

$$\text{I-Op}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S}_0 \quad \Gamma \vdash \mathtt{f} : \mathtt{S}_1 \quad \mathtt{S}_0 \otimes \mathtt{S}_1 = \mathtt{T}}{\Gamma \vdash \mathtt{e} \otimes \mathtt{f{:}T}} \qquad \text{I-Prop}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \mathit{lookup}(\mathtt{S}, \mathtt{n}) = \mathtt{T}}{\Gamma \vdash \mathtt{e.n{:}T}}$$

$$\text{I-Assign}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \mathtt{f{:}T} \quad \mathtt{T} \leqq \mathtt{S}}{\Gamma \vdash \mathtt{e} = \mathtt{f{:}T}}$$

$$\text{I-Func}\ \frac{\Gamma, \mathtt{this} : \mathtt{any}, \overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}} \qquad \overline{\mathtt{R}} \leqq \mathtt{T}}{\Gamma \vdash \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\} : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\}}$$

$$\text{I-Call}\ \frac{\Gamma \vdash \mathtt{e} : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{R}\} \quad \Gamma \vdash \overline{\mathtt{f}} : \overline{\mathtt{T}} \quad \overline{\mathtt{T}} \leqq \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{e}(\overline{\mathtt{f}}) : \mathtt{R}} \qquad \text{I-Assert}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \mathtt{S} \leqq \mathtt{T}}{\Gamma \vdash \mathtt{<T>e} : \mathtt{T}}$$

$$\text{I-AssertInf}\ \frac{\begin{array}{c}\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{M}}\} \qquad \{\overline{\mathtt{M}_p}\} = \{\mathtt{n} : \mathtt{T} \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}}\} \wedge \mathtt{T} \neq \mathtt{Undefined}\} \\ \{\overline{\mathtt{M}_p}\} \subseteq \mathit{properties}(\mathtt{I}) \qquad c_p = \{\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_p}\}\} \\ \{\overline{\mathtt{M}_{np}}\} = \mathit{properties}(\mathtt{I}) \setminus \{\overline{\mathtt{M}_p}\} \qquad c_{np} = \{\neg\mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \{\overline{\mathtt{M}_{np}}\}\} \\ v = c_p \cup c_{np} \qquad \hat{v}(\mathit{constraints}(\mathtt{I}))\end{array}}{\Gamma \vdash \mathtt{<I>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}}$$

$$\text{I-UpdateObj}\ \frac{\Gamma \vdash \mathtt{e} : \{\overline{\mathtt{M}}\} \qquad \Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{N}}\}}{\Gamma \vdash \mathtt{assign}(\mathtt{e}, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}) : \{\overline{\mathtt{M}}\} \Subset \{\overline{\mathtt{N}}\}}$$

$$\text{I-UpdateInf}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{I} \qquad \mathtt{I}' = \mathit{slice}(\mathtt{I}, \overline{\mathtt{n}}, \mathit{constraints}(\mathtt{I})) \qquad \Gamma \vdash \mathtt{<I'>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}' \\ \overline{\mathtt{n}} \in \mathit{dom}(\mathit{properties}(\mathtt{I})) \qquad \overline{\mathtt{n}} = \mathit{dom}(\mathit{properties}(\mathtt{I}'))\end{array}}{\Gamma \vdash \mathtt{assign}(\mathtt{e}, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}) : \mathtt{I}}$$

Figure 6.9: Typing rules of TIPC

$$\text{A-Trans}\ \frac{\mathtt{R} \leqq \mathtt{S} \quad \mathtt{S} \leqq \mathtt{T}}{\mathtt{R} \leqq \mathtt{T}} \qquad \text{A-Refl}\ \frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{S}} \qquad \text{A-AnyR}\ \frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{any}}$$

$$\text{A-Undefined}\ \frac{}{\mathtt{Undefined} \leqq \mathtt{void}} \qquad \text{A-Prim}\ \frac{\mathcal{I}(\mathtt{P}) \leqq \mathtt{T}}{\mathtt{P} \leqq \mathtt{T}}$$

$$\text{A-Object}\ \frac{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \vdash \diamond \quad \overline{\mathtt{M}}_1 \leqq \overline{\mathtt{M}}_2}{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \leqq \{\overline{\mathtt{M}}_2\}} \qquad \text{A-Prop}\ \frac{\mathtt{T} \vdash \diamond}{\mathtt{n} : \mathtt{T} \leqq \mathtt{n} : \mathtt{T}}$$

$$\text{A-CS}\ \frac{\overline{\mathtt{T}} \leqq \overline{\mathtt{S}} \quad \mathtt{R}_1 \neq \mathtt{void} \quad \mathtt{R}_0 \leqq \mathtt{R}_1}{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{R}_0 \leqq (\overline{\mathtt{y}} : \overline{\mathtt{T}}) : \mathtt{R}_1} \qquad \text{A-CS-Void}\ \frac{\overline{\mathtt{T}} \leqq \overline{\mathtt{S}} \quad \mathtt{R} \vdash \diamond}{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{R} \leqq (\overline{\mathtt{y}} : \overline{\mathtt{T}}) : \mathtt{void}}$$

$$\text{A-Interface}\ \frac{\begin{array}{c}\forall \mathtt{n} : \mathtt{S} \in \mathit{properties}(\mathtt{I}_0) : (\mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_1) \implies \mathtt{S} = \mathtt{T}) \\ c_0 = \{\neg \mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_0) \setminus \mathit{properties}(\mathtt{I}_1)\} \\ c_1 = \{\neg \mathtt{present}(\mathtt{n}) \mid \mathtt{n} : \mathtt{T} \in \mathit{properties}(\mathtt{I}_1) \setminus \mathit{properties}(\mathtt{I}_0)\} \\ \mathit{constraints}(\mathtt{I}_0) \cup c_1 \vDash_\ell \bigwedge \mathit{constraints}(\mathtt{I}_1) \wedge \bigwedge c_0\end{array}}{\mathtt{I}_0 \leqq \mathtt{I}_1}$$

$$\text{A-IntObj}\ \frac{\mathit{properties}(\mathtt{I}) \leqq \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} \qquad \mathit{constraints}(\mathtt{I}) \vDash_\ell \mathtt{present}(\overline{\mathtt{n}})}{\mathtt{I} \leqq \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

Figure 6.10: Assignment compatibility for types in TIPC

$$\text{I-EmpSeq}\ \frac{}{\Gamma \vdash \bullet : \bullet} \qquad \text{I-ReturnVal}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{return}\ \mathtt{e}; \overline{\mathtt{s}} : \mathtt{T}, \overline{\mathtt{R}}}$$

$$\text{I-Return}\ \frac{\Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{return}; \overline{\mathtt{s}} : \mathtt{void}, \overline{\mathtt{R}}} \qquad \text{I-ExpSt}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{e}; \overline{\mathtt{s}} : \overline{\mathtt{R}}}$$

$$\text{I-IfGeneral}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{boolean} \quad \Gamma \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1 \\ \Gamma \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2 \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}} : \overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}}}$$

$$\text{I-IfPresenceInterface}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{x} : \mathtt{I} \quad \mathtt{n} : \mathtt{S} \in \mathit{properties}(\mathtt{I}) \quad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}} \\ \mathtt{I}^- = \mathit{addConstraint}(\mathtt{I}, \neg\mathtt{present}(\mathtt{n})) \quad \Gamma \uplus \mathtt{x} : \mathtt{I}^- \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1 \\ \mathtt{I}^+ = \mathit{addConstraint}(\mathtt{I}, \mathtt{present}(\mathtt{n})) \quad \Gamma \uplus \mathtt{x} : \mathtt{I}^+ \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathtt{x.n} \equiv \mathtt{undefined})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}} : \overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}}}$$

$$\text{I-ITVarDec}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \mathtt{T} \leqq \mathtt{S} \quad \mathit{noDup}(\Gamma, \mathtt{x} : \mathtt{S}) \quad \Gamma \uplus \mathtt{x} : \mathtt{S} \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} : \mathtt{S} = \mathtt{e}; \overline{\mathtt{s}} : \overline{\mathtt{R}}}$$

Figure 6.11: Sequence typing rules in TIPC

$$\text{E-Id}\ \frac{\sigma(H, L, \mathtt{x}) = l}{\langle H, L, \mathtt{x}\rangle \Downarrow \langle H, (l, \mathtt{x})\rangle} \qquad \text{E-Lit}\ \frac{}{\langle H, L, \mathtt{l}\rangle \Downarrow \langle H, \mathtt{l}\rangle}$$

$$\text{E-this}\ \frac{\begin{array}{c}\sigma(H, L, @\mathtt{this}) = l_1 \\ H(l_1, @\mathtt{this}) = l\end{array}}{\langle H, L, \mathtt{this}\rangle \Downarrow \langle H, l\rangle} \qquad \text{E-Op}\ \frac{\begin{array}{c}\langle H_0, L, \mathtt{e}_1\rangle \Downarrow_v \langle H_1, \mathtt{l}_1\rangle \\ \langle H_1, L, \mathtt{e}_2\rangle \Downarrow_v \langle H_2, \mathtt{l}_2\rangle\end{array}}{\langle H_0, L, \mathtt{e}_1 \otimes \mathtt{e}_2\rangle \Downarrow \langle H_2, \mathtt{l}_1 \otimes \mathtt{l}_2\rangle}$$

$$\text{E-ObLit}\ \frac{\begin{array}{c}H_1 = H_0 * [l \mapsto new(l)] \\ \langle H_1, L, \mathtt{e}_1\rangle \Downarrow_v \langle H'_1, v_1\rangle \qquad H_2 = H'_1[(l, \mathtt{n}_1) \mapsto v_1] \\ \cdots \\ \langle H_m, L, \mathtt{e}_m\rangle \Downarrow_v \langle H'_m, v_m\rangle \qquad H = H'_m[(l, \mathtt{n}_m) \mapsto v_m]\end{array}}{\langle H_0, L, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}\rangle \Downarrow \langle H, l\rangle}$$

$$\text{E-Assign}\ \frac{\langle H_0, L, \mathtt{e}_1\rangle \Downarrow \langle H_1, (l, x)\rangle \qquad \langle H_1, L, \mathtt{e}_2\rangle \Downarrow_v \langle H_2, v\rangle}{\langle H_0, L, \mathtt{e}_1 = \mathtt{e}_2\rangle \Downarrow \langle H_2[(l, x) \mapsto v], v\rangle}$$

$$\text{E-Update}\ \frac{\begin{array}{c}\langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H'_0, l\rangle \qquad H_1 = H'_0 * [l_r \mapsto clone(H'_0(l), l_r)] \\ \langle H_1, L, \mathtt{e}_1\rangle \Downarrow_v \langle H'_1, v_1\rangle \qquad H_2 = H'_1[(l_r, \mathtt{n}_1) \mapsto v_1] \\ \cdots \\ \langle H_m, L, \mathtt{e}_m\rangle \Downarrow_v \langle H'_m, v_m\rangle \qquad H = H'_m[(l_r, \mathtt{n}_m) \mapsto v_m]\end{array}}{\langle H_0, L, \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\})\rangle \Downarrow \langle H, l_r\rangle}$$

$$\text{E-Prop}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H_1, l\rangle \quad l \neq \mathtt{null}}{\langle H_0, L, \mathtt{e.n}\rangle \Downarrow \langle H_1, (l, \mathtt{n})\rangle}$$

$$\text{E-Prop'}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H_1, \mathtt{l}\rangle \quad H_2 = H_1 * [l_{boxed} \mapsto box(\mathtt{l}, l_{boxed})]}{\langle H_0, L, \mathtt{e.n}\rangle \Downarrow \langle H_2, (l_{boxed}, \mathtt{n})\rangle}$$

$$\text{E-Call}\ \frac{\begin{array}{cc}\langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & \gamma(H_1, r) = l_1 \\ H_1(l_1) = \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L_1\rangle & This(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_v \langle H_2, v_1\rangle \quad \ldots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_v \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * act(l, \overline{\mathtt{x}}, \overline{v}, l_2) & \langle H', l : L_1, \overline{\mathtt{s}}\rangle \Downarrow \langle H'', \mathtt{return}\ v;\rangle\end{array}}{\langle H_0, L_0, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\rangle \Downarrow \langle H'', v\rangle}$$

$$\text{E-CallUndef}\ \frac{\begin{array}{cc}\langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & \gamma(H_1, r) = l_1 \\ H(l_1) = \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L_1\rangle & This(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_v \langle H_2, v_1\rangle \quad \ldots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_v \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * act(l, \overline{\mathtt{x}}, \overline{v}, l_2) & \langle H', l : L_1, \overline{\mathtt{s}}\rangle \Downarrow \langle H'', \mathtt{return};\rangle\end{array}}{\langle H_0, L_0, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\rangle \Downarrow \langle H'', \mathtt{undefined}\rangle}$$

Figure 6.12: Operational semantics of TIPC

$$\text{E-Func}\ \frac{H_1 = H_0 * [l \mapsto \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L\rangle]}{\langle H_0, L, \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\}\rangle \Downarrow \langle H_1, l\rangle}$$

$$\text{E-TypeAssert}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow \langle H_1, r_1\rangle}{\langle H_0, L, \mathtt{<T>e}\rangle \Downarrow \langle H_1, r_1\rangle}$$

$$\text{E-TypeAssertInf}\ \frac{\langle H_0, L, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}\rangle \Downarrow_v \langle H_1, l\rangle \quad H = H_1[(l, \mathtt{@interface}) \mapsto \mathtt{I}]}{\langle H_0, L, \mathtt{<I>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}\rangle \Downarrow \langle H, l\rangle}$$

Figure 6.12: Operational semantics of TIPC (continued)

$$\text{E-EmptySeq}\ \frac{}{\langle H, L, \bullet\rangle \Downarrow \langle H, ;\rangle} \qquad \text{E-Return}\ \frac{}{\langle H, L, \texttt{return}; \overline{\texttt{s}}\rangle \Downarrow \langle H, \texttt{return};\rangle}$$

$$\text{E-ReturnVal}\ \frac{\langle H, L, \texttt{e}\rangle \Downarrow_v \langle H_1, v\rangle}{\langle H, L, \texttt{return e}; \overline{\texttt{s}}\rangle \Downarrow \langle H_1, \texttt{return } v;\rangle}$$

$$\text{E-ExpSt}\ \frac{\langle H, L, \texttt{e}\rangle \Downarrow \langle H_1, r\rangle \quad \langle H_1, L, \overline{\texttt{s}}\rangle \Downarrow \langle H_2, s\rangle}{\langle H, L, \texttt{e}; \overline{\texttt{s}}\rangle \Downarrow \langle H_2, s\rangle}$$

$$\text{E-IfTrue}\ \frac{\begin{array}{c}\langle H, L, \texttt{e}\rangle \Downarrow_v \langle H_1, \texttt{true}\rangle \\ H_2 = H_1 * [l \mapsto ()] \\ \langle H_2, l : L, \overline{\texttt{t}}_1\rangle \Downarrow \langle H_3, s\rangle \\ \langle H_3, L, s; \overline{\texttt{s}}\rangle \Downarrow \langle H_4, s_r\rangle\end{array}}{\langle H, L, \texttt{if (e) } \{\overline{\texttt{t}}_1\} \texttt{ else } \{\overline{\texttt{t}}_2\}; \overline{\texttt{s}}\rangle \Downarrow \langle H_4, s_r\rangle}$$

$$\text{E-IfFalse}\ \frac{\begin{array}{c}\langle H, L, \texttt{e}\rangle \Downarrow_v \langle H_1, \texttt{false}\rangle \\ H_2 = H_1 * [l \mapsto ()] \\ \langle H_2, l : L, \overline{\texttt{t}}_2\rangle \Downarrow \langle H_3, s\rangle \\ \langle H_3, L, s; \overline{\texttt{s}}\rangle \Downarrow \langle H_4, s_r\rangle\end{array}}{\langle H, L, \texttt{if (e) } \{\overline{\texttt{t}}_1\} \texttt{ else } \{\overline{\texttt{t}}_2\}; \overline{\texttt{s}}\rangle \Downarrow \langle H_4, s_r\rangle}$$

$$\text{E-ITVarDec}\ \frac{\begin{array}{c}\langle H, L, \texttt{e}\rangle \Downarrow_v \langle H_1, v\rangle \\ H_2 = H_1 * [l \mapsto (\{\texttt{x} \mapsto v\})] \\ \langle H_2, l : L, \overline{\texttt{s}}\rangle \Downarrow \langle H_3, s\rangle\end{array}}{\langle H, L, \texttt{let x : S} = \texttt{e}; \overline{\texttt{s}}\rangle \Downarrow \langle H_3, s\rangle}$$

Figure 6.13: Operational semantics of sequences in TIPC

Chapter 7

$\text{TypeScript}_{\text{IPC}}$: Implementation of TIPC

In the previous chapter, we presented the formalisation of TIPC, a programming language with support for constraints between interfaces properties. This chapter presents a prototypical implementation of TIPC, called $\text{TypeScript}_{\text{IPC}}$. In TIPC, existing TypeScript interface definitions are replaced by advanced interface definitions with support for inter-property constraints. However, in order to enable the step-by-step integration of inter-property constraints into existing TypeScript applications, our implementation of TIPC supports both regular interfaces and interfaces with inter-property constraints. While TIPC contains a subset of TypeScript, $\text{TypeScript}_{\text{IPC}}$ is an extension of the complete TypeScript programming language. The implementation of $\text{TypeScript}_{\text{IPC}}$ is available at `https://github.com/noostvog/typescriptipc`[1] and is also published as an evaluated artefact [Oostvogels et al., 2018a].

We start this chapter by briefly describing the architecture and design of the TypeScript compiler in Section 7.1. Next, Section 7.2 covers the differences between the formalisation (TIPC) and the implementation ($\text{TypeScript}_{\text{IPC}}$), which can be attributed to technical limitations and to the co-existence of regular interface definitions and interface definitions with inter-property constraints. Finally, Section 7.3 presents the details of the implementation. In the implementation, we extend three applications and combine them:

- the TypeScript compiler[2];

[1] To avoid confusion with regards to terminology, occurrences of `predicate` are replaced with `constraint` in the function and variable names in this chapter.

[2] `https://github.com/Microsoft/TypeScript`

- a library that provides a DPLL (Davis-Putman-Logemann-Loveland) satisfiability solver[3];
- a library that provides a propositional sequent calculus prover[4].

In Chapter 9, the implementation will be used in our discussion of API specification languages, where we present a tool that generates the server stub in TypeScript$_{\text{IPC}}$ from a web API specification.

7.1 Architecture and Design

In this section, we explain the architecture of the TypeScript compiler. A compiler typically translates source code to an executable program. TypeScript's compiler differs from standard compilers in that it does not create an executable program. Instead, it translates the source code to its equivalent in JavaScript (without type annotations). These kinds of compilers are also called *transpilers*.

At a high level, the TypeScript compiler is structured as follows. In the first phase, the scanner reads the characters in the input source code and produces a sequence of *tokens*. The parser then generates an abstract syntax tree (AST) from the tokens. Next, the type checker verifies the type safety of the AST, and generates diagnostic error messages when this is not the case. Regardless of whether or not the program is type safe, the TypeScript compilation process always results in a JavaScript file. Each phase of the compiler resides in a separate source file.

The TypeScript compiler is written in TypeScript itself. Several parts of the TypeScript compiler were extended or adapted in order to incorporate the addition of inter-property constraints to interfaces. Below, we list all the parts of the compiler that required a change, and briefly explain what changed. In Section 7.3, we elaborate on these changes.

`types.ts` (Section 7.3.1) As the compiler of TypeScript is written in TypeScript itself, it uses types to facilitate the development and improve the correctness of its implementation. This file contains all type declarations that are used throughout the implementation of the compiler. Every type representation used in the TypeScript compiler has a property `kind` with an enum `SyntaxKind` value to identify the kind of type. This enum contains the type for, among others, keywords, tokens, expressions, statements, declarations and TypeScript types.

[3] `https://github.com/tammet/logictools`

[4] `https://www.nayuki.io/page/propositional-sequent-calculus-prover`

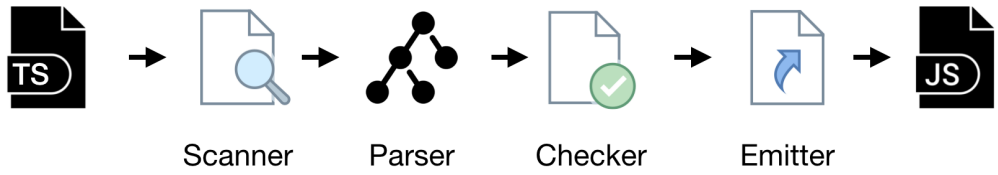


Figure 7.1: Diagram of the TypeScript compiler

`scanner.ts` (Section 7.3.2) The TypeScript scanner transforms the source code into a stream of tokens. All keywords and tokens are replaced by their equivalent in the `SyntaxKind` enum.

`parser.ts` (Section 7.3.3) The parser uses the token stream from the scanner to generate an abstract syntax tree of `Node`s, which form the basis of, among others, expressions, statements, declarations, literals and comments.

`checker.ts` (Section 7.3.4) The type checker performs the second semantic pass (the first semantic pass is performed by the binder, see below) of the source file in TypeScript. It uses the AST from the parser and the symbols from the binder to verify the type safety of the AST.

`emitter.ts` (Section 7.3.5) The emitter is the last step of the compiler process. Compiling a TypeScript program results in a JavaScript program. Next to omitting all type annotations, the emitter also transforms some TypeScript language constructs that are not supported by regular JavaScript, such as `let` declarations.

Figure 7.1 shows a diagram of the main parts of TypeScript compiler. There are also a few helper files that we have extended. These extensions are trivial and will not be discussed in this chapter.

`diagnosticMessages.json` During the scanning, parsing, binding and type checking, errors can occur. All possible error messages are collected in this file, which is used to generate a constant object called `Diagnostics` in which every property represents one error message or warning.

`binder.ts` The binder performs the first semantic pass over the code. The binder creates the symbol table and populates it with symbols. Symbols are used to connect linguistic entities that refer to the same declaration. This helps the

type checker to reason about named declarations such as interfaces, functions and modules. The binder also records the flow and antecedents, which are used by the type system to perform flow-sensitive type analysis.

`factory.ts` and `visitor.ts` The TypeScript compiler uses the factory design pattern and visitor design pattern by Gamma et al. [1995].

7.2 Differences between Formalisation and Implementation

Before we explain the details of how the formalisation of Chapter 6 maps onto the implementation, we first clarify some differences between the two. Chapter 6 presented the formalisations of a safe subset of TypeScript. On the other hand, TypeScript$_{\text{IPC}}$ is implemented on top of TypeScript, which includes features such as unsoundness, optional object properties and a more advanced flow-sensitive type analysis. In this section, we elaborate on the changes that were necessary in order to incorporate inter-property constraints in TypeScript.

Version The formalisations in Chapter 6 were based on version 0.9.5 of TypeScript. TIPC is built on top of TypeScript version 2.1.6. Most of the features added since version 0.9.5 are orthogonal to our extension of TypeScript with inter-property constraints, except for the strict null-checking mode. This mode was introduced in version 2.0, but already incorporated in our formalisations of Chapter 6.

7.2.1 Interface Definition

In TIPC, state-of-the-art interfaces are replaced with interfaces that support inter-property constraints. In order to stay consistent with existing TypeScript programs, we did not supplant the existing interface definition of TypeScript with the interface definitions from the paper. Instead, there are now two ways to define interfaces in TypeScript$_{\text{IPC}}$. Apart from the original TypeScript interfaces (in which properties can only be required or optional), interfaces can now also contain an extra section in which presence constraints can be expressed.

Listing 7.1 shows the interface definition of the `PrivateMessage` example in TypeScript$_{\text{IPC}}$. At the top, all properties are defined as optional. The second section contains all constraints: text is a required property, while there is an

```
interface PrivateMessage {
    text?      : string;
    userid?    : number;
    screenname?: string;
} constrains {
    present(text);
    or(and(    present(userid),  not(present(screenname))),
       and(not(present(userid)),     present(screenname)));
}
```

Listing 7.1: Definition of a TIPC interface in TypeScript$_{\text{IPC}}$

exclusivity constraint imposed on the ID and the name. The exclusivity constraint is rewritten using only `and`, `or` and `not`. In TypeScript$_{\text{IPC}}$, constraints are written in prefix notation instead of infix notation, and uses `not` instead of ¬.

Interface definitions are treated as IPC-interfaces (interfaces with inter-property constraints) if the interface contains at least one *constraint*. To avoid contradictions between the property list and the constraints, properties of IPC-interfaces have to be indicated as optional using a question mark in the interface property list and interface properties may not have the type `undefined`.

In TypeScript, reserved keywords may not be longer than eleven characters. Therefore, we have replaced the keyword for indicating the second part of the interface definition (`constraining`) by `constrains`.

7.2.2 Object Creation

In Section 4.1, we have explained that there are two kinds of initialising interface instances: via variable declaration or via type casting. As TIPC supports width subtyping, assignments such as

```
let pm: {text: string,   user_id: number, screen_name: string}
      = {text: "Hello!", user_id: 42,     screen_name: "Alice"};
```

are accepted by TIPC's type system. Therefore, TIPC limits the ways that interface instances can be created: the right-hand side of an assignment or cast must be an object literal.

While TIPC only allows the initialisation via a type cast, TypeScript$_{\text{IPC}}$ supports both type casts and variable declarations. TypeScript$_{\text{IPC}}$ is based on a more recent version of TypeScript (2.1.6 instead of 0.9.5) and disallows extra properties in an assignment of an object literal to an object literal type. However, it is still possible to have hidden properties when a variable of an interface type is assigned to a variable of an object type (literal or interface). Therefore, it is still

necessary for TypeScript$_{\text{IPC}}$ to restrict the assignment to IPC-interfaces to fresh object literals.

Moreover, caution is required: as TypeScript$_{\text{IPC}}$ is an extension of TypeScript, it supports both upcasts and downcasts. As a consequence, unsound casts will be accepted by TypeScript$_{\text{IPC}}$'s type system (as opposed to TIPC). For details, we refer to Section 5.2.

7.2.3 Assignment

Two kinds of interface definitions TypeScript$_{\text{IPC}}$ supports both regular interfaces and IPC-interfaces. To remain type safe, assignments from or to interface instances will fail when only one of the interfaces has inter-property constraints.

Updating multiple properties at once The functional simultaneous update of properties in TIPC (`assign`) is written as `objupdate` in TypeScript$_{\text{IPC}}$. Currently, `objupdate` only accepts instances of IPC-interfaces. Regular objects can simply use the `Object.assign` method from JavaScript.

Optional object properties In TIPC, all properties of an object literal type are required. TypeScript, on the other hand, allows object literal properties to be optional as well. To incorporate optional properties in TypeScript$_{\text{IPC}}$, the type system of TypeScript$_{\text{IPC}}$ has to be extended. More specifically, TypeScript$_{\text{IPC}}$ has to guarantee type safety when an IPC-interface is assigned to an expression of an object literal type. The relevant assignment compatibility rule (A-IntObj, see Figure 6.10) now has to verify whether or not the optional properties are also a part of the interface property list. Additional constraint checks are unnecessary as an optional presence constraint is always true.

Formally, the type system is extended with the rule A-IntObj' shown in Figure 7.2. The difference between the original A-IntObj and A-IntObj' is indicated with a grey background. Object literal types now contain required properties ($\overline{\texttt{M}}$) and optional properties ($\overline{\texttt{M?}}$). For an interface to be assignable to an object literal type, there are two requirements. First, the properties of the interface must be assignable to the properties of the object literal type (both the required and the optional properties). This ensures that the types of the corresponding properties are compatible. Second, all required properties of the object literal type have to be certainly present in the interface instance.

$$\text{A-IntObj'}\ \frac{\textit{properties}(\mathtt{I}) \leqq \{\overline{\mathtt{M}}, \overline{\mathtt{M?}}\} \qquad \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\} = \{\overline{\mathtt{M}}\} \quad \textit{constraints}(\mathtt{I}) \vDash_{\ell} \mathtt{present}(\overline{\mathtt{n}})}{\mathtt{I} \leqq \{\overline{\mathtt{M}}, \overline{\mathtt{M?}}\}}$$

Figure 7.2: Assignment compatibility for object literal types with optional properties in TypeScript$_{\text{IPC}}$

```
1 function getUser(pm: PrivateMessage): string | number {
2    if(pm.userid) {
3       return pm.userid;
4    }
5    return pm.screenname;
6 }
```

Listing 7.2: Reusing TypeScript's control-flow mechanism

7.2.4 If statements

TIPC uses `if` statements to find out more information about the presence or absence of object properties. That extra information is taken into account inside these `if` statements when property accesses and updates are type checked.

As we have already explained in Section 5.6, TypeScript already uses flow-sensitive type analysis to infer information about the types of variables. TypeScript$_{\text{IPC}}$ reuses this analysis to take inter-property constraints into account. This way, TypeScript$_{\text{IPC}}$ has a more advanced flow-sensitive analysis than TIPC that takes control flow constructs such as `return` and `break` into account.

The program shown in Listing 7.2 is accepted by TypeScript$_{\text{IPC}}$'s type checker: the `return` statement on line 3 is taken into account. Therefore, it is certain that the private message will not contain a user ID when leaving the `if` statement, and thus it is also certain that the screen name will be present.

TypeScript$_{\text{IPC}}$ reuses TypeScript's flow-sensitive analysis. As a consequence, TypeScript$_{\text{IPC}}$ has to take a different approach from TIPC on how to take the information from `if` statements into account. In TIPC, extra constraints extracted from the `if` statement are added to a new interface type, which is a copy of the original interface adding the new constraint (`I+` and `I-` in rule I-IfPresenceInterface of Figure 6.11). These extended types are used in the true and false branch. In TypeScript$_{\text{IPC}}$, the exact type of an identifier with an IPC-interface type is recalculated given the history of the identifier.

File name	Lines changed	Total lines of code
`binder.ts`	26	3395
`checker.ts`	1143	22918
`diagnosticMessages.json`	104	3354
`emitter.ts`	26	2808
`factory.ts`	29	3447
`parser.ts`	275	7759
`scanner.ts`	6	1917
`types.ts`	61	3954
`visitor.ts`	8	1417

Table 7.1: Lines changed per component of the compiler

7.3 Extending the TypeScript Compiler with Inter-property Constraints

In this section, we explain the changes made to the TypeScript compiler that were required in order to incorporate interfaces inter-property constraints. Section 7.3.1 shows which additions and changes were necessary to the type definitions, while Sections 7.3.2 to 7.3.5 explain the changes in every phase of the compiler. To give an idea of the scope of these changes, Table 7.1 shows the amount of lines that were changed per component of the compiler.

7.3.1 Types

The `types.ts` file contains the interface definitions and enum types that are used throughout the compiler.

The main type used in the TypeScript compiler is the enum `SyntaxKind`, which is used to identify the kind of an AST node. Nodes include keywords, expressions, statements and declarations. In TypeScript$_{\text{IPC}}$, we have extended the `SyntaxKind` enum with new keywords and expression kinds. Listing 7.3 shows the additions.

To reflect the additions to the syntax of TypeScript$_{\text{IPC}}$, we have added two new keywords: `ConstrainsKeyword` (used to indicate the second part of an interface definition) and `ObjectUpdateKeyword` (used to update multiple properties of an object with inter-property constraints simultaneously).

We also extended `SyntaxKind` with three new expression kinds (see Listing 7.3): one expression kind for `objupdate` and two new expression kinds for the constraints of an interface definition.

All new expression kinds have a corresponding expression type, which are

```
export const enum SyntaxKind {
    ...
    ConstrainsKeyword,
    ObjectUpdateKeyword,

    ObjectUpdateExpression,
    ConstraintLogicalExpression,
    ConstraintPresentExpression,
    ...
}
```

Listing 7.3: Extension of `SyntaxKind`

```
export interface ObjectUpdateExpression extends UnaryExpression {
    kind      : SyntaxKind.ObjectUpdateExpression;
    arguments: NodeArray<Expression>;
}
```

Listing 7.4: Definition of `ObjectUpdateExpression`

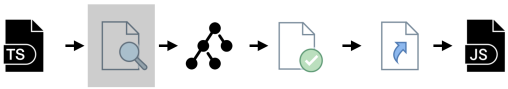
shown in Listings 7.4 and 7.5. The interface `ObjectUpdateExpression` is used to define the type of an `objupdate` expression. Next to the kind of the interface, this interface only indicates the list of arguments for a call to `objupdate`.

The TypeScript compiler is also extended with two constraint expression types. Next to the property `kind`, both interfaces have an `expression` property and an `arguments` property. For logical constraints, `expression` can be any logical connective, while for the present constraint this may only be `present` (this is verified in the type checker).

Finally, we have extended the type for interface declarations with inter-property constraints (Listing 7.6). To start, we extended the `InterfaceDeclaration` interface with a new property `constraints`: this is an array which contains expressions of type `ConstraintExpression`. Next, the `ResolvedType` interface had to be extended as well. This interface is the type of a *resolved* interface (among other types): next to the members and constraints declared in the interface definitions, resolved interfaces also contain members and constraints inherited from other interface definitions. `ResolvedType` is extended with two new properties: `origConstraints` which contains the constraints from the interface definition and `constraints` which also contains constraints from superinterfaces.

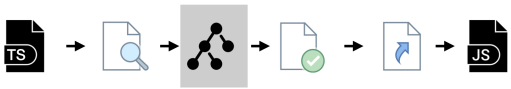
```
1  export type ConstraintExpression = ConstraintPresentExpression |
2                                     ConstraintLogicalExpression;
3
4  export interface  ConstraintLogicalExpression extends Node {
5      kind       : SyntaxKind.ConstraintLogicalExpression;
6      expression: Identifier;
7      arguments  : NodeArray<ConstraintExpression>;
8  }
9
10 export interface ConstraintPresentExpression extends Node {
11     kind       : SyntaxKind.ConstraintPresentExpression;
12     expression: Identifier;
13     arguments  : NodeArray<Identifier>;
14 }
```

Listing 7.5: Definition of `ConstraintExpressions`



7.3.2 Scanner Extensions

Changes to the scanner of the TypeScript compiler are minimal: we have added two keywords and mapped them onto their respective `SyntaxKind` (see Listing 7.7).



7.3.3 Parser Extensions

To support inter-property constraints in interface definitions, we have adapted the parser of the TypeScript compiler. When parsing interface declarations (Listing 7.8), we now also (optionally) parse inter-property constraints.

Listing 7.9 shows the definition of `parseInterfaceConstraints`. This function starts with parsing the `ConstrainsKeyword`. Note that this is optional: when there is none, the function returns an empty array. In the case that there is a `constrains` block, the parse function parses the statements between the curly braces (lines 4–6) and transforms them into an array of constraints (lines 7–8). `checkValidConstraints` expects an array of statements, checks whether they form valid constraints and returns an array of `ConstraintExpressions`. This array is returned as the result of `parseInterfaceConstraints`.

In short, `checkValidConstraint` receives an array of statements and verifies that every statement is a constraint expression. Recall that constraint expres-

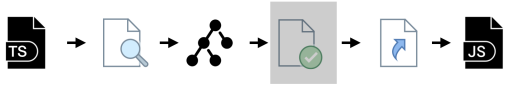
```
export interface InterfaceDeclaration extends DeclarationStatement {
    kind            : SyntaxKind.InterfaceDeclaration;
    name            : Identifier;
    typeParameters? : NodeArray<TypeParameterDeclaration>;
    heritageClauses?: NodeArray<HeritageClause>;
    members         : NodeArray<TypeElement>;
    constraints     : ConstraintExpression[];
}

export interface ResolvedType
                      extends ObjectType, UnionOrIntersectionType {
    members             : SymbolTable;
    properties          : Symbol[];
    callSignatures      : Signature[];
    constructSignatures: Signature[];
    stringIndexInfo?    : IndexInfo;
    numberIndexInfo?    : IndexInfo;
    origconstraints     : ConstraintExpression[];
    constraints         : ConstraintExpression[];
}
```

Listing 7.6: Extension of the interface definition types

sions in TypeScript$_{\text{IPC}}$ are in prefix notation: presence constraints are of the form `present(x)` and logical expressions are of the form `and(...,...)`. The TypeScript parser will parse all these constraints as call expressions, as they are syntactically equivalent to function calls. The function `checkValidConstraints` transforms these call expressions to constraint expressions. It starts with verifying that the statement is a `CallExpression`. The function name of the call expression must be either `present` or a logical connective. In the case of a `present` call, every argument has to be an `Identifier`. In the case of a logical connective, every argument has to be a valid constraint expression.

Finally, the parser is also extended for parsing `objupdate` expressions. For this, we reuse the existing `parseSimpleUnaryExpression` function.

7.3.4 Checker Extensions



This section gives an overview of the changes that were required to incorporate IPC-interfaces into TypeScript. The adaptations to the TypeScript type system are scattered throughout the more than 20,000 lines of code in `checker.ts`. This

```
const textToToken = createMap({
    ...
    "constrains": SyntaxKind.ConstrainsKeyword,
    "objupdate" : SyntaxKind.ObjectUpdateKeyword,
    ...
});
```

Listing 7.7: Extension of the scanner

```
function parseInterfaceDeclaration
         (fullStart : number,
          decorators: NodeArray<Decorator>,
          modifiers : NodeArray<Modifier>): InterfaceDeclaration {
    const node   = <InterfaceDeclaration>
         createNode(SyntaxKind.InterfaceDeclaration, fullStart);
    ...
    parseExpected(SyntaxKind.InterfaceKeyword);
    node.name    = parseIdentifier();
    ...
    node.members = parseObjectTypeMembers();
    node.constraints = parseInterfaceConstraints();
    return addJSDocComment(finishNode(node));
}
```

Listing 7.8: Extension of parsing interface declarations

section only highlights the parts of the type checker that were changed.

7.3.4.1 Interface Definition

The function `checkInterfaceDeclaration` performs semantic checks on interface declarations. This includes, among others, verifying whether the name of the interface is not a reserved name, that inherited properties are assignment-compatible, and that there are no duplicate properties.

When an interface definition has (inter-property) presence constraints, the function `checkInterfaceDeclaration` has to verify more properties (listed below). This is all covered by `checkInterfaceWithConstraintDeclarations`, defined in Listing 7.10. This function verifies that:

- the declared constraints are valid (line 4). More specifically, the function `checkInterfaceConstraintDeclaration` checks whether the constraint is a logical constraint declaration (with a valid logical connective) or a present constraint declaration. For logical constraint declarations, it checks whether

```
1  function parseInterfaceConstraints():NodeArray<ConstraintExpression>{
2      if (parseOptional(SyntaxKind.ConstrainsKeyword)) {
3          parseExpected(SyntaxKind.OpenBraceToken);
4          const constraints: NodeArray<Statement> =
5            parseList(ParsingContext.BlockStatements,
6                      parseStatement);
7          const parsedconstraints: NodeArray<ConstraintExpression> =
8            checkValidConstraints(constraints);
9          parseExpected(SyntaxKind.CloseBraceToken);
10         if (!parsedconstraints) {
11             return createMissingList<ConstraintExpression>();
12         }
13         return parsedconstraints;
14     } else {
15         return createMissingList<ConstraintExpression>();
16     }
17 }
```

Listing 7.9: Definition of `parseInterfaceConstraints`

they receive the correct number of arguments. For presence constraints, the function checks that all arguments are properties defined in the interface or its superinterfaces. We omit the straightforward implementation of the function `checkInterfaceConstraintDeclaration`.

Note that we use the constraints of the *resolved* interface type: this ensures that constraints inherited from other interfaces are also checked. The TypeScript compiler resolves object types such that properties inherited from superinterfaces are all merged into one object type. We have extended the function `resolveObjectTypeMembers` such that the constraints of an interface inheritance chain are also collected in the resolved object type;

- IPC-interfaces are used in the strict null-checks mode of TypeScript (line 6–8). `strictNullChecks` is a constant in the type checker;
- there are no properties of type `undefined` (line 12–14).
- all properties are declared as optional (lines 16–18);
- the set of constraints is satisfiable (lines 21–24). To check this we reused an existing library for satisfiability checking, by Tanel Tammet. This library is an implementation of the DPLL (Davis-Putnam-Logemann-Loveland) solver for the satisfiability of propositional logic formula.

```
function checkInterfaceWithConstraintDeclarations(type:InterfaceType){
  if (resolved.constraints && resolved.constraints.length > 0) {
    const resolved = resolveStructuredTypeMembers(type);
    resolved.constraints.map(checkInterfaceConstraintDeclaration);

    if (!strictNullChecks) {
      error(node, Diagnostics.IPCInterface_StrictNullChecks);
    }

    for (const member of node.members) {
      const type = getTypeOfSymbol(member.symbol);
      if (type === undefinedType) {
        error(node, Diagnostics.TypeNotUndefined, node.symbol.name);
      }

      if (!(member.symbol.flags & SymbolFlags.Optional)) {
        error(node, Diagnostics.MustBeOptional, node.symbol.name);
      }
    }

    if (!Proplog.solve(translatePreds(resolved.constraints, "-"),
                       "none")) {
      error(node, Diagnostics.Satisfiable, node.symbol.name);
    }
  }
}
```

Listing 7.10: Definition of `checkInterfaceWithConstraintDeclarations`

7.3.4.2 Property Access

When a property of an IPC-interface is accessed, the type checker has to consult the constraints of the interface in order to assign a type to the property access or reject it (as defined in I-Prop (Figure 6.9) and the *lookup* function (Figure 6.5)).

Listing 7.11 on page 150 shows the implementation of the function `checkPropertyAccessExpressionOrQualifiedName`. Lines 8–37 are new and cover property accesses on instances of IPC-interfaces; other parts of the function are omitted.

On line 15 and 16, we use the Propositional Sequent Calculus Prover by Project Nayuki to try to prove the presence and absence of the property being accessed. This can result in three situations:

- neither the presence nor the absence can be proven (lines 17–20): the function raises an error message and returns `unknownType`[5];

- the presence of a property can be proven (lines 21–31): the function should return the declared type in the property list of the interface. At first sight, it looks like `propType` is exactly this. However, because all properties in the interface definitions have to be declared as optional, TypeScript automatically produces a union type that adds `undefined`. Therefore, we have to remove the `undefined` type from the property union type.

  Caution is required, though: the TypeScript compiler maintains one object to represent a specific type. This means that common types such as `string | undefined` are reused in other places as well. Thus, removing `undefined` from `propType` would remove `undefined` out of the type for every variable of type `string | undefined`! Therefore, the function first clones `propType` and its property `types` using `Object.assign` (lines 22 and 23). Next, it filters out the `undefined` type (line 24). Finally, propType receives the filtered union type (line 25). The function `getUnionType` transforms an array of types to a union type, or one type if the array only contains one element.

- the absence of a property can be proven (lines 32-34): the function overwrites the defined type of the property in the interface definition with `undefined`.

[5] `unknownType` is not to be confused with the `unknown` type introduced in TypeScript 3.0. $\text{TypeScript}_{\text{IPC}}$ is based on TypeScript 2.1.6 and does not yet support the `unknown` type.

7.3.4.3 Assignment Compatibility

The function `objectTypeRelatedTo` checks whether two object types (object literal types, object types and interface types) are related by structure (see Section 6.3.2). To verify this, the function compares the properties, signatures and index types of both object types. TypeScript$_{\text{IPC}}$ extends this function such that the presence constraints of the interfaces are also taken into account. This corresponds to the logic of I-AssertInf (defined in Figure 6.9, an adapted version where object literals can also be assigned to interface variables using variable declarations), A-Interface and A-IntObj (defined in Figure 6.10).

The definition of `objectTypeRelatedTo` spans more than 400 lines of code. Therefore, we limit our discussion of the extensions made by TypeScript$_{\text{IPC}}$.

Before comparing properties, signatures and index types, TypeScript$_{\text{IPC}}$'s `objectTypeRelatedTo` first verifies whether the constraints are related, using the function `constraintsRelatedTo`. In the case that one of the object types has constraints, this function replaces the comparison of properties (`propertiesRelatedTo`). `constraintsRelatedTo` inspects the types of both objects and calls the corresponding helper function (either the function `constraintsRelatedToObjIntf`, `constraintsRelatedToIntfIntf` or `constraintsRelatedToIntfObj`). When there are no constraints involved, the function returns `Maybe` to indicate that properties still need to be compared. In the following two cases, `constraintsRelatedTo` raises an error:

- when an object literal type is assigned to an interface type with constraints, but the object literal type is not a fresh literal;
- when an interface with constraints is assigned to an interface without constraints, or the other way around.

In the rest of this section, we discuss the three helper functions.

`constraintsRelatedToObjIntf` When an object literal is assigned to an interface with inter-property constraints, the object literal has to be a valuation of the interface constraints. Listing 7.12 shows the implementation of the function `constraintsRelatedToObjIntf`. For every constraint, `isConstraintSatisfied` verifies whether the object literal satisfies the constraint. The implementation of this valuation function is straightforward and omitted.

`constraintsRelatedToIntfIntf` Listing 7.13 on page 152 shows the implementation of the function that covers the assignment compatibility check for an interface with constraints to another interface with constraints. This rule is a trans-

lation from A-Interface (Figure 6.10). This function starts with translating the constraints of the source and target interface from a JavaScript object to a string (such that it can later on be used in a library for logical entailment) (lines 7 and 8).

Before proving the logical entailment, the function first has to take the differences between both property lists into account. On lines 14–18, all properties from the target interface which are not present in the source interface are added as absent properties to the set of source constraints. Similarly, all properties from the source interfaces which are not present in the target interface are added as absent properties to the set of target constraints (lines 20–24).

On line 26, this function verifies whether or not the constraints from the target follow from the constraints of the source. To achieve this, we use the Propositional Sequent Calculus Prover again.

`constraintsRelatedToIntfObj` Listing 7.14 on page 153 shows the implementation of the function that covers the assignment compatibility check for an interface with constraints to an object type. This rule is a translation from the rule presented earlier in this chapter: A-IntObj' (Figure 7.2). For each property in the target object, this function checks the following:

- lines 9–20 cover the case that the target property is optional. In this case, the property has to be in the property list of the source interface;
- lines 21–43 cover the case of a required target property. In this case, it does not suffice to only check whether the property is part of the source interface. Instead, we have to prove that the properties of type `undefined` are absent in the interface (lines 23–32) and all the other properties are present (lines 33–41). Note that A-IntObj and A-IntObj' do not handle the case of object properties of type `undefined`.

7.3.4.4 Objupdate

The function `checkObjectUpdate` checks a call to `objupdate` and is an implementation of the rule I-UpdateInf (Figure 6.9).

Listing 7.15 on page page 154 shows the implementation of the function `checkObjectUpdate`. Lines 8–12 and 15–18 verify that the first argument is an IPC-interface and lines 18–20 verify whether the second argument is an object literal. Next, the function clones the target type (line 28, the definition of `cloneTarget` is omitted) and slices the target type such that only the relevant properties and constraints remain (line 29). Lines 32–34 verify that the

property names of the second arguments are identical to the property names of the sliced target type (the definition of `sameProps` is omitted). Finally, we reuse the existing function `checkTypeAssignableTo` which will call the function `constraintsRelatedToObjIntf` to verify whether the object literal satisfies the constraints of the sliced target (line 35).

Listing 7.16 on page 155 shows the implementation of *slice*. It calculates the transitive closure of all properties and constraints of the given interface, starting with the set of properties in the second argument of `objupdate` (`sourcePart`). The relevant constraints are stored in `collectedConstraints` (line 4) and the relevant properties are stored in `collectedProperties` (line 5). On line 6 and 7, the constraints that mention one of the properties of `sourcePart` are added to `collectedProperties`. Next, the properties of the collected constraints so far are collected and added to the set of properties (line 11). Again, new constraints that mention any of the new properties are added to `collectedProperties` as well as returned as result (line 21). This loop continues as long as there are new constraints (lines 8–22). Finally, properties and constraints that are not part of the transitive closure are removed from the (cloned) target type (line 23 and 24).

The two functions `getPropertiesFromConstraint`, `addConstraintsWithTheseProperties` are helper functions for *slice* and `removeUnusedProperties` and `removeUnusedConstraints` help with the construction of the sliced interface. Their implementation is straightforward and omitted.

7.3.4.5 Extra Info from If Statements

Earlier in this chapter, we have already explained how the implementation of flow-sensitive type analysis differs from the formalisation presented in the rule I-IfPresenceInterface (Figure 6.11). While the formalisation adds extra constraints to the interface type inside the `true` and `else` branch, TypeScript$_{\text{IPC}}$ reuses the flow-sensitive type analysis that is already built-in in the TypeScript compiler. More specifically, while type checking an identifier, TypeScript$_{\text{IPC}}$ verifies whether the property access is found inside an `if` statement that has extra information about its object. To implement this, the type checker functions for identifiers and `if` statements needs to be adapted.

Listing 7.17 on page 156 shows the changes to the implementation of the function `checkIdentifier`. If the identifier being checked in the function has an IPC-interface type (checks on lines 9–13), the type checker of TypeScript$_{\text{IPC}}$ loops through the antecedents in the history of the identifier (lines 16-36). If the identifier is found inside an `if` statement which verifies the presence of a property (lines 19–21) of which the identifier is a property as well (lines 22–29), then the presence (lines 30–31) or absence (lines 32-33) is added to the set of constraints.

Given the (possibly) extended set of constraints, the type checker uses the DPLL-solver again to verify the satisfiability of the extended set of constraints (lines 38-41). Finally, the extended set of constraints is assigned to the `constraints` property of the resolved type (line 42). This way, the extra constraints are taken into account in the rest of the type checking process.

Listing 7.18 on page 157 shows the changes to the implementation of the function `checkIfStatement`. Code with a grey background is new. Lines 3–17 check whether the `if` statement is verifying the presence of a property of IPC-interface instance. If this is the case, $\text{TypeScript}_{\text{IPC}}$ skips type checking the `if` statement (line 26) We explain why this is necessary using the code snippet in Listing 7.19.

In this function, the presence of `user_id` is verified before updating it in the `true` branch. If the type system checks the `if` statement (`pm.user_id`), this results in a type error because neither the presence nor the absence of `user_id` can be proved at this point. Therefore, type checking is skipped in this case, but $\text{TypeScript}_{\text{IPC}}$ does verify whether the property access inside the `if` statement is a property of the interface (lines 18–24).

```
1 function foo(pm: PrivateMessage) {
2   if(pm.user_id) {
3     pm.user_id = 43;
4   }
5 }
```

Listing 7.19: No type check for condition of `if` statement

Finally, constraints are reset after type checking the true (lines 32–34) and else branch (lines 36–38).

7.3.4.6 Diagnostic Messages

In each of the extensions made to the type checker of the TypeScript compiler, error messages are provided for type-unsafe code. These messages not only indicate to the user where the type error occurs, but also *why* it is not type safe. A clear error message enables the developer to quickly diagnose —and fix— the problem. In this section, we elaborate on the error messages of the compiler with regards to inter-property constraints. When there is a type error, the TypeScript compiler generates multiple error messages, from a general error message to a more detailed explanation of what is wrong.

In the following paragraphs, we discuss the most important error messages due to the incorrect use of inter-property constraints.

Unsatisfiable interface definitions For interface declarations, the type checker of TypeScript$_{\text{IPC}}$ verifies whether the constraints of an interface have at least one valid configuration of present and absent properties. The following code snippet shows an example: the interface `PrivateMessageWrong` extends the interface `PrivateMessage` with two constraints that require the presence of both user properties. This conflicts with the exclusivity constraint in `PrivateMessage`. TypeScript$_{\text{IPC}}$ throws an error message indicating that the set of constraints is not satisfiable.

```
1 interface PrivateMessageWrong extends PrivateMessage {
2
3 } constrains {
4   present(userid);
5   present(screenname);
6 }
7 /* error TS95024: Constraints of interface 'PrivateMessageWrong'
8                    have to be satisfiable */
```

As explained, TypeScript$_{\text{IPC}}$ is flow-sensitive: extra information from `if` statements on the presence or absence of properties is taken into account. This can result in an if statement where the refined interface type has unsatisfiable constraints. For example, the following code snippet shows two consecutive `if` statements that verify the presence of both `user_id` and `screen_name` in an instance of the `PrivateMessage` interface. In the true branch of the second `if` statement, the exclusivity constraint on `screen_name` and `user_id` conflict with the newly added knowledge: the presence of both `screen_name` and `user_id`. In this case, TypeScript$_{\text{IPC}}$ throws an error indicating that the constraints of the refined interface are not satisfiable.

```
1 function foo(pm: PrivateMessage) {
2   if (pm.userid) {
3     if (pm.screenname) {
4       pm.screenname;
5     }
6   }
7 }
8 /* error TS95025: Constraints in if statement are unsatisfiable
9                   because of extra knowledge from the if test */
```

Invalid Assignment from Object Literals to Interface Instances When an object literal does not satisfy the interface constraints, the error message con-

sists out of two parts: the first error message (lines 2 and 3) contains the general error message which indicates that there is an error in the assignment. The second error message (lines 4 and 5) contains the detailed information, listing exactly which constraint caused the valuation to fail.

```
1 let msg: PrivateMessage = {text: "Hello"};
2 /* error TS2322: Type '{ text: string; }' is not assignable to
3                  type 'PrivateMessage'.
4    Constraint present(userid) XOR present(screenname) was not
5     satisfied in type { text: string; }  /*
```

Invalid Property Accesses Accessing a property that is part of an inter-property constraint object fails when that property is not guaranteed to be present or absent. In that case, TypeScript$_{\text{IPC}}$ generates an error message that indicates that the required guarantees do not follow from the interface constraints. Moreover, the error message provides a hint to how the developer can resolve this error.

```
1 function getUserId(msg: PrivateMessage): number {
2   return msg.userid; //ERROR
3   /*error TS95009: Cannot access userid from the object because
4     its interface does not guarantee the presence of userid.
5     Use a non-undefined type guard.*/
6 }
```

Objupdate When multiple properties are updated simultaneously using the built-in function `objupdate` in TypeScript$_{\text{IPC}}$, it is important that all *relevant* properties are updated together. When this is not the case, this results in the following two error messages. The following code snippet shows an example where the call to `objupdate` only updates the `user_id` of a `PrivateMessage` instance. The first error message (lines 2–4) indicate that not all properties of which `user_id` is a part, are provided. The second error message shows a more detailed error message, indicating that an object literal which only contains `user_id` cannot satisfy the exclusivity constraint of `PrivateMessage`.

```
1 let msg2: PrivateMessage = objupdate(msg1, {userid: undefined});
2 /* error TS95021: All properties from the constraints in which the
3      properties of the second argument of objupdate are mentioned,
4      must be a part of the second argument
5    error TS2322: Type '{ userid: undefined; }' is not assignable
6                  to type 'PrivateMessage'.
7     Constraint present(userid) XOR present(screenname) was not
8       satisfied in type { userid: undefined; }  */
```

Interface-interface compatibility Because of the structural typing in TypeScript, there are several reasons why the assignment of one interface instance to another can fail. In the current implementation of TypeScript$_{\text{IPC}}$, the error message only indicates that the assignment failed due to the incompatible interface constraints. In the future, we plan on adding more detailed error messages, depending on whether the error occurs due to the lack of width subtyping, the differences in the property lists are incompatible interface constraints.

```
let msg1: PrivateMessageAll = {text: "Hello",
                               userid: 42,
                               screenname: "Alice"};
let msg2: PrivateMessageId  = msg1;
/* error TS2322: Type 'PrivateMessageAll' is not assignable
                  to type 'PrivateMessageId'.
    Invalid assignment of type PrivateMessageAll
                  to type PrivateMessageId.
    Check if constraints are correct*/
```

7.3.5 Emitter Extension

Syntax-wise, TypeScript$_{\text{IPC}}$ extends TypeScript in two ways: an extended interface definition and a new language construct for updating multiple properties of an IPC-interface instance. The new interface definition does not impact the emitter of TypeScript: TypeScript already removes all types from the source code, including interface definitions. This is regardless of the `constrains` extension.

However, the emitter needs to support the other extension to the TypeScript syntax: `objupdate`. Listing 7.20 on page 158 shows the extension. Recall that this function returns a copy of its first argument, in which the properties of the second argument are added or updated. We cannot translate `objupdate` to `Object.assign`, as this function updates its first argument. Instead, we use the *spread* operator from JavaScript, which is the functional equivalent of `Object.assign`. Listing 7.21 on page 158 shows an example.

7.4 Conclusion

In this chapter, we presented TypeScript$_{\text{IPC}}$, the implementation of the TIPC formalisations presented in the previous chapters. While the formalisations presented inter-property constraints in a subset of TypeScript, TypeScript$_{\text{IPC}}$ incorporated interfaces with inter-property constraints into the full TypeScript programming

language. This chapter started with a brief overview of the architecture and design of the TypeScript compiler. Next, the chapter listed the differences between the formalisation and the implementation. Finally, we highlighted the changes between the compilers of TypeScript and $\text{TypeScript}_{\text{IPC}}$. As mentioned earlier, the complete implementation of the $\text{TypeScript}_{\text{IPC}}$ compiler can be found on GitHub[6].

In the next chapter of this dissertation, we discuss work related to TIPC.

[6] `https://github.com/noostvog/TypeScriptIPC`

```
1  function checkPropertyAccessExpressionOrQualifiedName
2              (node: PropertyAccessExpression | QualifiedName,
3              left: Expression | QualifiedName,
4              right: Identifier) {
5    ...
6    let propType = getTypeOfSymbol(prop);
7    ...
8    if (node.kind == SyntaxKind.PropertyAccessExpression) {
9      if (getObjectFlags(apparentType) & ObjectFlags.Interface) {
10       const resolvedT =
11         resolveStructuredTypeMembers(<ObjectType>apparentType);
12       const constraints = resolvedT.constraints;
13       if (constraints.length > 0) {
14         const constraintStr = translateConstraints(constraints);
15         const provePresent=prove(constraintStr+">"+node.name.text);
16         const proveAbsent =prove(constraintStr+">!"+node.name.text);
17         if (!provePresent && !proveAbsent){
18           error(node, Diagnostics.CannotAccessProperty);
19           return unknownType;
20         } else if (provePresent) {
21           if (propType.flags & TypeFlags.Union) {
22             if(contains((<UnionType>propType).types,undefinedType)){
23               let propTypeClone: UnionType =
24                           <UnionType>Object.assign({}, propType);
25               propTypeClone.types =
26                 Object.assign([], (<UnionType> propType).types);
27               let filtered = propTypeClone.types.filter
28                                       (t => t !== undefinedType);
29               propType = getUnionType(filtered);
30             }
31           }
32         } else if (proveAbsent) {
33           propType = undefinedType;
34         }
35       }
36     }
37   }
38   ...
39 }
```

Listing 7.11: Definition of `checkPropertyAccessExpressionOrQualifiedName`

```
function constraintsRelatedToObjIntf
            (source: Type,
             constraints: ConstraintExpression[],
             mainLevel: boolean,
             reportErrors: boolean): Ternary {
  for (const constraint of constraints) {
    if (!isConstraintSatisfied(source, constraint, reportErrors)) {
      if (mainLevel && reportErrors) {
        reportError(Diagnostics.NotSatisfied,
                    constraintToString(constraint),
                    typeToString(source));
      }
      return Ternary.False;
    }
  }
  return Ternary.True;
}
```

Listing 7.12: Definition of `constraintsRelatedToObjIntf`

```
function constraintsRelatedToIntfIntf
            (source: Type,
             constraintsS: ConstraintExpression[],
             target: Type,
             constraintsT: ConstraintExpression[],
             reportErrors: boolean): Ternary {
  let constraintSourceStr = translateConstraints(constraintsS);
  let constraintTargetStr = translateConstraints(constraintsT);
  const sourceprops = getPropertiesOfType(source);
  const targetprops = getPropertiesOfType(target);
  const sourcepropNames = sourceprops.map((x:Symbol) => x.name);
  const targetpropNames = targetprops.map((x:Symbol) => x.name);

  for (const prop of targetprops) {
    if (sourcepropNames.indexOf(prop.name) == -1) {
      constraintSourceStr += " & !" + prop.name;
    }
  }

  for (const prop of sourceprops) {
    if (targetpropNames.indexOf(prop.name) == -1) {
      constraintTargetStr += " & !" + prop.name;
    }
  }

  if (!prove(constraintSourceStr + " > " + constraintTargetStr)) {
    if (reportErrors) {
      reportError(Diagnostics.InvalidAssignment,
                  typeToString(source),
                  typeToString(target));
    }
    return Ternary.False;
  }
  return Ternary.True;
}
```

Listing 7.13: Definition of `constraintsRelatedToIntfIntf`

```
function constraintsRelatedToIntfObj
            (source: Type,
             constraintssource: ConstraintExpression[],
             target: Type,
             reportErrors: boolean): Ternary {
  const sourceprops = getPropertiesOfType(source);
  const targetprops = getPropertiesOfType(target);
  for (const prop of targetprops) {
    if (prop.flags & SymbolFlags.Optional) {
      const names = sourceprops.map(x => x.name);
      if (names.indexOf(prop.name) == -1) {
        if (reportErrors) {
          reportError(Diagnostics.UnknownProperty,
                      prop.name,
                      typeToString(target),
                      typeToString(source));
        }
        return Ternary.False;
      }
    } else {
      const propType = getTypeOfSymbol(prop);
      const constraintStr = translateConstraints(constraintssource);
      if (propType === undefinedType){
        if (!prove(constraintStr + " > !" + prop.name)) {
          if (reportErrors) {
            reportError(Diagnostics.PropertyNotAbsent,
                        prop.name,
                        typeToString(source));
          }
          return Ternary.False;
        }
      } else {
        if (!prove(constraintStr + " > " + prop.name)) {
          if (reportErrors) {
            reportError(Diagnostics.MissingProperty,
                        prop.name,
                        typeToString(source));
          }
          return Ternary.False;
        }
      }
    }
  }
  return Ternary.True;
}
```

Listing 7.14: Definition of `constraintsRelatedToIntfObj`

```
1  function checkObjectUpdate
2              (node: ObjectUpdateExpression,
3               contextualMapper?: TypeMapper): Type {
4    ...
5
6    let target = checkExpression(args[0]);
7    const sourcepart = checkExpression(args[1]);
8    if (!(target.flags & TypeFlags.Object
9        && getObjectFlags(target) & ObjectFlags.Interface)) {
10     error(node.arguments[0], Diagnostics.FirstArgInterface);
11     return unknownType;
12   }
13   const resolvedT = resolveStructuredTypeMembers(<ObjectType>target);
14   const constraints = resolvedT.constraints;
15   if (constraints === undefined || constraints.length == 0) {
16     error(node.arguments[0], Diagnostics.SecondArgObjectLiteral);
17     return unknownType;
18   }
19
20   if (!(getObjectFlags(sourcepart) & ObjectFlags.ObjectLiteral
21       && sourcepart.flags & TypeFlags.FreshLiteral)) {
22     return unknownType;
23   }
24
25   const resolvedS =
26              resolveStructuredTypeMembers(<ObjectType>sourcepart);
27
28   let slicedTarget = cloneTarget(resolvedT);
29   slice(slicedTarget, <FreshObjectLiteralType>sourcepart);
30   const resolvedST =
31            resolveStructuredTypeMembers(<ObjectType>slicedTarget);
32   if (!sameProps(resolvedS.properties, resolvedST.properties)) {
33     error(node, Diagnostics.SlicePropsError);
34   }
35   checkTypeAssignableTo(sourcepart, slicedTarget, node);
36   return target;
37 }
```

Listing 7.15: Definition of `checkObjectUpdate`

```
1   function slice(target: ResolvedType,
2                  sourcePart: FreshObjectLiteralType): ResolvedType {
3     let constraints = target.constraints;
4     let collectedConstraints=createNodeArray<ConstraintExpression>();
5     let collectedProperties= Object.assign([],sourcePart.properties);
6     let result =
7               addConstraintsWithTheseProperties(collectedProperties);
8     while (result.length !== 0) {
9       let moreProperties: Symbol[] = [];
10      for(const constraint of result) {
11        let props = getPropertiesFromConstraint(constraint);
12        for (const prop of props) {
13          const symbol=createSymbol(SymbolFlags.Property,prop.text);
14          if (collectedProperties.map(x => x.name).indexOf(prop.text)
15                                                              == -1) {
16            collectedProperties.push(symbol);
17            moreProperties.push(symbol);
18          }
19        }
20      }
21      result = addConstraintsWithTheseProperties(moreProperties);
22    }
23    target = removeUnusedProperties(target, collectedProperties);
24    target = removeUnusedConstraints(target, collectedConstraints);
25    return target;
26  }
```

Listing 7.16: Definition of `slice`

```
function checkIdentifier(node: Identifier): Type {
  const symbol = getResolvedSymbol(node);
  if (symbol === unknownSymbol) {
    return unknownType;
  }
  ...
  const flowType = getFlowTypeOfReference(node, ...);
  ...
  if (getObjectFlags(flowType) & ObjectFlags.Interface) {
    const resolvedT =
                resolveStructuredTypeMembers(<ObjectType>flowType);
    let constraints = Object.assign([], resolvedT.origConstraints);
    if (constraints.length > 0) {
      if (node.flowNode) {
        let ant: any = node.flowNode;
        while (ant) {
          if (ant.flags & FlowFlags.Condition) {
            const propacc = ant.expression;
            if(propacc.kind===SyntaxKind.PropertyAccessExpression){
              if((<PropertyAccessExpression>propacc).expression.kind
                      === SyntaxKind.Identifier) {
                if ((<Identifier>(<PropertyAccessExpression>propacc).
                     expression).text === node.text) {
                  const s1 = getResolvedSymbol(<Identifier>
                    (<PropertyAccessExpression>propacc).expression);
                  const s2 = getResolvedSymbol(node);
                  if (s1 === s2) {
                    const presentC = createPresentConstraint(
                        (<PropertyAccessExpression>propacc).name);
                    if (ant.flags & FlowFlags.TrueCondition) {
                      constraints.push(presentC);
                    } else if(ant.flags&FlowFlags.FalseCondition){
                      constraints.push(createNotConstraint(presentC));
          } } } } } }
          ant = ant.antecedent;
        }
      }
      if (!Proplog.solve(
                 translateConstraints(constraints, "-"), "none")) {
        error(node, Diagnostics.NotSatisfiableInIf);
      }
      resolvedT.constraints = constraints;
    }
  }
 return assignmentKind
          ? getBaseTypeOfLiteralType(flowType)
          : flowType;
}
```

Listing 7.17: Definition of `checkIdentifier`

```
function checkIfStatement(node: IfStatement) {
  checkGrammarStatementInAmbientContext(node);
  let specialCosntraintIf = false;
  if (node.expression.kind
                    === SyntaxKind.PropertyAccessExpression){
    const paeExpression: PropertyAccessExpression =
                 <PropertyAccessExpression>(node.expression);
    const paeExpressionType =
                   checkExpression(paeExpression.expression);

    if (getObjectFlags(paeExpressionType)
                                    & ObjectFlags.Interface) {
      let objectType = resolveStructuredTypeMembers(
                               <ObjectType>paeExpressionType);
      specialConstraintIf = objectType.constraints !== undefined
                           && objectType.constraints.length > 0;
    }
    if (specialConstraintIf) {
      const accessedProperty = paeExpression.name.text;
      const prop = getPropertyOfType(paeExpressionType,
                                      accessedProperty);
      if (!prop) {
        error(paeExpression, PropertyDoesNotExist);
  } } }
  if (!specialConstraintIf) {
    checkExpression(node.expression);
  }
  checkSourceElement(node.thenStatement);
  if (node.thenStatement.kind === SyntaxKind.EmptyStatement) {
    error(node.thenStatement, Diagnostics.EmptyBody);
  }
  if (specialConstraintIf) {
    objectType.constraints = objectType.origConstraints;
  }
  checkSourceElement(node.elseStatement);
  if (specialConstraintIf) {
    objectType.constraints = objectType.origConstraints;
  }
}
```

Listing 7.18: Definition of `checkIfStatement`

```
function emitObjectUpdateExpression(node: ObjectUpdateExpression) {
    write("{ ...");
    emitExpression(node.arguments[0]);
    write(", ...");
    emitExpression(node.arguments[1]);
    write("}");
}
```

Listing 7.20: Definition of `emitObjectUpdateExpression`

```
//The following TypeScriptIPC code
let pm : PrivateMessage = {text: "Hello!", user_id: 42};
let pm2: PrivateMessage =
       objupdate(pm, {user_id: undefined, screen_name: "Alice"};
// is emitted as:
var pm  = {text: "Hello!", user_id: 42};
var pm2 = {...pm, ...{user_id: undefined, screen_name: "Alice"}};
// which results in the following value for pm and pm2:
{text: "Hello!", user_id: 42} //pm (unchanged)
{text: "Hello!", user_id: undefined, screen_name: "Alice"} //pm2
```

Listing 7.21: Example of emitting an `objupdate` call

Chapter 8

Related Work

Until now, this dissertation has introduced a new programming language called TIPC. This programming language distinguishes itself from existing programming languages with its unconventional interface definitions. Instead of limiting constraints on the presence of properties to a per-property basis, TIPC allows the combination of presence constraints on different properties. Constraints are combined using operators from propositional logic, and the type system of TIPC uses concepts from propositional logic to statically guarantee that the constraints uphold.

In this chapter, we situate this work in past and ongoing research. More specifically, we discuss several other research domains in type systems that are related to our work.

8.1 Dependent Types

In the field of research on type systems, new research efforts push boundaries by making static types as expressive as possible from a decidability point of view. Dependent type systems are the most expressive kind of type systems: a type in a dependently typed programming language may depend on *values*. However, this expressivity comes with a price: decidability. After discussing dependent types in more detail, the next section will discuss refinement types. Refinement types are a light-weight variant of dependent types, which trade a bit of expressivity for decidability.

[8] Related Work

In a dependently typed programming language, types are parametrised over values [Martin-Löf, 1975]. Examples of dependent types are:

- `let x: {v:number | 0 < v}` to indicate that a variable may only contain numbers greater than zero;
- `let x: array[n]` to indicate that an array must have exactly `n` elements;
- `function append(arr1: array[X], arr2: array[Y]): array[X+Y] { ... }` to define a function that appends two arrays of a certain length. The return type is the accumulation of both lengths;

These examples are all small in size, but dependent types can be used to impose very complex constraints. For example, the work of Bove and Dybjer [2009] shows how dependent types can be used to define the type of a sorted list (where the type ensures the list is sorted) and a binary search tree (where the type ensures that the binary search tree condition is satisfied). Work by Ek et al. [2009] explains how red–black trees (balanced binary trees) can be implemented in the dependently typed programming language Agda.

Both Howard [1980] and De Bruijn [1970] extended the simply typed lambda calculus with support for dependent types. Howard extended work by Curry [1934], who defined the correspondence between propositional logic and types: to prove a mathematical proposition, its corresponding type must be inhabited (there must exist a value with that type). Later, Howard [1980][1] extended this correspondence with predicate types. As propositions are types, predicates become *dependent types*, where the type depends on something. For example, the type `array[X]` depends on the value of `X`. In 1975, Martin-Löf [1975] incorporated the Curry-Howard correspondence into type theory, called *intuitionistic type theory*. Independently, De Bruijn [1970] created the mathematical language AUTOMATH, which employs a dependent type system.

There exist several languages that implement intuitionistic type theory. Some of these languages focus on theorem proving, resulting in proof assistants such as Coq [Barras et al., 1997] and NuPRL [Constable et al., 1985]. Other languages focus more on the programming aspect, resulting in dependently typed programming languages such as Agda [Norell, 2007], Cayenne [Augustsson, 1998], Epigram [McBride, 2005], Idris [Brady, 2011] and F* [Swamy et al., 2011].

As types in dependently-typed programming languages are very expressive, there is a wide range of errors than can be caught at compile time. However, this comes at the cost of decidability. Some languages, such as Cayenne, preserve the

[1] The original manuscript was written in 1969.

full expressivity of their programming language but remain undecidable. Other languages choose to apply restrictions on either the expressiveness of the type or the language.

As dependently-typed programming languages are very expressive, they are capable of expressing constraints between properties of an object. However, incorporating the full power of dependent typing in existing programming languages is an active topic of research [Eisenberg, 2016; Kent and Tobin-Hochstadt, 2015]. Almost all dependently-typed programming languages are functional and have severe restrictions, such as total functions and complex proof annotations. On the other hand, TIPC does not need the full expressive power of dependent typing. By extending the type system only with support for inter-property constraints, we are able to incorporate the extensions in an imperative programming language without requiring complex proof annotations.

8.2 Refinement Types

The power of dependently typed programming languages comes with a price: in order for the type system to prove type safety, functions have to be total and developers need to provide complex proof annotations. Refinement types [Owre et al., 1998; Xi and Pfenning, 1998] try to find a balance between the power of the type system and the annotation effort that is needed from the developer in a dependently typed programming language. More specifically, the types in a refinement type system are limited to decidable predicate logic.

For example, Freeman and Pfenning [1991] limit refinements on types to refinements on the structure of algebraic datatypes. Liquid types by Rondon et al. [2008] (short for Logical Qualified Data types) allow all refinements that are expressible in predicates of decidable logic. Restricting the expressiveness of refinement on types enables the use of an SMT (Satisfiability Modulo Theories) solver to automatically deduce the satisfiability of a first-order logic formula. An SMT solver is a generalisation of a SAT solver: the latter works only on boolean expressions while the former works on predicates from a higher-level theory. By using an SMT solver, the programming language removes the proof annotation burden from the programmer.

Refinement types can be used to refine types to specific subsets like “all even numbers” or “strings of length N”, but other research tracks focus on using refinement types for specific purposes. For example, Xi and Pfenning [1998] use refinement types to eliminate dynamic array bound checking. Kawaguchi et al. [2009] use refinement types to verify complex data structure invariants, such as the red–black invariant of red–black trees and the binary search invariant in a tree.

Bengtson et al. [2011] verify authentication properties of cryptographic protocols with a refinement type system.

The approach of refinement types is very general: any type can be refined using predicates, as long as its satisfiability is deducible with SMT solvers. In this respect, TIPC differs from refinement type languages: only interface types can be refined in TIPC and only on the presence of the properties. By limiting the expressiveness of refinements in TIPC, we aim to minimise the impact of the advanced type system to the developer. It also facilitates maintaining static decidability: as refinements in TIPC are only imposed on the presence of properties, logical entailment and SAT solvers suffice to type check TIPC programs.

Several research efforts have a different approach regarding the tractability of refinement type languages. Hybrid type checking by Flanagan et al. [2006] combines a refinement type system with dynamic contract checking. Whenever the static type system cannot ensure type safety, hybrid type checking inserts dynamic checks. In F*, Swamy et al. [2016] allow developers to switch to manually providing proofs whenever the required proofs exceed the capability of the SMT solvers. Finally, Lehmann and Tanter [2017] introduce the concept of gradual typing into refinement type languages. Because of the limited expressiveness, TIPC does not have the decidability issues present in refinement type languages. Incorporating gradual types into TIPC is an interesting research avenue that falls out of scope of this dissertation. Chapter 10 contains a detailed discussion of gradually typed inter-property constraints.

Besides research on new languages and new applications for refinement types, a separate research track focuses on the practical implementation of refinement types in existing programming languages. Xi and Pfenning [1999] incorporates refinement types in the ML programming language. Another research effort includes refinement types in a functional dynamic programming language [Chugh et al., 2012b]. Vazou et al. [2014] propose a type system for refinement types in Haskell, Kent et al. [2016] incorporate refinement types in Racket and Vekris et al. [2016] integrates refinement types in TypeScript.

In the rest of this section, we will discuss refinement types to the applications that are close to our work in detail: research efforts that incorporate refinement types into imperative dynamic programming languages and object-oriented programming languages.

8.2.1 Refinement Types For Dynamic and Object-Oriented Programming Languages

Hoop Flanagan et al. [2006] introduce Hoop, a hybrid object-oriented program-

ming language with refinement types and object invariants. Hoop aims to provide a type system that allows for very expressive object interfaces. As they allow any boolean predicate as a refinement of a type, it is possible to define presence constraints between object properties. However, Hoop requires refinements and object invariants to be pure, which means that refinement cannot be imposed on mutable object properties. The refinement language on objects in TIPC is less expressive, and caters to constraints often found in API documentation. However, this allows for *mutable* object properties in TIPC, on which refinements may be expressed. As Hoop is a hybrid language, refinements that cannot be checked statically are translated into dynamic checks. TIPC limits the refinement on object interfaces, eliminating the need for dynamic checks.

X10 Nystrom et al. [2008] introduce the statically typed object-oriented programming language designed for concurrent and distributed systems, called X10. This programming language supports constrained types, which is a form of dependent types. Similar to TIPC, constraints can be imposed on the state of objects. However, the main difference between constraints in X10 and TIPC is the expressivity of the constraints: constraints in X10 can be any boolean predicate expression on the value of fields. This comes at the price of immutability: constraints in X10 can only reference immutable fields. While constraints in TIPC only impose restrictions on the presence and absence of properties, it is possible to impose constraints on mutable properties of an object.

RSC Vekris et al. [2016] introduce RSC, a lightweight refinement system for TypeScript. Refinements in RSC stem from an SMT-decidable logic, and can be imposed on any type, including classes and class fields. As a result, constraints between class fields can be written down in RSC. However, RSC imposes the same restrictions as we have already seen in HOOP and X10: RSC guarantees soundness by restricting refinements to only concern the values of immutable fields. Again, this differs from TIPC where refinement may only concern the presence or absence of properties, but can be imposed on mutable fields.

After presenting the formalisations of RSC, Vekris et al. elaborate on more features of TypeScript could be incorporated in RSC. They show how information from `if` statements can be incorporated in the refinements of types. The same idea is incorporated in the formalisations of TIPC. They also elaborate on their immutability restriction for refinements. By introducing a new kind of mutability called `Unique`, the type system would have to guarantee that there is only one reference to a certain object. This way, the type system would be able to safely allow mutations on fields that are part of a refinement type. A similar approach

could alleviate restrictions on the simultaneous update of properties part of an inter-property constraint. We elaborate on this in Section 10.3.

DJS Finally, we elaborate on Dependent JavaScript (DJS), by Chugh et al. [2012a]. DJS is a statically typed variant of a subset of JavaScript, which includes refinement types and strong updates. As with other refinement type systems, refinements in DJS are based on an SMT-decidable logic. DJS also supports strong updates, which allow an assignment to change the type of a reference. This is only possible with a form of Alias Types (by Smith et al. [2000]), which ensure that there is only one reference to a location in the heap. Using the combination of refinement types and strong updates, it is possible in DJS to express inter-property constraints using refinements in DJS, even on mutable properties. A disadvantage that stems from this expressiveness is the required annotations throughout the program to accommodate the alias typing system in DJS.

Listing 8.1 shows how inter-property constraints can be expressed in DJS. It shows a function definition for `exactlyOne`, which receives one argument, an object. This object should either contain either a `user_id` property or a `screen_name` property. The comment block between the function parameter list and the function body contains the function type. The function definition is first parametrised with two location variables `LL1` and `LL2` and one heap variable `Heap`. It states that this function takes one argument called `obj` with the type indicating the location of the object on the heap. This heap object has a single prototype object (in location LL2). The prototype object of `obj` contains a dictionary. This dictionary is refined with a double implication (`iff`) indicating that if the prototype object (on location `LL2` in the heap) contains `user_id`, it may not contain a `screen_name`, and the other way around. At the end, we indicate that the prototype object of `obj` lives at the location `LL2` (`> LL2`). As this function has an empty function body, the function returns the top type and nothing changes in the heap.

```
1  var exactlyOne = function(obj) /*: [;LL1,LL2;Heap]
2      (obj:Ref(LL1))
3    / Heap +
4      (LL1 |-> _:{Dict|
5        (iff
6          (objhas v "user_id" Heap LL2)
7          (not (objhas v "screen_name" Heap LL2)))} > LL2)
8   -> Top / same */
9  {
10 };
```

Listing 8.1: Inter-property constraints in DJS

As can be seen in Listing 8.1, writing down the type of an object with inter-property constraints is not trivial. In order to express the type, developers need to have knowledge of the way objects are represented in a heap. Moreover, the required details on the heap representation in the type of a function are cumbersome to write down. In the paper, the authors of DJS acknowledge this significant annotation overhead. TIPC, on the other hand, aims to be a lightweight extension of TypeScript and requires programmers to only have knowledge about interfaces and basic propositional logic. TIPC achieves this by having a less expressive constraint language than DJS. However, constraints in TIPC suffice to express almost all of the constraint found in web APIs and web API documentation.

Important to note is that the alias types in DJS have an impact on *all* the aspects of the programming language. Although Listing 8.1 only shows the definition of a function with inter-property constraints, the entire program needs to be extended with annotations with regards to the heap. This is in contrast with TIPC, in which we aim to have a minimal impact on the syntax of TypeScript. This promotes the integration of inter-property constraints in existing TypeScript programs. In the future, we plan on extending TIPC with support for an imperative simultaneous update of properties while keeping these goals in mind, by incorporating a limited version of alias types. We elaborate on this in Section 10.3.

A note on contract programming Contracts are a way to impose preconditions and postconditions on functions [Findler and Felleisen, 2002]. Research on contract systems can be divided into two categories: latent contracts and manifest contracts. In a latent contract system, only *dynamic* checks are generated. Manifest contracts systems also generate dynamic checks, but they incorporate the guarantees given by these dynamic checks into the static analysis as well. Manifest contract systems are very closely related to work by Greenberg et al. [2010], Ou et al. [2004] and the research on hybrid refinement type checking mentioned earlier in this section, where the parts that cannot be proven with an SMT-solver are checked dynamically. We do not elaborate on latent contract systems, as they do not involve any static analysis.

Conclusion To conclude, we discussed two refinement type systems for object-oriented programming languages (Hoop and X10), one refinement type system for TypeScript (RSC) and one refinement type system for JavaScript (DJS). Although all of these refinement type systems are intrinsically able to express inter-property constraints, there are also many differences between these languages and TIPC. We group the different languages by their approach:

- The programming languages Hoop, X10 and RSC impose the same restriction on refinements: they only support inter-property constraints in the initialisation phase: updating properties that are part of inter-property constraints is impossible. By limiting refinements to the presence and absence of properties, TIPC is able to allow refinements without sacrificing mutability for single-property updates.

- The fourth programming language we discussed was DJS, which has a refinement type system rich enough to express inter-property constraints and allow mutability. This is achieved by its form of alias tracking, which ensures that there is always only one reference to an object. We have shown an example in which inter-property constraints are expressed in DJS, but which also shows the disadvantage of DJS: its complex type annotation language, which needs to be provided throughout the *entire* program. This is in sharp contrast to TIPC, where a simple constraint logic is incorporated into interface definitions.

- A final difference between these refinement type systems and TIPC is how the type system verifies valid subtypings. Just like TIPC, regular refinement type systems also use logical entailments or logical implications to verify a subtyping or assignment compatibility relationship. As the focus of refinements in TIPC is on the presence and absence of properties, TIPC adds additional inferred logical propositions on present or absent properties to both sides of the entailment. This way, more subtyping and assignment compatibility relationships will be accepted by the type checker, improving the usability of refined types for programmers.

Table 8.1 gives an overview of the trade-offs made in refinement type systems and the type system of TIPC. The table evaluates the type systems on three aspects, where TypeScript forms the baseline:

- the power of the type system: evaluated according to the power of the logic that can be expressed in type definitions;

- the expressivity of the programming language, compared to the expressivity of TypeScript;

- the type annotation overhead: we consider type annotations an *overhead* when they do not directly deal with the definition of documentation constraints.

	Power Type System	Language Expressivity	Annotation Overhead
Hoop	++	− −	=
X10	++	− −	=
RSC	++	− −	=
DJS	++	=	− −
TIPC	+	−	=
TypeScript	=	=	=

Table 8.1: Comparison between refinement type systems, TIPC and TypeScript

Refinement type systems enable the definition of types which can be expressed in decidable predicate logic. In Hoop, X10 and RSC there is no annotation overhead, but they prohibit mutability on objects on which refinements are imposed. DJS, on the other hand, does not limit the expressivity of the programming language which comes at the price of annotations with regards to the heap throughout the program. Types in TIPC are tailored to the expression of inter-property constraints and are thus less powerful than refinement types. This is an improvement over the type system of TypeScript without requiring extra annotations or disallowing mutable objects.

8.3 Type Systems for TypeScript

Chapter 5 already introduced the TypeScript programming language, which is an optionally typed superset of JavaScript. Because of the lack of support for inter-property constraints in TypeScript, this dissertation introduces a variant of TypeScript, called TIPC.

In recent years, several research efforts have proposed a static type system for (a subset of) JavaScript. The different type systems for JavaScript focus on different key features of JavaScript or cover a different subset of the programming language. In this section, we give an overview of these type systems.

In recent years, several formalisations for TypeScript have been proposed:

FTS As already mentioned earlier in previous chapters of this dissertation, TIPC is an extension of earlier work by Bierman et al. [2014]. In this paper, the sound and unsound features of TypeScript are formalised into two calculi: safeFTS and prodFTS. The first calculus focuses on supporting the sound subset of TypeScript, including features such as contextual typing and the lack of block

scoping in JavaScript. The second calculus, prodFTS is an extension of safeFTS which includes unsound features of TypeScript, such as unchecked downcasts and unchecked indexing. FTS stays true to the optional typing of TypeScript: they ensure a full erasure of the type annotations in the compiled JavaScript code.

Safe TypeScript Rastogi et al. [2015] introduce a *safe* compilation mode for TypeScript, for which soundness is guaranteed. Safe TypeScript is a variant of TypeScript that has a gradual type system, instead of the default optional type system in TypeScript. This is achieved by generating runtime type checks in dynamically typed code (which uses type `any`), or where the boundary between dynamically typed code and statically typed code is crossed. The type system of Safe TypeScript employs runtime type information from the generated checks and existing checks. Although Safe TypeScript generates runtime checks, Rastogi et al. ensure that the runtime type information from the generated runtime checks does not break the underlying JavaScript semantics.

StrongScript Richards et al. [2015] also introduce a gradually typed variant of TypeScript, called StrongScript. In StrongScript, there are three kind of types: dynamic types (`any`), optional types and *concrete types*. Optional types employ the same behaviour as the types in regular TypeScript. Concrete types add a new kind of type to TypeScript that has stronger guarantees than optional types: StrongScript guarantees that at runtime, these types only contain values of those types. Optional types, on the other hand, cannot uphold this guarantee because of the unsound features in TypeScript, such as unchecked casts.

Summary We have discussed three programming languages related to TypeScript: FTS, Safe TypeScript and StrongScript. FTS served as the starting point of the programming language in this dissertation. It aims to be a true representation of the syntax, semantics and type system of TypeScript, including its optional types and full erasure of type after compilation. TIPC is based on the sound subset of FTS. Safe TypeScript and StrongScript focus on improving the combination between sound and unsound parts of TypeScript. In order to increase the static guarantees, Safe TypeScript as well as StrongScript incorporate a gradual typing system into TypeScript. All of these approaches are orthogonal to the goal of this dissertation: we do not focus on the optional or gradual type system of TypeScript. Instead, we extend the type system of TypeScript with a more expressive kind of interface definition.

8.4 Type Systems for JavaScript

The challenge of TypeScript (the basis for TIPC) was to retrofit a statically typed language onto an until then dynamically typed language. Dynamically typed languages are very permissive, and JavaScript is a prime example of that. When designing a static type system for JavaScript, there must be a careful consideration of which dynamic features of JavaScript need to be supported.

There already exist many research efforts in static analysis and type systems for JavaScript, such as Gardner et al. [2012]; Guha et al. [2011]; Jang and Choe [2009]; Jensen et al. [2009]; Lerner et al. [2013]; Politz et al. [2012, 2014]. We limit our discussion to research in the type system domain that focuses on the dynamic nature of objects in JavaScript:

$\mathbf{JS_0^T}$ Early work in type systems for JavaScript was done by Anderson et al. [2005], who introduce a type inferencer for JavaScript. The focus of their statically typed version of JavaScript, called $\mathrm{JS}_0^{\mathrm{T}}$, is on the dynamic features on objects in JavaScript: the dynamic addition of fields to objects and the dynamic reassignment of fields and methods in an object. This is achieved by having two kinds of object members: definite object members and potential object members. Potential objects members become definite when they have a value assigned to them. This allows the type system to only allow the access of properties to those that already have received a value.

The work of Anderson et al. is extended by Zhao [2010]. They reuse the potential and definite object members to support implicit object extensions, i.e. when a function call has dynamic object extensions as a side effect. Moreover, they support strong updates (where the type of an object member may change throughout the program), but only for recently created objects.

Core JavaScript Around the same time as Anderson et al., Thiemann [2005] presented a type system for JavaScript that also focuses on the dynamic features of JavaScript. More specifically, Thiemann present a statically typed programming language called Core JavaScript, which prevents accessing undefined properties and identifies silent type conversions. To achieve this, Core JavaScript uses singleton types and first-class record types.

$\mathcal{RAC}$ Heidegger and Thiemann [2010] introduce *recency types* for JavaScript. In this work, the authors propose a static type system for JavaScript that supports the common *initialisation phase* of objects in JavaScript, where an empty object is extended with properties shortly after its creation. Their recency-aware calculus

($\mathcal{RAC}$) supports these dynamic additions of properties in the initialisation phase through a flow-sensitive type system and strong updates for objects. After the initialisation phase, the type system falls back to flow-insensitive typings.

SJS Choi et al. [2015] introduce SJS, a programming language with a type inferencer for a significant subset of JavaScript. In order to achieve efficient compiling, they introduce a type system that statically guarantees a fixed object layout. Choi et al. statically ensures that JavaScript objects with prototype inheritance are used in a type-safe manner: fields present in the object or in its prototype chain can be accessed, fields present in the object can be modified.

Chandra et al. [2016] propose an extension of SJS, in which they introduce a type inferencer for SJS. Additionally, they have support for abstract objects, first-class methods and recursive objects. Using rows in object types, Chandra et al. have a more fine-grained way to track properties in the object and prototype chain. Their abstract types ensure that the type system limits unsafe property accesses.

Flow Chaudhuri et al. [2017] introduce Flow, a static type checker for JavaScript. Flow inferences types and refines types using a flow-sensitive analysis, even for mutable variables. In order to ensure type safety, type refinements are invalidated after an assignment (which are tracked with effects) and type refinement of object properties are invalidated conservatively (Flow is not heap sensitive).

Conclusion there are already many research efforts that focus providing a type system for objects in JavaScript. In this dissertation, we do not describe a type system to verify existing JavaScript object features. Instead, we improve the expressivity of a type system for TypeScript by introducing a new kind of interface definition. This also requires advanced rules for property accesses and updates by the type system to guarantee type soundness.

8.5 Occurrence Typing

In TIPC, interface types with inter-property constraints can be narrowed: when an `if` statement verifies the presence or absence of a property, this information is taken into account in the true and else branch. This new information *narrows* (or *strengthens*) the type: given the extra information, TIPC can construct a more detailed type representing the object. As a consequence, different occurrences of the same identifier may have different types throughout the program. This idea

is also known as *occurrence typing*. In this section we describe the origin and evolution of occurrence typing.

The concept of occurrence typing was first introduced by Tobin-Hochstadt and Felleisen [2008], and later on refined [Tobin-Hochstadt and Felleisen, 2010]. In the first paper, Tobin-Hochstadt and Felleisen extended the type system of Typed Scheme with occurrence typing. The second paper highlights some weaknesses in this approach and presents a more powerful version that explicitly supports logical combinators in conditionals and increases expressivity by supporting structure selectors.

Occurrence typing also appears in other research, such as λ_S: a scripting programming language with a flow typing system by Guha et al. [2011]. It focuses on characteristics typical in scripting languages, such as imperative features. Kashyap et al. [2013] introduce a static analysis for JavaScript with type refinements. They especially focus on implicit assumptions made for JavaScript operations, such as undefined and null checks and primitive or object checks.

Modern type systems for programming languages with dynamic type checks support occurrence typing from the beginning. As we have already covered in Section 5.6, TypeScript narrows types inside conditional branches, as well as a form of flow analysis with respect to `return` statements. Hack [Facebook Inc, b], is a dialect of PHP with a gradual type system which also takes information from conditionals into account when type checking the branches. Flow [Facebook Inc, a], a static type checker for JavaScript, also supports type refinements [Chaudhuri et al., 2017]. Bonnaire-Sergeant et al. [2016] introduce Typed Clojure which adds optional types to Clojure. They adapt occurrence types to fit the features characteristic to Clojure.

Several of the refinement type languages we have seen earlier in this chapter narrow refinement types according to conditional branches: Freeman and Pfenning [1991], Rondon et al. [2008], Chugh et al. [2012b], Chugh et al. [2012a] and Vekris et al. [2016] among others.

8.6 Conclusion

In this chapter, we briefly summarised other research that aims to provide complex types to impose advanced constraints on expressions.

Dependent type systems are the most expressive form of type systems, where types may be predicated on values. However, this expressivity comes with the price of decidability, requiring the programmers to provide proofs themselves. A lightweight form of dependent types, called refinement types, strive to find a balance between expressive types and decidability. Benefitting from the tremendous

progress made in the field of SMT solvers, refinement types are limited to logic that is SMT-decidable. Incorporating refinement types into imperative object-oriented programming languages either requires limitations on the expressivity of the language, or complex type definitions.

TIPC can be classified as a *lightweight refinement type system*, as it only imposes constraints on the presence of expressions. It presents a low barrier to entry for developers that want to augment existing (imperative) programs to support inter-property constraints, without sacrificing expressivity of the language or simplicity in the type definition.

TIPC is a variant of TypeScript, which adds optional types to JavaScript. Throughout the years, several type systems for JavaScript were proposed, each focusing on a subset of JavaScript with characteristic JavaScript features. The focus of these proposals is statically supporting existing JavaScript features, while the type system of TIPC adds a new feature that is applicable to all object-oriented programming languages. More recently, several formalisations for TypeScript and improvements thereupon have been proposed. The formalisations of TIPC are based upon work by Bierman et al. [2014], while TIPC is orthogonal to other work on TypeScript that focuses on the gap between type safe and type unsafe code. Finally, we highlighted existing work on occurrence typing. TIPC uses occurrence typing to support property accesses and updates guarded by presence tests.

Chapter 9

Inter-property Constraints in Practice

This dissertation has introduced the concept of inter-property constraints. In Section 2.3.2, we have showed the relevance of inter-property constraints with an empirical study in the documentation of web APIs. Subsequently, we have incorporated inter-property constraints into TypeScript, a statically typed programming language often used for developing web applications.

Recall that the original motivation for inter-property constraints stemmed from the documentation of APIs, and web APIs in particular. These APIs are either specified in textual documentation or using a machine-readable specification language, from which human-readable documentation can be generated, but also automated tests for APIs and code. Incorporating inter-property constraints into machine-readable web API specification languages has a great impact on several facets of the development of web applications. Ed-douibi et al. [2018] confirm that the lack of support for inter-property constraints in specification languages causes inter-property constraints to remain unchecked in automated tests. In this chapter, we investigate how the concept of inter-property constraints can be incorporated in machine-readable specification languages and their tools.

We start with an overview of existing specification languages (Section 9.1) and what constructs they provide to express inter-property constraints (Section 9.2) Section 9.3 presents a new specification language with special constructs to express common kinds of inter-property constraints. Finally, this chapter elaborates on how incorporating inter-property constraints into specification languages also benefits the accompanying tools. This first three sections of this chapter are an extension of Oostvogels et al. [2017] (sections 3 and 4): this chapter also covers relevant features that have been introduced in API specification languages since the publication of the paper.

9.1 Web API Specification Languages

Web API specification languages are machine-readable languages used to formalise the specifications of web APIs. Machine-readable specification languages for web APIs have been around since 2000, with the introduction of WSDL (Web Services Description Language, Christensen et al. [2001]). Since then, many new languages have emerged, such as WADL (Web Application Description Language, Hadley [2006]), OpenAPI specification (formerly known as Swagger), API Blueprint and RAML (RESTful API Modeling Language). Many of the API specification languages use prevalent markup languages such as XML, JSON , MSON (Markdown Syntax for Object Notation) and YAML as their syntax.

A specification file contains a machine-readable version of the web API documentation. The details vary per specification language, but the overall structure information encoded in a specification file is roughly the same. In general, a specification file contains a part with metadata about the web API (such as versioning, licensing and hosting information) and a list of entry points together with their details. For every entry point, a specification file contains a description and a list of operations. Each operation is characterised by an HTTP verb (GET,POST, ...) and sometimes specific headers sent by a client (`Accept`). Per operation, the specification contains a list of parameters and a definition of its response.

For every parameter, a specification file details its name, a description and a set of imposed constraints. Examples of constraints that can be imposed on parameters are:

- type;
- required or optional;
- minimum length and maximum length;
- minimum value and maximum value;
- uniqueness of items in array.

All the constraints listed above are constraints about one single parameter. However, some specification languages do have language constructs that allows the specification of constraints between parameters as well. In the next section, we will discuss these in detail.

9.2 Inter-property Constraints in Specification Languages

In this section, we will discuss language constructs of API specification languages that allow the expression of constraints *between* parameters (or properties). We have investigated nine web API specification languages: WSDL, WADL, OpenAPI, MSON, RAML, JSON Schema, WifL [Danielsen and Jeffrey, 2013], Web IDL [Bae et al., 2014] and hRESTS [Kopecký et al., 2008]. The first five languages are API specification languages that are commonly used by API providers, while the latter three specification languages are academic research artefacts. We also include JSON Schema in our discussion — even though it only validates one object at a time — because it supports some inter-property constraints.[1]

Our investigation shows that only two out of these specification languages — namely JSON Schema and OpenAPI — supports inter-property constraints at all. The rest of this section will discuss the language constructs found in OpenAPI or JSON Schema that support inter-property constraints.

9.2.1 oneOf (OpenAPI specification, JSON Schema)

The first language construct we will discuss is `oneOf`, which is part of the OpenAPI specification as well as JSON Schema. This language construct receives an array of JSON schemas (a collection of requirements) and requires that exactly one of those schemas is valid for the given object. The following code snippet shows an example of `oneOf`: in order to satisfy this JSON schema, a JavaScript instance must be an object with a property user that either has type `string` or `number`.

```
1 { oneOf: [
2     { type: "object",
3       properties: {user: {type: "string"}}},
4     { type: "object",
5       properties: {user: {type: "number"}}}]}
```

Listing 9.1: Example of `oneOf` for exclusivity constraints

At first, the `oneOf` construct appears suitable for exclusivity constraints (defined in Section 2.1.1), but we show that this is not the case with a counterexample in Listing 9.2. This example attempts to encode the exclusivity constraint found in the Twitter documentation on private messages, where an object may only contain either a `screen_name` or a `user_id`. For objects that contain one of the

[1] JSON Hyper-Schema is an extension of JSON Schema for describing APIs. However, it does not add additional expressiveness for describing web APIs.

```
1 { oneOf: [
2    { type: "object",
3      required: ["screen_name"],
4      properties: {"screen_name": {"type": "string"}}},
5    { type : "object",
6      required: ["user_id"],
7      properties: {"user_id": {"type": "number"}}}]
8 }
```

Listing 9.2: Attempt at using `oneOf` for exclusivity constraints

properties, such as {`user_id: 42`} and {`screen_name: "Alice"`}, this JSON schema works as expected.

However, an object literal with both properties such as {`screen_name:42, user_id:42`} would be accepted as well: the `screen_name` property is not a string, therefore the first schema is not considered valid, and therefore the `oneOf` constraint passes! This is not a good fit for the exclusivity constraints found in web APIs: we want to ensure that exactly one of the properties is present.

9.2.2 discriminator (OpenAPI specification)

The next language construct we will discuss is the `discriminator`, provided by the OpenAPI specification. Using a `discriminator`, developers can change the requirement for a JavaScript instance, given a certain value. The `discriminator` construct can be used to express present-value dependency constraints (defined in Section 2.1.2.2), where the presence of a property depends on the value of another property.

A `discriminator` expects two properties: `propertyName` to indicate which property will contain the value and `mapping` which is an array that contains a mapping of strings onto JSON schemas. Depending on the `propertyName`, JSON Schema will pick the correct schema from `mapping` and validate the JavaScript object against that schema.

Listing 9.3 shows a translation of a value-present dependency constraint to OpenAPI with a `discriminator`. In the Google Maps API, the `infoWindow` property is only taken into account when the property `suppressInfoWindow` is set to `true` when rendering directions. To express this, the OpenAPI specification has to contain two schemas. The first schema defines the base components of the API entry point (we have limited the property list to three of the properties). Using the `discriminator`, we indicate that an extra schema has to be used whenever the value of `infoWindow` is `true`. This extra schema `Extra` contains an `allOf`

construct to combine the properties of the original schema with a new schema that only contains the `infoWindow` property.

Translating present-value dependency constraints to an OpenAPI specification using `discriminator` constructs is very unwieldy. Although this example only contains one dependency constraint, it is already very verbose. In combination with other (inter-property) constraints, it can become very difficult to quickly see which constraints are imposed.

9.2.3 if-then-else (JSON Schema)

The previous section showed how the `discriminator` in OpenAPI enables the definition of present-value constraints. In JSON Schema, present-value constraints may be expressed using another construct: `if-then-else`. Listing 9.4 (page 191) shows an example of the `infoWindow` example from the Google Maps API:

It contains two schema definitions: one without the `infoWindow` property and one with `infoWindow`. The first definition also has the `additionalProperties` property to indicate that the list of properties is exhaustive: otherwise, the `infoWindow` property could be added without errors. We elaborate on the property `additionalProperties` in Section 9.2.5.

The `if-then-else` construct is the second part of the definition and the condition verifies the value of `suppressInfoWindow`. In the case that the property `suppressInfoWindow` is set to `true`, the object needs to satisfy the definition `DirectionsRenderer`, otherwise it needs to adhere to the definition `DirectionsRendererExtra`.

Because of `additionalProperties` it is impossible to reuse the definition `DirectionsRenderer` for the `DirectionsRendererExtra` definition. This causes a lot of duplication in the specification, which leads to more verbose specifications and complicates maintainability.

9.2.4 dependencies (JSON Schema)

The fourth language construct that enables defining some of the inter-property constraints is found only in JSON Schema. Using `dependencies`, developers may declare that one property must be present if another property is present. This maps directly onto present-present dependency constraints (defined in Section 2.1.2.1). Furthermore, double implication constraints can also be defined using the `dependencies` language construct twice.

Listing 9.5 shows an encoding in JSON Schema of the present-present dependency constraint specified in Table 2.2. This constraint, found in the Facebook API specification, states that extra information about a link may only be provided

```
{
  "components": {
    "schemas": {
      "DirectionsRenderer": {
        "type": "object",
        "properties": {
          "directions": {
            "type": "string"
          },
          "draggable": {
            "type": "boolean"
          },
          "suppressInfoWindow": {
            "type": "boolean"
          }
        },
        "discriminator": {
          "propertyName": "suppressInfoWindow",
          "mapping": {
            "false": "Extra"
          }
        }
      },
      "Extra": {
        "allOf": [
          {
            "$ref": "#/components/schemas/DirectionsRenderer"
          },
          {
            "type": "object",
            "properties": {
              "infoWindow": {
                "type": "string"
              }
            }
          }
        ]
      }
    }
  }
}
```

Listing 9.3: Use of the `discriminator` to define an inter-property constraint

if the link itself is present. We can use `dependencies` to specify that the extra information properties depend on the base property `link`.

```
{type: 'object', properties: {link:        {type: 'string'},
                              picture:     {type: 'string'},
                              name:        {type: 'string'},
                              caption:     {type: 'string'},
                              description: {type: 'string'}},
                 dependencies: {picture:     ['link'],
                                name:        ['link'],
                                caption:     ['link'],
                                description: ['link']}}
```

Listing 9.5: JSON Schema encoding for dependency constraints

Double implication constraints can be encoded in a similar manner. The snippet below shows an encoding in JSON Schema of the double implication constraint specified in Table 2.3. This constraint was found in the Twitter API. The location of a tweet may be specified using the properties `long` and `lat`, but both properties must be present or absent together. This can be encoded using `dependencies` as a mutual dependency between both properties.

```
{type: 'object', properties: {long: {type: 'number'},
                              lat:  {type: 'number'}},
                 dependencies: {long: ['lat'],
                                lat:  ['long']}}
```

Listing 9.6: JSON Schema encoding for double implication constraints

For humans writing or reading the specification, it can be difficult to see which dependency maps to which logical constraint. Likewise, it is difficult to combine multiple constraints on parameters in JSON Schema. Readability and maintainability could be improved by separating constraints from the structure of the object, eg. by having language constructs for defining custom constraints.

9.2.5 Conclusion

The previous four sections covered the language constructs found in existing API specification languages that support inter-property constraints. Although three of those language constructs are part of JSON Schema, we argue that JSON Schema is not a good candidate for a specification language for web APIs with inter-property constraints. First, the discussed language constructs are not a perfect fit for the categories of inter-property constraints that we have found in Chapter 2. Second, because the language constructs do not map directly onto a kind of inter-property constraints, the error messages do not give a clear indication of which

constraints were not satisfied. Third, the kinds of inter-property constraints that can be expressed using the discussed language constructs are limited and other kinds of logical operators to express inter-property constraints are not supported either. Finally, JSON Schema is a very permissive language: fields that were not described in the schema are allowed by default. This is undesired behaviour while validating web APIs: the list of parameters should be exhaustive and extra parameters should be rejected. To ensure that unmentioned fields are rejected, the field `additionalProperties` must be added and set to `false` in *every* JSON Schema object, or every absent field must be added with a `false` schema.

Because of the minimal support for inter-property constraints in existing API specification languages, we argue that there is a need for a specification language that enables the specification of all kinds inter-property constraints (next to single-property constraints). Moreover, this language needs to be future-proof such that new kinds of inter-property constraints can be easily supported in the language as well. In the next section, we present a specification language that embeds support for inter-property constraints in its core.

9.3 OAS-IP: A Novel Constraint-Centric Specification Language

In this section we introduce OAS-IP, a new specification language for web APIs, focused on defining and imposing constraints on properties of entry points. By defining constraints using propositional logic, writers of API specifications can factor out patterns in constraints and impose constraints on single or multiple properties. This enables the discovery and implementation of novel inter-property constraints.

OAS-IP is an extension of the OpenAPI specification language, which aims to be a vendor-neutral specification language for web services, and is supported by many companies such as Google and Microsoft. OAS-IP offers two extensions to the specification language, both described below. The first extension enables developers to define predicates for common constraints, and the second introduces a new way to impose constraints on query and payload properties. Constraints on path and header parameters are not supported, as they were not found during our survey performed in Chapter 2.

As the goal of TIPC is to verify inter-property constraints at compile-time, TIPC only supports presence constraints. As there is no compile-time restriction for specification languages, the constraint language of OAS-IP is a superset of the constraint language in TIPC.

$$
\begin{aligned}
\mathsf{v} \in \text{Values} &::= \text{Number, String, Boolean or Parameter} \\
\mathsf{f} \in \text{Fields} &::= \mathsf{s} \mid \mathsf{f.s} \mid \mathsf{f.[]} \\
\mathsf{t} \in \text{Types} &::= \mathsf{string} \mid \mathsf{number} \mid \mathsf{boolean} \mid \mathsf{object} \mid \mathsf{t[]} \mid \mathsf{null} \\
\mathsf{c} \in \text{Constraint} &::= \mathsf{o} \mid \mathsf{lc} \mid \mathsf{s}(\mathsf{v}_1, ..., \mathsf{v}_n) \\
\mathsf{o} \in \text{Operations} &::= \mathsf{present(f)} \mid \mathsf{type(f)}{=}\mathsf{t} \mid \mathsf{length(f)} \oplus \mathsf{v} \mid \mathsf{value(f)} \oplus \mathsf{v} \\
\mathsf{lc} \in \text{Logical connectives} &::= \mathsf{and(c,\ c)} \mid \mathsf{or(c,\ c)} \mid \mathsf{not(c)} \mid \mathsf{implic(c,\ c)} \mid \mathsf{iff(c,\ c)} \\
\oplus \in \text{Math operators} &::= =, \neq, <, >, \leq, \geq \\
\mathsf{cd} \in \text{Constraint definitions} &::= \mathsf{s}(\mathsf{s}_1, ..., \mathsf{s}_n) = \mathsf{c}
\end{aligned}
$$

Figure 9.1: Syntax definition for constraints

9.3.1 Constraint Definitions

OAS-IP aims to be a *constraint-centric* specification language, by providing a uniform way to express constraints on fields. The goal of designing a new specification language is to be able to give developers the freedom to define and impose any kind of constraint on fields and subsequently check their code with respect to these constraints. To enable this, OAS-IP provides a constraint language to developers which allow them to define custom constraints.

Figure 9.1 shows the syntax of the constraint language in OAS-IP: a constraint is a logical formula that consists of operations over properties, joined together with logical connectives. This constraint language is a superset of the constraint language proposed in Chapter 4: next to constraints on the presence of fields, OAS-IP also supports constraints on properties of fields. Incorporating more kinds of constraints into the programming language TIPC (formalised in Chapter 6) is discussed in Chapter 10.

Properties target regular (s) and nested fields (f.s), as well as "array" fields (f.[]) — constraints which apply to every element of the targeted array. A constraint is either an operation, a logical combination of operators or a constraint definition. Operations test properties on fields, such as whether a field is present, its type, and restrictions on its value or its length. As usual, precedence can be controlled by parentheses.

Given these constraint building blocks, we can express the different kinds of inter-property constraints identified in Chapter 2. Listing 9.7 shows the definition of these constraints, using logical connectives to combine operations on fields. Expressing the exclusivity constraint requires that either `present(f1)` or `present(f2)` is true. Dependent fields are expressed using an implication. Finally, double implication constraints are expressed with a double implication: `f1`

must be present when `f2` is present, and vice versa.

To promote reusability, constraint definitions enable the abstraction of common constraint patterns. These patterns are defined at the top level of an OAS-IP specification, under the key `x-constraints-definitions`[2]. This key can be defined on the global level of an OpenAPI specification file, and the defined constraints can be used throughout the web API specification.

Listing 9.8 shows the definition of several single-property constraints that are commonly found in web API specification languages.

```
1 x-constraint-definitions:
2   - minimum(f, v)       := value(f) >= v
3   - minimumLength(f,v)  := length(f) >= v
4   - required(f)         := present(f)
5   - string?(f)          := type(f) = string
6   - number?(f)          := type(f) = number
```

Listing 9.8: Sample constraint definitions in the YAML syntax

9.3.2 Constraints

The second extension made to OpenAPI specification is the addition of the property `x-constraints`. This key is used to impose constraints on parameters using the constraint syntax or custom definitions. This extension offers an additional way to define constraints; OAS-IP still takes the regular OpenAPI specification constraints into account as well.

Listing 9.9 (page 192) shows the machine-readable specification for the Private-Message example of the Twitter API. In OpenAPI, and thus in OAS-IP, entry points are grouped under the `paths` key, with the different HTTP methods they support nested under the entry point. Lines 5–7 list the properties for the POST method of this entry point, including single-property constraints to impose a type on the parameters. The constraints on the parameter for this entry point are listed in the `x-constraints` section, indicating that `text` is a required field and that exactly one of screen name and user ID must be supplied.

9.3.3 Comparison with Other Web API Specification Languages

So far, this section introduced OAS-IP, a specification language for web APIs with constructs for defining constraints by means of predicates over fields. While existing specification languages have a predefined limited set of constraints, constraints in OAS-IP can be defined using any logical combination of the provided

[2]Custom fields in the OpenAPI specification are prefixed with `x-`.

operations. Moreover, complicated predicate combinations are abstracted to a custom constraint definition. This is an advantage over existing specification languages, where lack of abstractions gives rise to readability and maintainability issues. For example: defining a present-value dependency constraint in OpenAPI or JSON Schema requires to make a custom kind of schema that inherits from the original schema.

To gauge the expressiveness of OAS-IP for modelling existing web APIs, we examined the constraint keywords of OpenAPI and JSON Schema and attempted to replicate them with constraint definitions in OAS-IP. Apart from the type and whether the field is required, the majority of keywords define constraints on either numeric values of fields or the size of arrays or objects, and thus supported with the existing operations in OAS-IP: using `length` and `value`. Most of the other constraints can be supported by adding new operations such as regex expressions (for `pattern`), mathematical operations (for `multipleOf`) and a new language construct for verifying the uniqueness of items in array (for `uniqueItems`). OAS-IP currently does not support the JSON Schema `items` and `additionalItems` constraints which provide a different schema for each item of an array.

A final difference is the handling of unspecified fields. The `patternProperties` keyword allows constraining properties whose name matches a regular expression, while the `additionalProperties` keyword either validates or forbids unknown properties. As we mentioned before, OAS-IP defaults to rejecting requests with unknown fields. This is a conscious choice to give optimal guarantees about correct requests. If desired, both can be supported with the addition of *patterned fields*: for example, the pattern `string?(metadata./^x-/)` would require that unspecified fields of the `metadata` object starting with `x-` must be strings.

9.4 Inter-property Constraints in Specification Language Tools

The main functionality that comes along with machine-readable specification languages is the generation of human-readable documentation. However, the existence of a machine-readable specification also enables the generation of a variety of other tools. These tools can provide a wide range of functionality:

- creating and editing specifications in a disciplined way;
- testing of APIs by generating mock-ups from the specification;
- facilitate server-side development by generating server stubs: a skeleton of code can be generated from the given machine-readable specification, gen-

erating all the necessary code for entry points, and verifying that incoming data effectively satisfies the constraints imposed by the specification;

- facilitate client-side development by generating client SDKs: libraries that provide a regular function for each entry point and take care of sending the HTTP request details behind the scenes.

The introduction of inter-property constraints affects every aspect of the tooling suite. First, when creating and editing API specifications, API developers could get the opportunity to create and impose a wide variety of constraints on fields, including inter-property constraints. Second, inter-property constraints could to be taken into account when testing APIs. Third, the generators of server-side stubs and client-side SDKs could support inter-property constraints as well. This would result in a more complete coverage of all the constraints that are imposed by the APIs, in contrast to the subset of constraints that could be defined by existing specification languages.

As an example of how tools for constraint-centric specification languages can incorporate support for inter-property constraints, we have developed two prototype tools for OAS-IP: a runtime verification tool and a compile-time code generation tool. In the remainder of this section, we will elaborate on these tools.

9.4.1 `VerifyRequest` library

In this section we introduce **`VerifyRequest`**, a library that accompanies the web API specification language OAS-IP introduced in Section 9.3. This library verifies whether the constraints imposed on the data of a request are satisfied at runtime, and can be deployed on both sides of the client–server divide. This library is written in regular TypeScript and is independent of TIPC: **`VerifyRequest`** does not verify inter-property constraints at compile-time.

Recall that we have discussed the various ways that existing web services handle and report errors in inter-property constraints in Chapter 2. A first kind of behaviour is to silently ignore the violation, meaning that results can vary as the server code is changed. A second behaviour is to produce textual error messages, which may be hard to decipher. In both cases it can be very hard to determine why a given request does not yield the desired result.

To aid developers on both sides of the client–server divide, we have developed **`VerifyRequest`**, a library which dynamically verifies that requests made in web applications are correctly formed. If all constraints are satisfied, the request is executed and a result is returned. Otherwise, **`VerifyRequest`** returns an error message indicating which constraints were not satisfied. This improves on both

kinds of error behaviours by rejecting invalid requests and producing an informative error message.

Client side On the client side, we have developed a wrapper for the popular Node.js `request` library for making API requests. Whenever the program attempts to make a request, `VerifyRequest` looks up the constraints for the data of that entry point in its specifications using the URL and HTTP method, and verifies whether they are satisfied.

The following code snippets shows an example of a small program that uses `VerifyRequest`. First, developers need to import the `VerifyRequest` library. This library replaces the import of `request`. `VerifyRequest` only adds one function to the interface of the original library: `addDefinition`. This function allows the registration of API specifications, such that `VerifyRequest` can later on verify whether requests are compliant with their specifications. Lines 5-10 in Listing 9.10 shows the sending of a POST request, which is identical to a POST request with the `request` library. However, the `VerifyRequest` library first verifies whether constraints are satisfied, before executing the request. In the case of Listing 9.10, `VerifyRequest` will show a warning indicating that the exclusivity constraint on the user properties is not satisfied.

Server side `VerifyRequest` can also be deployed as middleware on the server side; there it primarily serves as a way of validating requests before they are acted upon. However, `VerifyRequest` also offers the benefit of providing *uniform behavior* in the face of erroneous requests. Violations of inter-property constraints are thus detected and reported according to the (externally accessible) specification, instead of leaving this to web service developers.

The implementation of `VerifyRequest` can be found on GitHub[3]. Both client and server side can easily be adapted to other AJAX-like libraries (jQuery) and HTTP servers (Koa) respectively. While our prototype happens to be written in JavaScript, the principle can be applied to other languages, both on the client and server side.

9.4.2 Client SDK Code Generator

The second tool for OAS-IP is `SDKCodeGen`: a code generation tool that forms the basis for a client SDK (Software Development Kit). The result of `SDKCodeGen` is a module that contains a function for each entry point in the specification, written in the TIPC programming language. Constraints written in OAS-IP are

[3] `https://github.com/noostvog/Verify-Request`

translated to TIPC: this way, constraints are automatically verified by the type system. The generated module can also serve as a starting point for a client SDK, in which authentication and request functionality can be inserted.

The main functionality of `SDKCodeGen` is **the translation of the constraints listed in an OAS-IP specification to a TIPC interface**. `SDKCodeGen` only takes constraints into account that can be expressed in TIPC. In other words, constraints on the type of fields and the presence of fields are translated to TIPC, but constraints that depend on the value or length of a field are not taken into account.

As an example, Listing 9.12 on page 190 shows the code that is generated by `SDKCodeGen` for a subset of the specification of the Twitter API. As input, `SDKCodeGen` receives the OAS-IP specification in Appendix C. A snippet of this specification can be found in Listing 9.11 on page 189: this snippet describes the entry point for creating a private message in Twitter. It lists the three parameters for the entry point: `text`, `user_id` and `screen_name`. The `user_id` must be a number, while the other two need to be strings. `text` is indicated as a required field. Using the extensions present in OAS-IP, we are able to express the inter-property constraint as well: `x-constraints` contains the exclusivity constraint between the user ID and the screen name, using a custom constraint (whose definition can be found in the full API specification).

The complete API specification contains two additional entry points:

- updating a status: this entry point contains a lot of fields, most of which are optional, except for the `status` field which is required and `long` and `lat`, on which a double implication constraint is imposed.

- add a member to a list which receives 6 fields: `list_id` and `slug` on which an exclusivity constraint is imposed, as well as between `screen_name` and `user_id` and between `owner_screen_name` and `owner_id`. Additionally, there is a dependency constraint between the two owner fields and `slug`.

Given the input in Appendix C, `SDKCodeGen` will generate the output listed in Listing 9.12 on page 190. This output is a TIPC module that contains a function for all the HTTP methods defined for each entry point in the OAS-IP specification. All constraints on the presence of properties and between properties are translated to a TIPC interface. Next, this interface is imposed on the argument of the accompanying function. This ensures that the TIPC developer of the client side constructs the object correctly.

The implementation of the `SDKCodeGen` tool can be found at GitHub[4].

[4] `https://github.com/noostvog/SDKMockupGenerator`

9.5 Conclusion

We started this chapter by introducing machine-readable API specification languages and explained their advantages in web development. After having given an overview of the most popular machine-readable specification languages, we showed that current machine-readable API specification languages fall short regarding expressing inter-property constraints. There are two specification languages that are able to express some of the different kinds of inter-property constraints we have introduced in Chapter 2: the OpenAPI specification and JSON Schema. Each have three language constructs (two of which are shared among both specification languages) that are able to express one (sub)kind of an inter-property constraint. However, the language constructs are not a direct mapping onto an inter-property constraint which causes the specifications to be very verbose. This leads to unmanageable specifications and unclear error messages.

As a solution to those problems, we introduced a new specification language called OAS-IP. The goal of OAS-IP is to be constraint-centric such that many kinds of constraints can be expressed with its constraint language, including constraints over multiple fields. The constraint language of OAS-IP is more expressive than TIPC's constraint language, because there is no limitation for requirements to be verifiable at compile time. Moreover, we showed that the constraint language of OAS-IP is as expressive as existing specification languages regarding single-property constraints: almost all of the constraints that exist in JSON Schema and OpenAPI specification can be expressed with the constraint language of OAS-IP, and others can be added with minimal additions to the language.

Finally, this chapter showed how OAS-IP can impact the development of web applications:

- As a first example, we have developed a prototype implementation called `VerifyRequest`, which **verifies whether the data of a request satisfies the constraints imposed on the data** in the specification. On the client-side, this is implemented as a wrapper around an existing `request` library. This enables developers to automatically check whether or not requests satisfy the given constraints with minimal changes to the code. On the server-side, this library can be used to verify the data before processing the request.

- The second example is a tool that generates a server stub written in TIPC, given a specification written in OAS-IP. This tool **translates the constraints imposed by the specification to interfaces in TIPC** and generates a function for each entry point. `SDKCodeGen` is a great example

of how the incorporation of inter-property constraints in both specification languages and programming languages can enable the improvement of the web application development cycle. On the client-side, this tool can serve as a simulation of the server-side code. The interfaces guarantee that the data will satisfy the constraints that will be imposed by the actual server implementation. On the server-side, this tool can serve as a starting point for the implementation of the server. The interfaces with inter-property constraints ensure the correct usage of incoming data.

```
"/direct_messages/new": {
  "post": {
    "description": "sends a new direct message to specified user",
    "security": [
      {
        "oauth": [
          "basic"
        ]
      }
    ],
    "parameters": [
      {
        "name": "user_id",
        "in": "query",
        "description": "description",
        "type": "number",
        "required": false
      },
      {
        "name": "screen_name",
        "in": "query",
        "description": "screen name of user receiving message",
        "type": "string",
        "required": false
      },
      {
        "name": "text",
        "in": "query",
        "description": "text of your direct message",
        "type": "string",
        "required": true
      }
    ],
    "x-constraints": [
      "xor(user_id, screen_name)"
    ],
    "responses": {
      "200": {
        "description": "OK",
        "schema": {
          "$ref": "#/definitions/Messages"
        }
      }
    }
  }
}
```

Listing 9.11: Snippet of the Twitter specification in OAS-IP

```
export module Twitter{
    export interface PostDirectmessagesnew {
        user_id?: number;
        screen_name?: string;
        text?: string;
    } constrains {
        present(text);
        or(and(present(user_id), not(present(screen_name))),
           and(not(present(user_id)),present(screen_name)));
    }
    export function postDirectmessagesnew(body: PostDirectmessagesnew){
        // your implementation here
    }
    export interface PostStatusesupdate {
        status?: string;
        in_reply_to_status_id?: number;
        possibly_sensitive?: string;
        lat?: string;
        long?: string;
        place_id?: number;
        display_coordinates?: string;
        trim_user?: string;
        media_ids?: string;
    } constrains {
        present(status);
        and(implic(present(lat), present(long)),
            implic(present(long),present(lat)));
    }
    export function postStatusesupdate(body: PostStatusesupdate){
        // your implementation here
    }
    export interface PostListmemberscreate {
        list_id?: number;
        slug?: string;
        screen_name?: string;
        user_id?: number;
        owner_screen_name?: string;
        owner_id?: number;
    } constrains {
        or(and(present(slug), not(present(list_id))),
           and(not(present(slug)), present(list_id)));
        or(and(present(user_id), not(present(screen_name))),
           and(not(present(user_id)), present(screen_name)));
        implic(present(slug),
               or(and(present(owner_screen_name), not(present(owner_id))),
                  and(not(present(owner_screen_name)), present(owner_id))));
        implic(present(owner_screen_name),present(slug));
        implic(present(owner_id),present(slug));
    }
    export function postListmemberscreate(body: PostListmemberscreate){
        // your implementation here
    }
}
```

Listing 9.12: Output of **SDKCodeGen** for the Twitter specification

```
{
  "definitions": {
    "DirectionsRenderer": {
        "type": "object",
        "properties": {
          "directions": { "type": "string" },
          "draggable" : { "type": "boolean" },
          "suppressInfoWindow": { "type": "boolean" }
        },
       "additionalProperties": false
    },
   "DirectionsRendererExtra": {
        "type": "object",
        "properties": {
          "directions": { "type": "string" },
          "draggable" : { "type": "boolean" },
          "suppressInfoWindow": { "type": "boolean" },
          "infoWindow":  { "type": "string" }
        }
   }
  },
  "if":{
    "properties":{
      "suppressInfoWindow":{"const": true}
    }
  },
  "then":{
    "$ref": "#/definitions/DirectionsRenderer"
  },
  "else":{
    "$ref": "#/definitions/DirectionsRendererExtra"
  }
}
```

Listing 9.4: Example of the if-then-else construct in JSON Schema

```
x-constraint-definitions:
  - xor(f1, f2)              := or(and(present(f1), not(present(f2))),
                                   and(present(f2), not(present(f1))))
  - pp-dependent(f1, f2)     := implic(present(f1), present(f2))
  - pv-dependent(f1, f2, v) := implic(present(f1), value(f2) = v)
  - vv-dependent(f1, f2, v1, v2)
                             := implic(value(f1) = v1, value(f2) = v2)
  - doubleimplic(f1, f2)     := iff(present(f1), present(f2))
  - nand(f1, f2)             := not(and(f1, f2))
```

Listing 9.7: Sample constraint definitions in the YAML syntax

```
paths:
 /direct_messages/show:
   post:
      parameters:
        - { name: screen_name, in: query, type: string }
        - { name: user_id, in: query, type: string }
        - { name: text, in: query, type: string}
      x-constraints:
        - present(text)
        - xor(screen_name, user_id)
```

Listing 9.9: Expressing constraints for an API operation in OAS-IP

```
let request = require('./APIVerifyTool/verify-and-request.js');

request.addDefinition(twitterDefinition);

request.post({url: "api.twitter.com/1.1/direct_messages/new.json",
              /* oauth information */
              form: {user_id: 42,
                     screen_name: "Alice"
                     text: "Hello"}},
            function(e,r,user){ console.log(user)});
```

Listing 9.10: Example use of **`VerifyRequest`** library

Chapter 10

Conclusion

This dissertation introduced the concept of *inter-property constraints*, i.e. constraints between properties. We identified their presence in real-world web APIs and incorporated support for inter-property constraints into several aspects of the development cycle for clients and servers of web APIs. In this final chapter, we summarise this dissertation. We provide an overview of the contributions and discuss avenues for future research.

10.1 Summary

We started this dissertation with the identification of inter-property constraints. Chapter 2 performed a study of the documentation of libraries and categorised examples based on how the constraints between the properties are combined. This resulted in four categories that are commonly found in real-world APIs, based on the logical operator used to combine the constraints: exclusivity constraints, dependency constraints, double implication constraints and NAND constraints.

Next, we explored how integrating inter-property constraints into a mainstream statically typed programming language affects the creation of objects and the accessing and updating of object properties (Chapter 3). This resulted in seven requirements that form a blueprint for incorporating inter-property constraints into any statically typed programming language. In Chapter 4, we tackled these requirements in a variant of TypeScript called TIPC. The changes to the syntax were minimal: the interface definitions had to be extended to support inter-property constraints and the `assign` construct was added to enable the safe update of properties part of an inter-property constraint. Satisfying the requirements entailed including concepts from propositional logic into the type system. For every requirement, we discussed the impact of this change on the expressivity

of the programming language.

After a brief detour to explain the idiosyncrasies of TypeScript's type system (Chapter 5), we incorporated the ideas of Chapter 4 into an existing formalisation of TypeScript. More specifically, we presented the syntax, typing rules, and operational semantics of TIPC (Chapter 6). In the type system, typing rules and assignment compatibility rules had to be changed to guarantee type safe property accesses, assignments, type casts and `assign` calls. The operational semantics was extended with support for `assign` and tags to indicate interface instances. These tags enabled proving the soundness of TIPC where, in the light of inter-property constraints, we proved that interface instances will —at runtime— always contain a combination of properties that satisfies the interface constraints.

We incorporated the extensions proposed in TIPC in the official implementation of TypeScript, resulting in $\text{TypeScript}_{\text{IPC}}$ (Chapter 7). In order to remain backward-compatible with existing TypeScript applications, $\text{TypeScript}_{\text{IPC}}$ contains both regular interfaces and interfaces with inter-property constraints.

In Chapter 9, we returned to documentation for web APIs (the inspiration for inter-property constraints) and we found that current API specification languages lack support for inter-property constraints. We introduced OAS-IP, an extension of OpenAPI with support for inter-property constraints. We introduced two development tools that accompany OAS-IP: `VerifyRequest` and `SDKCodeGen`. The first tool generates verifies request data at runtime, given an OAS-IP specification file. The second tool translates constraints in an OAS-IP file to TIPC interface definitions, which enables the compile-time verification of inter-property constraints.

10.2 Restating the Contributions

Identification and Classification of Inter-property Constraints Our first contribution is the study of inter-property constraints in real-world API documentation. We showed that inter-property constraints are common in six popular web APIs: Google Maps, Twitter, YouTube, Flickr, Facebook and Amazon. We categorised the instances of inter-property constraints: exclusivity constraints, dependency constraints, double implication constraints and NAND constraints.

Statically typed programming language with support for inter-property constraints We identified seven requirements that object-oriented programming languages need to take into account when incorporating inter-property constraints, and proposed a solution for every requirement. We aimed for a type system that can be integrated in any object-oriented programming language for

web development. Therefore, we incorporated these ideas in TypeScript, which resulted in the programming languages TIPC (formalisation) and TypeScript$_{\text{IPC}}$ (implementation). Moreover, the changes made to TypeScript's type system do not increase the annotation overhead and have only a small impact on the expressivity of the language:

- **Annotation Overhead:** In TIPC, interface definitions can be extended with syntax to express inter-property constraints. These annotations are not *overhead*, but enable programmers to better formulate their intentions.

- **Required Code Changes:** To guarantee soundness, we had to restrict property accesses in TIPC: only properties that are certainly present or absent can be accessed. Other properties require an explicit presence test before they can be accessed. For property accesses, there are four cases to consider when converting existing TypeScript programs to TIPC:

  - The TypeScript code already performed `if` tests before accessing a property. In this case, TIPC uses the information from the `if` tests to allow the property access. Thus, no code changes are required.

  - The TypeScript code did not perform a test before accessing the property. In this case, this might be a bug in the code: although the requirement could not be listed explicitly in the TypeScript code, the requirement still needs to be checked before the property in question can be used. TIPC flags this bug: the code changes (i.e. `if` statements) required by TIPC actually ensure that the program is bug free.

  - In some cases, the type system of TIPC is not powerful enough and requires more tests than strictly necessary. For example, information from `if` statements is disregarded when the object is passed to a function which expects the original interface type. These code changes qualify as an undesired burden for the developer, but increasing the power of the type system to prevent this —for example, by using alias types or by allowing the developer to explicitly specialise interface types— would create a significant annotation overhead.

  - Updating properties that are part of an inter-property constraint must be done simultaneously to preserve type safety. Transforming subsequent single-property assignments to a simultaneous assignment is a required code change in order for TIPC to accept the code. We argue that this code change is minimal and can often be automated.

Machine-readable Specification Language We showed that existing machine-readable specification languages for web APIs have limited support for inter-property constraints. We have extended the OpenAPI specification language with support for every kind of inter-property constraint, resulting in OAS-IP. This specification language has an extensive constraint language that allows the definition of reusable inter-property constraints on the presence, type, and value of properties. Moreover, we showed the expressivity of OAS-IP is on par with existing specification languages regarding single-property constraints.

We presented two tools that accompany the OAS-IP specification language: `VerifyRequest` and `SDKCodeGen`. The `VerifyRequest` tool tests that data satisfies the constraints imposed in the specification language, including the custom (inter-property) constraints. `VerifyRequest` shows that including inter-property constraints in specification languages makes the accompanying tools more effective. The `SDKCodeGen` tool translates the data and their constraints in an OAS-IP specification file to interface definitions, which enables the compile-time verification of constraints by TIPC. Together, `SDKCodeGen` and TIPC improve the capabilities of specification language tools and web development in general.

10.3 Future Work

In this section, we discuss possible avenues for future research.

10.3.1 Value-dependency Constraints

In Chapter 2, we identified four kinds of inter-property constraints: exclusivity constraints, dependency constraints, double implication constraints and NAND constraints. While almost all these constraints between properties impose restrictions on the presence of properties, we also found examples of dependency constraints that impose restrictions on *values* of properties. In a PV-dependency constraint, the allowed values for a property depend on the presence of another property, or the other way around where the presence of a property depends on the value for another property. In a VV-dependency constraint, the allowed values for a property depend on the value of another property.

Inter-property constraints in TIPC are limited to constraints on the presence of a property. The API specification language of Chapter 9 already supports a superset of the TIPC constraint language, in which constraints on values can be expressed. In the future, we plan on adding support for value-dependency constraints to TIPC.

Syntactically, we plan on adding the `value` keyword to inspect values of properties. Listing 10.1 shows an example. This interface is a translation of the ModerationStatus entry point of the YouTube API. It contains three properties: `id`, `moderationStatus` and `banAuthor`. As line 8 shows, the `banAuthor` property may only be present if the `moderationStatus` is set to `rejected`.

```
1 interface ModerationStatus {
2     id: string;
3     moderationStatus: string;
4     banAuthor: boolean;
5 } constraining {
6     present(id);
7     present(moderationStatus);
8     present(banAuthor) → (value(moderationStatus) == "rejected");
9 }
```

Listing 10.1: TIPC interface with value-dependency constraints

To incorporate value-dependency constraints in the type system of TIPC, we will look at *singleton types* as a starting point. Singleton types allow the developer to specify exactly what value the expression must have. TypeScript already supports singleton types in the form of *number literal types* and *string literal types*. Listing 10.2 shows an example of literal types in TypeScript. Only the exact literal can be assigned to a literal type (lines 1–2, 4–5), but literal types remain assignable to their general type (lines 3 and 6).

```
1 let status1: "rejection" = "rejection";      //OK
2 let status2: "rejection" = "heldForReview"; //ERROR
3 let status3: string      = status1;         //OK
4 let x: 42     = 42; //OK
5 let y: 42     = 24; //ERROR
6 let z: number = x;  //OK
```

Listing 10.2: Singleton types in TypeScript

Singleton types enable the type system to know the value of an expression at compile time. This way, the type system is able to verify constraints that impose restrictions on values, at initialisation time. The rules for accessing a property remain unchanged, as a value-constraint does not affect the presence or absence of a property.

The type system of TIPC uses logical entailment to verify the assignment compatibility of expressions. In a language with value-dependency constraints, constraints are expressed in a form of predicate logic instead of in propositional logic. While the current type system uses logical entailment for propositional logic, the logical entailment of the new type system will have to take values of properties

into account as well. As a consequence, the boolean satisfiability solvers (used to check whether a set of constraints is satisfiable) and propositional sequent calculus provers (to verify a logical entailment) will have to be replaced by their counterpart in predicate logic (SMT solvers and predicate sequent calculus provers).

Updating a property create new challenges in a type system with value-dependency constraints. In TIPC as it was presented in Chapter 6, a property may be updated (with any value with the same type as in the interface definition) if the property is certainly present. However, in the light of value-dependency constraints, this does not suffice to guarantee type safety: updating a property might invalidate a value-dependency constraint.

Listing 10.3 shows an example. On line 1–3, the object literal satisfies all constraints of the `ModerationStatus` interface. On line 4, the `moderationStatus` property is updated with a new value. In a type system without value-dependency constraints, this property would be considered type safe, as `moderationStatus` is guaranteed to be present. In a type system with value-dependency constraints, this update is type unsafe: the update on line 4 will invalidate the third inter-property constraint in the `ModerationStatus` interface and thus needs to be rejected by the type system.

```
1 let status: ModerationStatus = {id: 12,
2                                 moderationStatus: "rejection",
3                                 banAuthor: true};
4 status.moderationStatus = "heldForReview";
```

Listing 10.3: Unsafe property update

To conclude, a type system with value-dependency constraints needs to impose even more restrictions regarding single-property updates: if the property is part of a value-dependency constraint, it may not be updated. Future research has to show whether there are ways to weaken this restriction.

10.3.2 Imperative Multi-update

TIPC is an imperative object-oriented programming language. Properties of objects on which inter-property constraints are imposed can be updated, as long as its presence or absence is certain. In Chapter 3, we have shown the need for a language construct that updates multiple properties simultaneously. To meet these needs, we introduced the language construct `assign` in TIPC. To keep TIPC sound when combining `assign` with `if` statements, we chose to make `assign` functional (by returning a new object instead of modifying its first argument).

Listing 10.4 demonstrates this problem. The code snippet starts with two variable declarations: the first one creates an instance of the interface `PrivateMessage`,

while the second variable declaration points to the first one. Next, the presence of `user_id` is verified in an `if` statement. As we have seen in Chapters 4 and 6, the type system will add an extra constraint (indicating the presence of `user_id`) to the `pm` object. The true branch of the conditional first simultaneously updates the second variable (`pm2`), switching from `user_id` to `screen_name`. Next, the `user_id` property of `pm` is updated with a new ID. Because of the extra constraint in `pm`, the type system will accept this assignment. However, because `pm` and `pm2` point to the same object, this will lead to a `PrivateMessage` object that contains both a user ID and a screen name!

```
1 let pm: PrivateMessage  = {text: "Hello!", user_id: 42};
2 let pm2: PrivateMessage = pm;
3
4 if (pm.user_id !== undefined) {
5   assign(pm2, {user_id: undefined, screen_name: "Alice"});
6   pm.user_id = 43;
7 }
```

Listing 10.4: Aliasing problems with an imperative version of a simultaneous update

The root of the problem is situations where there are multiple references to the same object: it is only safe to update an object if there is only one reference to it. To safely lift the functional restriction on `assign`, there must be a guarantee that there is only one reference to the object.

In Chapter 8, we have already discussed the work of Chugh et al. [2012a], in which they present a refinement type system for JavaScript with imperative updates. DJS achieves type safety by using flow-sensitive heap types, a form of *Alias Types* which guarantees that there are is only one cell in the heap that refers to a particular object. Chapter 8 contains an example of the type annotations in DJS and shows that there is a heavy annotation load because of the heap types.

In TIPC, we strive to keep the type annotation burden as low as possible: programmers only have to change the interface definition. Moreover, the complexity of type annotations in TIPC is very low, as it only requires basic knowledge of logical connectives. A middle ground between the immutability restriction in TIPC and the heavy annotation load on DJS might be the work on *uniqueness types* [Boyland et al., 2001; Gordon et al., 2012; Militão et al., 2014]. An expression of a *unique type* is guaranteed to have only one reference to it. Existing work on uniqueness types could form the starting point for integrating mutable simultaneous updates into TIPC.

10.3.3 Gradual Typing For Inter-property Constraints

A key feature of TypeScript, the base language for TIPC, is the optionality of its type system (as already explained in Section 5.1). When developers do not assign a type to a variable, this variable receives the top type `any`. However, the transition from typed code to untyped code in TypeScript is unchecked. In other words, the types in TypeScript have no effect on its compiled output (a JavaScript program): all type annotations are stripped from the TypeScript program while compiling to JavaScript.

This is in contrast to a programming language with a gradual type system, which inserts run-time checks between typed code and untyped code in the program. These checks ensure that transitions between typed and untyped code are type safe: only untyped code can lead to program crashes. There already exists work towards gradual typing for TypeScript, most notably the work by Richards et al. [2015] and Rastogi et al. [2015]. As already explained in Chapter 8, they ensure type safety between typed and untyped parts of TypeScript code by inserting run-time checks.

In this section, we do not want to elaborate on gradual types on the level of types. Instead, we want to investigate how we can introduce finer-grained gradual types for inter-property constraints into TIPC. Listings 10.5 to 10.7 show an example of how inter-property constraints can benefit from fine-grained gradual types. The code snippet in Listing 10.5 shows a function with as parameter a `PrivateMessage` object. Inside this function, there is a (type unsafe) update of the `user_id` property.

```
1 function getInfo(pm: PrivateMessage) {
2   pm.user_id = 43;
3 }
```

Listing 10.5: TIPC program

In TypeScript, this program is compiled to JavaScript, despite the unsafe property access. As a consequence, this property update might still result in a `PrivateMessage` object that contains both properties at runtime.

```
1 function getInfo(pm) {
2   pm.user_id = 43;
3 }
```

Listing 10.6: Compiled TIPC program with to JavaScript

On the other hand, a programming language with fine-grained gradual types would be able to insert the necessary checks to prevent this property access at runtime. Listing 10.7 shows what the compiled version of Listing 10.5 could look

like with a fine-grained gradual type system. On line 2, there is a runtime check (`assert`) that verifies the presence of `user_id`. If `user_id` is not present, the function is aborted before the invalid property update is executed.

```
1 function getInfo(pm) {
2   assert (pm.user_id !== undefined);
3   pm.user_id = 43;
4 }
```

Listing 10.7: Gradual type system for TIPC

Gradual types on the level of type refinement is already explored in work by Lehmann and Tanter [2017]. In the future, we would like to explore how a version of their gradual refinement types can be incorporated in TIPC to lift some of the restrictions (on for example the property accesses) by delegating the generation of `if` statements to the gradual type system.

10.3.4 Portability to Other Programming Languages

In this dissertation, we incorporated inter-property constraints into TIPC, an imperative object-oriented programming language with structural types. In this section, we elaborate on how inter-property constraints can be incorporated in a type system with a nominal types and classes, such as TypeScript, Java, Ruby and Smalltalk.

10.3.4.1 Nominal Type System

As we have already explained in Section 5.4, TIPC has a structural type system. In a structural type system, different types are compared based on their structure. The structural type system has a significant impact on how objects with inter-property constraints are compared in TIPC. The type system has to verify whether the presence constraints of one object also satisfy the constraints of the other object. This is in contrast to a nominal type system, where types are compared based on their name.

Recall two interfaces we have previously introduced earlier: `PrivateMessage` (defined in Listing 3.2) and `PrivateMessageId` (defined in Listing 3.9). Both have a required `text` property, but differ in their user properties. `PrivateMessage` has an exclusivity constraint between two user properties `user_id` and `screen_name`, while users in `PrivateMessageId` can only be identified by the (required) property `user_id`. Listing 10.8 shows a function with one parameter of interface type `PrivateMessageId`. In the body of the function, this parameter is assigned to a variable of type `PrivateMessage`. A structural type system such as the one of

TIPC accepts this code snippet, as every valid `PrivateMessageId` object will also be a valid `PrivateMessage` object. A nominal type system, on the other hand, will reject this code snippet, as `PrivateMessage` and `PrivateMessageId` are two different interfaces.

```
1 function foo(pm: PrivateMessageId) {
2   let pmi: PrivateMessage = pm;
3   //...
4 }
```

Listing 10.8: Assignment in structural and nominal type systems

10.3.4.2 Classes

In TypeScript (and in other object-oriented programming languages), developers have the choice to define the type of an object with an interface type or a class type. In TIPC, inter-property constraints are incorporated into interface types instead of class types. This choice was intentional: TypeScript interfaces form a lightweight way to assign types to object literals. This is in contrast to classes, which have to be constructed using the class constructors. In this section, we elaborate on how inter-property constraints can be incorporated into classes.

Listing 10.9 on page 208 shows a proposal for classes with support for inter-property constraints. The class definition is written in TypeScript, but easily translatable to other programming languages with classes. We have chosen to incorporate this extension into classes using annotations on properties and method definitions. In this snippet, we have translated the `PrivateMessage` interface to a class.

Properties When declaring the properties of a class, developers need to provide the constraints on the presence or absence of those properties. Our proposal uses annotations to register single-property constraints and inter-property constraints. These annotations replace all presence constraints that could be imposed on the properties in the original programming language.

The `PrivateMessage` class has three properties (`text`, `screen_name` and `user_id`) and two inter-property constraints on those properties: `text` is a required property and there is an exclusivity constraint between the properties `user_id` and `screen_name`.

To guarantee that the presence constraints are not bypassed, it is crucial that the properties are all declared as private properties. This moves the responsibility of keeping the constraints satisfied to the constructors and the methods that modify properties.

Constructors One way of defining constructors for classes with inter-property constraints is to define a constructor for every valid combination of properties. In the case of `PrivateMessage`, this would result into two constructors: one with `text` and `user_id` and one with `text` and `screen_name`. However, in some cases this approach is not feasible. For example, if both user properties would be strings, this would result in two constructors that expect two strings which cannot be expressed using constructor overloading.

We propose to list all properties of the class as arguments of the constructor. On top of that, the constructor is annotated with presence constraints indicating the valid combinations of the arguments (where absent arguments have the `undefined` (or similar) value). Given the annotations, we are certain that all presence constraints are satisfied when all properties of the class are updated with the corresponding argument.

A downside of this approach is the repetition of the presence constraints (as they are already specified for the properties of the class), which can be solved by having *list-all-constraints* annotation. Note that, as a consequence of defining presence constraints in the annotations, all arguments of the constructor are optional by default. As a consequence, a constructor receives many arguments of which a part will be set to `undefined`. In the case of many properties of which only a few have to be present, this leads to cluttered constructor calls.

Lines 8 and 9 show the annotations for the constructor definition on lines 10 to 14. The annotations impose constraints on the parameters of the constructor: `text` must be provided as an argument for the constructor, as well as exactly one of the user arguments. The other user argument must have an `undefined` value. Given the guarantees from the annotations on the constructor arguments, it is safe to update all properties of the class with their corresponding constructor argument.

Note that it does not suffice to only list the presence constraints on the arguments of a constructor. More specifically, there are two issues that have not been addressed yet. First, as not all properties are instantiated simultaneously, the object is in an invalid state up until the end of the constructor. Second, there is no guarantee that the object being built in the constructor satisfies all presence constraints. Even when the constraints on the constructor match the constraints at the beginning of the class, it is still possible that not all arguments are (correctly) assigned to the properties of the class. This can also lead to an invalid states.

To solve this, we plan on drawing inspiration from work that solves similar challenges, such as *delayed types* by Fahndrich and Xia [2007] and *recency types* by Heidegger and Thiemann [2010].

Setters For every method that modifies properties of the class, it is of key importance to guarantee that the (inter-property) constraints defined on lines 2 and 3 remain satisfied. The key insight is that setters must be defined per cluster, instead of per property. We have already seen the concept of clusters when we introduced TIPC's `assign` (Section 3.5), where each cluster corresponds to a transitive closure of properties and constraints.

For the `PrivateMessage` class, there are two clusters: `text` is a singleton cluster and `user_id` and `screen_name` also form a cluster. This setters of the class `PrivateMessage` correspond to these two clusters: `text` can be updated using `setText`, and the `user_id` or `screen_name` can be updated using `setUser`. Just as with constructors, method arguments are optional by default: `setText` is annotated with a constraint indicating that `text` is required. The method `setUser` is responsible to safely update both user properties. To reflect this, `setUser` lists two arguments (an ID and a name) together with an annotation to indicate that exactly one of the arguments should receive a meaningful value. Given this constraint, it is safe to update both properties in the body of `setText`: one of them is guaranteed to be `undefined`.

Note that methods that modify data will have the same challenges regarding temporarily invalidating constraints.

Getters As we have said in the beginning of this section, the properties of the class have to private. As a consequence, these properties cannot be updated without setters. Therefore, it is type safe to allow an accessor for each property: a getter does not give a developer the power the change that property. For properties that are certainly present (such as `text`), the return type can be the defined type for that property. Properties for which neither presence nor absence can be proven, the return type can be a union type of the defined property type and `undefined`. Contrary to TIPC, this is safe in the context of classes because of the *private* restriction on properties.

Note that there is no need for constraint annotations as getters do not have arguments.

Listing 10.9 showed an example of classes with inter-property constraints in TypeScript. Most of the characteristics of TypeScript classes can be translated to other class-based programming languages in a straightforward way. One of the aspects that requires caution is the notion of optionality in the different programming languages. Having the guarantee that a property of a certain type does not contain an `undefined` or `null` value, is essential to preserve type safety. For example, types in C# are not nullable by default and may only contain `null` when the type

is explicitly labelled as `nullable`. In Java, on the other hand, it is allowed to assign `null` to any object. To restrict this, Java could be extended with`@NonNull` annotations or restrictive types (such as work by Fähndrich and Leino [2003]; Papi et al. [2008]).

In the rest of this section, we briefly discuss the impact of classes with inter-property constraints in several facets of object-oriented programming.

Initialisation Interface instances in TypeScript can be initialised by assigning any object. To ensure type safety, TIPC has to impose a restriction on those assignments: only object literal types may be assigned to interfaces. This way, the type system has an exact view of which properties are present and absent in the object. A class, on the other hand, can only be initialised by calling its constructor. As this automatically gives an exact view on present and absent parameters, the need for an object literal restriction is nullified.

Nominal Typing In the previous section, we elaborated on the effects of a nominal type system on the interfaces in TIPC. As classes are always used in combination with a nominal type system, the same effects apply: an instance can only be assigned to a variable when both expressions have the same class type, or when the source class is a subtype of the target class.

In TIPC, the type system uses information from `if` statements to be able to permit more property accesses and updates. To achieve this, interface types are narrowed by adding extra constraints to their constraint set. Because TIPC is structurally typed, narrowed interface instances can be used wherever the original interface is expected. Further research is necessary to investigate how this approach can be translated to a nominal type setting.

Design patterns When incorporating inter-property constraints into the class definitions of an object-oriented programming language, it is worthwhile to take a look at how inter-property constraints affect the software design patterns by the Gang of Four [Gamma et al., 1995].

One of the patterns is the builder pattern: a design pattern that can be used when the initialisation logic of a class is complex. This is especially useful in the light of inter-property constraints, where the initialisation of a class requires all presence constraints to be satisfied. In short, the builder pattern separates the construction of an object from its representation by delegating the initialisation logic to a separate *builder* class. The builder class has several methods to configure the object, and a `build` method that actually constructs the object.

Listing 10.10 shows an example of how the builder pattern can be used by developers. The object `privateMessageBuilder` is an instance of the builder class for the extended `PrivateMessage` class in which both sender and receiver must be identified, either by the user ID or the screen name. It contains several configuration methods to set the values of message (`setText`), the sender (`setSenderId` or `setSenderName`) and the receiver (`setReceiverId` and `setReceiverName`).

```
1 let pm1: PrivateMessage = privateMessageBuilder.setText("text")
2                                                .setSenderId(42)
3                                                .setReceiverName("Bob")
4                                                .build(); //OK
5
6 let pm2: PrivateMessage = privateMessageBuilder.setText("text")
7                                                .setSenderId(42)
8                                                .setSenderName("Bob")
9                                                .build(); //ERROR
```

Listing 10.10: Constructing a `PrivateMessage` object with a builder class

We envision that at the end of the construction (i.e. when the `build` method is called), the builder object (statically) verifies whether all constraints of the original object are satisfied. In Listing 10.10, the first build chain results in a valid `PrivateMessage` object, as all presence constraints are satisfied. In the second build chain (in the assignment of `pm2`), the exclusivity constraints on both the sender and the receiver are violated, and this assignment should be rejected by the type system. To achieve this, inspiration might be drawn from the work on delayed types and recency types [Fahndrich and Xia, 2007; Heidegger and Thiemann, 2010].

10.4 Concluding Remarks

Throughout the history of computer science, libraries enable reusability of code and the incorporation of third-party functionality. They provide their functionality via an Application Programming Interface (API), which forms the contract between the user and the library. In order for an API call to succeed, the constraints imposed by the library need to be satisfied. These constraints include restrictions on the values, types and presence of properties. Most of these constraints are clearly listed in the documentation of APIs, supported by specification languages and well-studied in the domain of type systems.

In this dissertation, we focus on an —until now— unexplored kind of constraint: constraints *between* properties. From a survey of the documentation of APIs, we determined that these constraints are all combined using operators

from propositional logic. The most common kinds of constraints are exclusivity constraints (XOR), dependency constraints (implication), double implication constraints and NAND constraints.

To enable the compile-time verification of inter-property constraints, we extended the interface definitions in TypeScript, resulting in TIPC. This involved significant changes to the type system, in order to ensure that objects at runtime always satisfy their interface constraints. However, the impact on the expressivity of the programming language remains minimal.

To enable the runtime verification of inter-property constraints, we have extended the machine-readable OpenAPI specification language, resulting in OAS-IP. This enables the definition of inter-property constraints outside of statically typed programming languages. Dynamically typed programming languages can use tools generated from a specification file to automatically verify data against the specification.

This dissertation aims to be a stepping stone for mainstream programming languages to support constraints between properties. We hope to open up research avenues to integrate full-flexed inter-property constraints into the type system without affecting the expressivity of the programming language. In addition, we intend for OAS-IP to serve as a starting point to integrate inter-property constraints in specification languages. Together, this results in development tools that are better tailored to the needs found in the web development domain.

```
class PrivateMessage {
  @C(present(text))
  @C(present(user_id) XOR present(screen_name))
  private text: string;
  private user_id: number;
  private screen_name: string

  @C(present(msg))
  @C(present(id) XOR present(name))
  constructor(txt: string, id: number, name: string) {
    this.text        = txt;
    this.user_id     = id;
    this.screen_name = name;
  }

  @C(present(txt))
  setText(txt: string) {
    this.text = txt;
  }

  @C(present(id) XOR present(name))
  setUser(id: number, name: string) {
    this.user_id = id;
    this.screen_name = name;
  }

  getText(): string {
    return this.text;
  }

  getUserId(): number | undefined {
    return this.user_id;
  }

  getScreenName(): string | undefined {
    return this.screen_name;
  }
}

let pm1 = new PrivateMessage("Hello!", 42, undefined);        //OK
let pm2 = new PrivateMessage("Hello!", undefined, "Alice");   //OK
let pm3 = new PrivateMessage("Hello!", undefined, undefined);//ERROR
let pm4 = new PrivateMessage("Hello!", 42, "Alice");          //ERROR
```

Listing 10.9: A design for TypeScript classes with inter-property constraints

Appendices

Appendix A

Object Literal Restriction

This appendix shows a small study for the object literal restriction in TIPC (explained in Section 4.1). This study was performed in May 2017. It consists out of code snippets from the first 128 search results on the code repository website `https://github.com/` for the keyword '`twitter-node-client`' and the first 112 search results for the keyword '`gapi.client.youtube`'. The former is an SDK (Software Development Kit) for Twitter, and the latter is the YouTube SDK. In every project, we looked for every usage of those SDKs and checked whether the data for a request call was an object literal. In case it was not an object literal, we checked whether the object was defined right above the request call or not.

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal
https://github.com/agraff/twitter-midi	`twitter.getTweet({ id: '1111111111'}, error, success);`	yes
https://github.com/arjovanramshorst/chatbot	`const data = twitter.getSearch({ 'q': '#delft', 'count': 100 }`	yes
https://github.com/artem-d/appdirect_twitter_test	`twitter_client.getUserTimeline({ screen_name: req.query.twitterAccount, count: req.query.tweetCount }`	yes
https://github.com/boynux/dahoam16/	`twitter.getSearch({'q': '#dahoam16', 'count': 100, 'since_id': state.max_id, result_type: 'recent'}, error, success);`	yes
https://github.com/bradturner3/twittertr	`var tweet = { status: status } // Posting the tweet twitter.postTweet(tweet, console.log, console.log)`	no, right above
https://github.com/dancinturtle/SecondPizza	`var data = twitter.getUser({ screen_name: username}, function(error, response, body){`	yes
https://github.com/Flyette/API-Twitter	`tweets = twitter.getMentionsTimeline({ count: '10'}, error, success);`	yes
https://github.com/gfjalar/nyuadhack-bot	`var tweet = { status: status } // Posting the tweet twitter.postTweet(tweet, console.log, console.log)`	no, right above
https://github.com/krissrex/ISFiT-2017-projector	`tRest.getSearch({'q':`${TAGS.join(' OR ')} OR ${ USER_NAMES.map((it) => 'from:' + it).join(' OR ') } since:2017-02-9 until:2017-02-20`, 'count': 5, 'result\_type':'recent'},`	yes
https://github.com/lsortiz/test-repository-2	`twitter.getMentionsTimeline({ count: '1'}, error, success);`	yes
https://github.com/majetisiri/Twitter-Client	`twitter.getUserTimeline({ screen_name:screenName}, error, success);`	yes
https://github.com/marvinrenaud/liri-node-app	`twitter.getUserTimeline({ screen_name: 'MrMarcReno', count: '10' }`	yes
https://github.com/mrjones91/Monthly-Challenge	`var thingiething = twitter.getUser({screen_name: data.screen_name}, boundError, boundSuccess);`	yes
https://github.com/ralphbs/Loop	`params = {screen_name: news_source, count: '50', exclude_replies: 'true'}; twitter.getUserTimeline(params, error, function(timeline){ timeline = JSON.parse(timeline); var number_of_users = 0; timeline.forEach(function(entry){ number_of_users++; })`	no, right above
https://github.com/robbiemu/CMST301-project2	`twitter.getTweet({id}, reject, resolve)`	yes, but special kind of object literal({id} results in {id: value-of-id}
https://github.com/sschand/ThirdHome	`var data = twitter.getUser({ screen_name: username}, function(error, response, body){ res.status(404).send({ "error" : "User Not Found" }); }, function(data){ res.send({ result : { "userData" : data } }); });`	yes
https://github.com/tigranbs/oneday	`twitter.getSearch({ q: 'since:' + date.toMysqlFormat() , geocode: city.latitude + "," + city.longitude + ",15000mi" , count: 100 , result_type: "recent" }`	yes
https://github.com/volokasse/GrowthHackingTwitter	`twitter.getUser({'screen_name': 'Volokasse'}`	yes
https://github.com/yingqiaogit/tweetanalysis	`twitterHdl.getUser({screen_name: username}`	yes
	`var query = { 'q': '#nyuadhack -filter:retweets -filter:replies' + lastFilter, 'result\_type': 'recent', 'count': 100 } // Searching for tweets matching the query twitter.getSearch(query, console.log, successCallback)`	no, right above
	`var data = twitter.getSearch({'q':'#'+hashname, 'count': 10}, function(error, response, body){ res.status(404).send({ "error" : error });`	yes
	`var data = twitter.getSearch({'q':'#'+hashname, 'count': 5},`	yes
	`var query = { 'q': '#nyuadhack -filter:retweets -filter:replies' + lastFilter, 'result\_type': 'recent', 'count': 100 } // Searching for tweets matching the query twitter.getSearch(query, console.log, successCallback)`	no, right above
	`twitter.getSearch({'q':'#tocados','count': 4}, error, success_busqueda);`	yes
https://github.com/ACECentre/SpeechBubble/	`twitter.getUserTimeline( { screen_name: 'acecentre', count: limit },`	yes
https://github.com/Jakehp/tweethorizon	`twitterClient.getUserTimeline({ screen_name: handle, count: '1000'}, error, success);`	yes
	`twitterClient.getUserTimeline({ screen_name: handle, count: '1'}, error, success);`	yes
https://github.com/dday34/twitter-report/	`twitter.getSearch({'q':topic,'count': 10}, error, handleTweetsResponse(success));`	yes
https://github.com/cbadal/site1/	`twitter.getUserTimeline({ screen_name: 'designmami', count: '30'}`	yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal

https://github.com/AravindVenkataraman/Twitter-Integration-Project	twitter.getUserTimeline({
	screen_name: param.screen_name,
	since: param.since,
	until: param.until,
	count: param.count
	}
		yes

https://github.com/thiagoxvo/fisl-twitter-app	twitter.getSearch({'q': hashtag,'count': 100}	yes
https://github.com/fslone/search	twitter.getSearch({"q":req.params.query, "count": 10, "result_type":"popular"},	yes
https://github.com/BrandonLittell/JadenSmith	twitter.getUserTimeline({ screen_name: _id, count: 20, max_id: tweetsCache.get(_id).lastTweet, exclude_replies: true, include_rts: false}	yes
	twitter.getUserTimeline({ screen_name: _id, count: 20, exclude_replies: true, include_rts: false}	yes
https://github.com/lansingcodes/lubot	twitter-client.post-tweet do status: text	yes
https://github.com/mattiasewers/material-todo	twitter.getUser('/users/show.json',{ 'screen_name': username}	yes
	twitter.getUserTimeline('/statuses/user_timeline.json',{ 'screen_name': username, count: '10'}	yes
https://github.com/elaineo/personalityTest	twitter.getUserTimeline({ screen_name: name, include_rts: 'false', count: '2000'}	yes

https://github.com/chrisxwan/ipsify	var options = {
	screen_name: twitterHandle,
	count: 40 * numParagraphs,
	exclude_replies: true,
	include_rts: false
	}
	twitter.getUserTimeline(options, function (data)
		no, right above

https://github.com/traxedas/habana	twitter.getUserTimeline({ screen_name: '', count: '10'}	yes
	twitter.postTweet({ status: post}	yes
https://github.com/lernerd/my-nodejs-code	twitter.getSearch({'q':cfg.query, 'lang':'en', 'count': 100, 'result_type':'recent'}, twitter_error, twitter_success);}	yes
https://github.com/Vheissu/aurelia-fortune	twitter.getSearch({'q':'#wisdom OR #advice OR #quote -RT -retweet -@ -heaven -god -http -bible: -https:', 'result_type': 'recent', 'lang':'en', 'count': 100}, error, success);	yes
https://github.com/sarvesh123/04082015	twitter.postTweet({ status: tweet}, error, success);	yes

https://github.com/troyef/AADemo	var options = {
	q: keyword,
	count: 100, include_entities: false,
	};
	twitter.getSearch(options, error, function(results){
	callback.call(undefined, JSON.parse(results));
	});
		no, right above

https://github.com/anapaulagomes/tweetcast	twitter.getSearch({'q':'#DevFestSudeste','count': 10}, error, success);	yes

https://github.com/peterhudec/twitter-geo-search	twitter.getSearch({
	q: req.query.q,
	count: 100,
	result_type: 'recent',
	geocode: req.query.geocode
	}
		yes

https://github.com/wanadev/wanashare	twitter.postMedia({media_data: media.get().toString("base64")},	yes
	twitter.postTweet({ status: message, media_ids: [media_id] },	yes

https://github.com/rja-xx/Govie	twitter.getUser({screen_name: user.username}, function (err) {
	res.status(500).json({message: "Server error!", errors: [err]});
	return res;
	}
		yes

https://github.com/jbbskinny/JBEngine	twitter.getSearch({'q': searchString, 'count': 10}, error, success);	yes
https://github.com/thinkocapo/twitterquery	twitter.getUserTimeline({screen_name: handle , count: numTweets },	yes
https://github.com/rohitsakala/SMAI_Project	twitter.getSearch({'q':'Lebron James','count': 10}, error, success);	yes
https://github.com/nagahar/school_search	twitter.getSearch({'q': q.msg, 'count': 10, 'lang': 'ja'}	yes

https://github.com/jovinbm/Ulan182	twitter.getUser({
	user_id: options.profile.id
	}
		yes

https://github.com/tedpennings/site-lambdas	const timelineOpts = {
	screen_name: 'thesleepyvegan',
	count: 50,
	exclude_replies: true,
	include_rts: false,
	trim_user: true
	}
	twitter.getUserTimeline(timelineOpts,
		yes

https://github.com/yconoclast/twitterfinder	twitter.getUserTimeline(
	{
	screen_name: req.body.username,
	count: '200',
	contributor_details: false,
	trim_user: true,
	include_rts: true
	}
		yes

https://github.com/sarahbethfederman/Technical-Interview-Data	twitter.getCustomApiCall('/statuses/retweets.json',{ id: '638769329903960064', count: 100 }, onError, onSuccess);	yes
https://github.com/Aditya-Shibrady/TADAPP	twitter.getUserTimeline({ screen_name: 'aditya_shibrady', count: '2'}, error, success);	yes
https://github.com/samundrak/periscope-livestream-mobile-to-web	twitter.getHomeTimeline({ count: '1'}, error, success);	yes
https://github.com/MarkusSN/hackathon-netlight	twitter.getSearch({ q: decodeURIComponent(req.params.query), count: 100, lang: 'en' }, success, success.bind(res));	yes
https://github.com/asthinasthi/twitter_trends	twitter.getSearch({'q':'#anirudh', 'count':10}	yes
	twitter.getCustomApiCall('/trends/place.json', {id: woeid}	yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal
	`twitter.getCustomApiCall('/trends/available.json',{}`	yes
	`twitter.getCustomApiCall('/search/tweets.json',{'q':topic, 'woeid':woeid}`	yes

https://github.com/guardian/metagu	`const tweet = {`
	`status: \`@${recipient} ${message}\`,`
	`in_reply_to_status_id: replyTweetId`
	`};`
	`twitter.postTweet(`
	`tweet,`
	`error => { observer.onError(error); observer.onCompleted(); },`
	`resp => { observer.onNext(resp); observer.onCompleted(); }`
	`);`
		no, right above

https://github.com/danielsomekh/capstone	`twitter.getSearch({'q': searchQuery,'count': 6, 'filter':'images', 'lang': 'en', 'result_type': 'recent'},`	yes

https://github.com/jonbstrong/upload-twitter-list	`twitter.listAddMembers({`
	`owner_screen_name:"YOUR_HANDLE",`
	`slug:"LIST_NAME_HERE",`
	`screen_name:"HANDLE1,HANDLE2,HANDLE3"`
	`}`
		yes

https://github.com/jcadruvi/TwitterApp	`twitter.getUserTimeline(` `{screen_name: 'joshcadruvi', count: '200', trim_user: true},`	yes
https://github.com/ulfmagnetics/one_word_weather_api	`var twitterOpts = {screen_name: options.screenName, count: '1'};` `twitter.getUserTimeline(twitterOpts,`	yes
https://github.com/tyholby/CS201	`twitter.getUserTimeline({ screen_name: req.query.user, count: '10'},`	yes
https://github.com/Flyette/integration-reseaux-sociaux	`twitter.getMentionsTimeline({ count: '1'}, error, success);`	yes

https://github.com/Flavien94/Exos	`twitterClient.getSearch({`
	`'q': '#marseille',`
	`'count': 20,`
	`'geocode': '43,5,200mi',`
	`}`
		yes

https://github.com/bingoooo/simplonMIP-test	`twitter.getMentionsTimeline({ count: '1'}, error, success);`	yes

https://github.com/stijnvanhulle/ControlYourHome	`twitter.getHomeTimeline({ count: '10'}, function (err, response, body) {`
	`res.json({success:false,error:err});`
	`}`
		yes

https://github.com/pbooth01/Twitter-User-Finder	`twitter.getSearch({'q':'#haiku','count': 10}`	yes
	`twitter.getCustomApiCall('/users/search.json',{q:searchString, include_entities: 'false'}`	yes
	`twitter.getCustomApiCall(('/statuses/user_timeline.json'),{screen_name: searchString, count: '50'}`	yes
	`twitter.getUser({ screen_name: searchString}`	yes
https://github.com/schen384/cs3750-twitter-test	`twitter.getHomeTimeline({ count: '10'}, error, getFeed)`	yes
https://github.com/shahgurpreet/WebDevSpring2016	`twitter.getCustomApiCall('/search/tweets.json?max_id='+token+'&q='+tag+'&include_entities=true&count=33&filter=images',{}, error, success);`	yes
https://github.com/MeeralQureshi/UofTHacks2016	`var tweet = twitter.getTweet({ id: '690978123861626880'}, error, success);`	yes
https://github.com/KennyMack/authentication	`witter.getHomeTimeline({ count: '10', page: 1},`	yes

https://github.com/Karine-Jamet/ex-leaftlet	`twitter.getSearch({`
	`'q': '',`
	`'geocode': coord,`
	`'count': 10`
	`}`
		yes

https://github.com/chiquelo/tweetfetcher	`twitter.getSearch({`
	`'q': possibleSearches[1],`
	`'result_type': 'popular',`
	`'lang': 'en'`
	`},`
		yes

https://github.com/jdormit/eighthnotegames-site	`twitter.getUserTimeline({screen_name: user, count: tweet_count, exclude_replies: true},`	yes

https://github.com/kvonhorn/tripper	`var searchRequest = gapi.client.youtube.search.list({`
	`part: 'snippet', q: searchTerm, maxResults: Session.get("resultsPerPage")`
	`});`
		yes

https://github.com/falr95/pulsion	`gapi.client.youtube.search.list({part:"snippet", q:"gMdJzUVHRwU | D0BsgJxw208"})`	yes

https://github.com/dpiatek/youtube-playlister	`var params = {}, request;`
	
	`if (!gapiLoaded || !query) return;`
	`params.q = query;`
	`params.part = 'snippet';`
	`params.type = 'video';`
	`request = gapi.client.youtube.search.list(params);`
		no, right above

https://github.com/zarkhaari/ngyoutube	`gapi.client.youtube.search.list({`
	`q: 'dogs',`
	`part: 'snippet'`
	`});`
		yes

https://github.com/EdS0ng/Ionic-YoutubeMusicPlayer	`gapi.client.youtube.playlists.list({`
	`part: 'snippet, contentDetails',`
	`mine: true,`
	`maxResults:20`
	`})`
		yes

	`gapi.client.youtube.playlistItems.list({`
	`part: 'snippet',`
	`playlistId:id,`
	`maxResults:50`
	`})`
		yes

	`gapi.client.youtube.videos.list({`
	`part:'snippet, contentDetails',`
	`regionCode:'US',`
	`chart:'mostPopular',`
	`maxResults:10`
	`})`
		yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal
	gapi.client.youtube.videos.list({ part:'snippet, contentDetails', regionCode: 'US', maxResults: 10, chart: 'mostPopular', pageToken: results.nextPageToken });	yes
	gapi.client.youtube.playlistItems.list({ part:'snippet', playlistId: id, maxResults:20 })	yes
	gapi.client.youtube.playlistItems.list({ playlistId:query, part:'snippet', maxResults: 20, pageToken: results.nextPageToken });	yes
	gapi.client.youtube.search.list({ q:query, part:'snippet', type: 'video', maxResults: 20 });	yes
	gapi.client.youtube.search.list({ q:query, part:'snippet', type: 'video', maxResults: 20, pageToken: results.nextPageToken });	yes
	gapi.client.youtube.playlistItems.list({ part:'snippet', playlistId: id, maxResults: 50 })	yes
	gapi.client.youtube.activities.list({ part:'snippet, contentDetails', maxResults: 10, home:true });	yes
	gapi.client.youtube.playlistItems.list({ part:'snippet', playlistId: id, maxResults: 50 })	yes
	gapi.client.youtube.channels.list({ part:'contentDetails, snippet', mine:true });	yes

	gapi.client.youtube.playlistItems.insert({
	part:"snippet",
	resource:{
	"snippet":{
	"playlistId": playlistId,
	"resourceId": {
	"kind":"youtube#video",
	"videoId": videoId
	}
	}
	}
	})
		yes

	gapi.client.youtube.videos.list({ part:'contentDetails', id: videoId });	yes
	gapi.client.youtube.search.list({ relatedToVideoId: videoId, part:'snippet', type: 'video', maxResults: 50 });	yes
	gapi.client.youtube.videos.rate({ id:link, rating:str });	yes
https://github.com/julioz2/film-finder	var video = gapi.client.youtube.search.list({ part: "snippet", type: "video", q: query, maxResults: 1 });	yes
https://github.com/ICodeMyOwnLife	gapi.client.youtube.search.list({ q: 'pokemon', part: 'snippet' });	yes
https://github.com/conchan/favsmr	gapi.client.youtube.search.list({ q: helperTags + currentCategory, order: order, type: "video", part: "snippet", maxResults: "10" });	yes
https://github.com/wjm9696/rebuilt_youtube	gapi.client.youtube.search.list({ q: q, part: 'snippet', type: 'video', maxResults: 20 });	yes
https://github.com/kevinkorte/discover	gapi.client.youtube.videos.list({ id: id, part: 'snippet' });	yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal
https://github.com/mnivelles/UMovie-Project--Client--team04	`var request = gapi.client.youtube.search.list({ q: q, part: 'snippet', fs:'0' });`	yes
https://github.com/lidorl/dy-youtuber	`gapi.client.youtube.search.list({ q: params, part: 'snippet', maxResults: 50, type: 'video' });`	yes
https://github.com/ICodeMyOwnLife/YoutubeAPI	`gapi.client.youtube.search.list({ q: q, part: 'snippet' });`	yes
https://github.com/Mr21/youtube-playlists-manager	`gapi.client.youtube.search.list, { type: 'channel', part: 'snippet', q: name },`	yes
	`queryParams = { part: 'snippet,status,contentDetails', maxResults: 50 }; if (channelId) queryParams.channelId = channelId; else queryParams.mine = true; gapi.client.youtube.playlists.list, queryParams,`	no
	`body = { part: 'snippet', resource: { id: pl.id, snippet: { title: this.newName ? this.newName : this.name } } }; if (this.newPrivacy) { body.part += ',status'; body.resource.status = { privacyStatus: this.newPrivacy }; } gapi.client.youtube.playlists.update, body,`	no
	`body = {}, type = this.status === 'del' ? 'delete' : this.status === 'add' ? 'insert' : 'update'; if (type !== 'delete') { body.part = 'snippet'; body.snippet = { playlistId: pl.id, position: this.posB - 1, resourceId: { kind: this.video.kind, videoId: this.video.videoId }, }; } if (type !== 'insert') body.id = this.video.id; body = [gapi.client.youtube.playlistItems[type], body,`	no
	`gapi.client.youtube.playlistItems.list, { playlistId: id, part: 'snippet', maxResults: 50 },`	yes
https://github.com/GLO3102H15/team04	`gapi.client.youtube.search.list({ q: q, part: 'snippet', fs:'0' });`	yes
https://github.com/madeinfree/musictron	`gapi.client.request({ method: 'GET', path: '/youtube/v3/search', params: { part: 'snippet', maxResults: 50, type: 'video', order: 'date', videoDuration: 'short', q: params.q } }`	yes
	`gapi.client.request({ method: 'GET', path: 'youtube/v3/videos', params: { part: 'contentDetails', id: params.videoId } })`	yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal

https://github.com/AndersFly666/andersfly666.github.io	$http.get('https://www.googleapis.com/youtube/v3/search',{
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	type: 'video',
	maxResults : '48',
	orderBy: 'viewsCount',
	part: 'snippet',
	videoEmbeddable: true,
	q: $scope.query
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/videos', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet',
	chart: 'mostPopular',
	maxResults: '42'
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/videos', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, statistics',
	chart: 'mostPopular',
	maxResults: '12',
	pageToken: $scope.data.nextPageToken
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/search', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet',
	maxResults: '12',
	type: 'video',
	orderBy: 'viewsCount',
	videoEmbeddable: true,
	relatedToVideoId: $scope.video.videoId
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/videos', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'statistics',
	id: $scope.video.videoId,
	fields: 'items/statistics'
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/channels', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	mine: true,
	access_token: youtubeService.getToken(),
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/playlistItems', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	playlistId: $scope.history,
	maxResults: 30,
	access_token: youtubeService.getToken()
	
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/playlistItems', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	playlistId: $scope.favorites,
	maxResults: 21
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/subscriptions', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	mine: true,
	order: 'unread',
	maxResults: 21,
	access_token: youtubeService.getToken()
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/playlistItems', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	playlistId: $scope.favorites,
	maxResults: 3,
	pageToken: $scope.favData.nextPageToken
	}
	})
		yes

	$http.get('https://www.googleapis.com/youtube/v3/subscriptions', {
	params: {
	key: 'AIzaSyC15w1zx5-BlDbO_DSBrv7DDrizxSTtwn8',
	part: 'snippet, contentDetails',
	mine: true,
	order: 'unread',
	maxResults: 3,
	access_token: youtubeService.getToken(),
	pageToken: $scope.subData.nextPageToken
	}
	})
		yes

Github projects that use a Twitter SDK called "twitter-node-client" and the Youtube SDK	Code snippet	Argument = object literal

https://github.com/vkolova/tuber	`this.gapi.client.request({`
	`  path: '/youtube/v3/channels',`
	`  params: {`
	`  part: 'snippet',`
	`  mine: true`
	`  },`
		yes

	`this.gapi.client.request({`
	`  path: '/youtube/v3/videos',`
	`  params: {`
	`  part: 'status,player',`
	`  id: this.videoId`
	`  },`
		yes

	`gapi.client.youtube.channels.list({` `  mine: true, part: 'id, contentDetails, statistics, snippet, brandingSettings'});`	yes

	`var requestOptions = {`
	`  playlistId: playlistId,`
	`  part: 'snippet, contentDetails',`
	`  maxResults: 10`
	`  };`
	`gapi.client.youtube.playlistItems.list(requestOptions);`
		no, right above

	`  var requestOptions = {`
	`  channelId: channelId,`
	`  mine: true,`
	`  part: 'snippet, contentDetails, status, contentDetails',`
	`  maxResults: 10`
	`  };`
	`  var request = gapi.client.youtube.playlists.list(requestOptions);`
		no, right above

https://github.com/loikg/banana	`gapi.client.youtube.search.list({`
	`  q: query,`
	`  part: 'snippet'`
	`  })`
		yes

https://github.com/JuanDaniel1995/web_kids	`gapi.client.youtube.playlistItems.insert({`
	`  part: 'snippet',`
	`  resource: {`
	`  snippet: {`
	`  playlistId: playlist_id,`
	`  resourceId: details`
	`  }`
	`  }`
	`  });`
		yes

	`gapi.client.youtube.playlists.insert({`
	`  part: 'snippet,status',`
	`  resource: {`
	`  snippet: {`
	`  title: title,`
	`  description: 'A private playlist created with the YouTube API'`
	`  },`
	`  status: {`
	`  privacyStatus: 'private'`
	`  }`
	`  }`
	`});`
		yes

	`gapi.client.youtube.playlists.insert({`
	`  part: 'snippet,status',`
	`  resource: {`
	`  snippet: {`
	`  title: title,`
	`  description: 'A private playlist created with the YouTube API'`
	`  },`
	`  status: {`
	`  privacyStatus: 'private'`
	`  }`
	`  }`
	`});`
		yes

https://github.com/WesAspinall/youtube-playlist	`gapi.client.youtube.playlistItems.list({`
	`  part: 'snippet',`
	`  playlistId: plistId,`
	`  maxResults: 8`
	`  });`
		yes

https://github.com/fachhoch/testproject567678	`gapi.client.youtube.search.list({`
	`  q: searchText,`
	`  part: 'snippet',`
	`  maxResults:'50'`
	`  });`
		yes

	`gapi.client.youtube.search.list({`
	`  q: q,`
	`  part: 'snippet'`
	`});`
		yes

https://github.com/tom76kimo/video-wall	`gapi.client.youtube.activities.list({`
	`  home: true,`
	`  part: 'snippet',`
	`  maxResults: 10`
	`  });`
		yes

	`gapi.client.youtube.channels.list({`
	`  part: 'contentDetails',`
	`  id: channelId,`
	`  maxResults: 20`
	`  });`
		yes

	`gapi.client.youtube.playlistItems.list({`
	`  part: 'snippet',`
	`  playlistId: playlistId,`
	`  maxResults: 10`
	`  });`
		yes

https://github.com/anilarya/wittyParrot	`gapi.client.youtube.search.list({`
	`  part: 'snippet',`
	`  channelId: 'UCqhNRDQE_fqBDBwsvmT8cTg',`
	`  order: 'date',`
	`  type: 'video'`
	`  })`
		yes

Github projects that use a Twitter SDK called “twitter-node-client” and the Youtube SDK	Code snippet	Argument = object literal
https://github.com/rejishyni/rgbano	gapi.client.youtube.search.list({ q: searchtext, part: 'snippet', type:'video', order: sortVal, maxResults: 10, location: loc, locationRadius: locRad });	yes
	gapi.client.youtube.videos.list({ id: videoId, part: 'snippet,statistics' });	yes
	gapi.client.youtube.videos.list({ id: videoId, part: 'snippet,statistics' });	yes
https://github.com/WestBrian/Tubify	gapi.client.youtube.search.list({ q: query, part: 'snippet', type: 'video', maxResults: '15' });	yes
	gapi.client.youtube.search.list({ q: query, part: 'snippet', type: 'video', maxResults: '15' });	yes
https://github.com/prafmathur/TwilioMusicPlayer	gapi.client.youtube.search.list({ q: query, part: 'snippet' });	yes
https://github.com/Millsky/godj	$scope.client.search.list({ q: $scope.searchParams, part: 'snippet' });	yes
https://github.com/webinos-apps/webinos-appChallenge	gapi.client.youtube.search.list({q: searchTerm,part: 'snippet' });	yes
https://github.com/hackfoldr/hackfoldr	request = gapi.client.youtube.videos.list({'id': it.id, 'part':'snippet'}	yes
https://github.com/codeNovels/Devops-Dashboard	gapi.client.youtube.playlistItems.list({ part: 'snippet', playlistId: 'UUpEYMEafq3FsKCQXNliFY9A' maxResults: 10 });	yes
https://github.com/piclemx/GLO3102-team2	gapi.client.youtube.search.list(part: 'snippet', maxResults: 1, q: query });	yes
https://github.com/mikedcm/MostlyJs	gapi.client.youtube.search.list({ part:"snippet", fields:"pageInfo(totalResults),items(id(videoId),snippet(title,description))", chart:"mostPopular", regionCode:"GB", type:"video", q:encodeURIComponent("Jaguar"), order:"viewCount", publishedAfter:"2015-01-01T00:00:00Z" });	yes
	gapi.client.youtube.search.list({ part:"snippet", type:"video", q:encodeURIComponent("nice"), order:"viewCount", publishedAfter:"2015-01-01T00:00:00Z" });	yes
https://github.com/ICodeMyOwnLife/typings	gapi.client.youtube.search.list({ q: 'pokemon', part: 'snippet' });	yes
https://github.com/cinoyter/cherry	gapi.client.youtube.videos.list({ part: "snippet", id: id });	yes
https://github.com/chriscasper/Kittentopia	gapi.client.youtube.search.list({ part: 'snippet', q: 'kittens', maxResults: 8, safeSearch: 'strict', pageToken: pageToken });	yes
https://github.com/KoenVanDerHoeven/Maximinute	gapi.client.youtube.search.list({ q: q, part: 'snippet' });	yes
https://github.com/JTGeek/thinkfultube	gapi.client.youtube.search.list({ q: q, part: 'snippet', type: 'video', });	yes

https://github.com/jcapps/youtube-channel-angular2	var subscribeParams = {
	'part': 'id, snippet',
	'snippet': {
	'resourceId': {
	'kind': 'youtube#channel',
	'channelId': CHANNEL_ID
	}
	}
	};
	gapi.client.youtube.subscriptions.insert(subscribeParams);
		no, right above

Github projects that use a Twitter SDK called “twitter-node-client” and the Youtube SDK	Code snippet	Argument = object literal
https://github.com/laalex/youtube2mp3	gapi.client.youtube.search.list({ q: keywords, part: 'snippet' });	yes
https://github.com/monojack/ewtube-ng	gapi.client.youtube.subscriptions .list({ mine: true, part: 'snippet,contentDetails', maxResults: count })	yes
https://github.com/iuriifavi/favirocks	_gapi.client.request({ 'path': 'https://content.googlapis.com/youtube/v3/channels? part=snippet%2CcontentDetails', 'params': ['channelId=UCGh4Gh0FxKMP1-rDnUTUtZQ'] })	yes

https://github.com/nosmit01/samples	gapi.client.youtube.search.list({ // search videos
	q: search.keyword,
	part: 'snippet',
	location: lat + ',' + long,
	locationRadius: '100mi',
	type: 'video',
	maxResults: 50,
	pageToken: nextPage
	});
		yes

	gapi.client.youtube.videos.list({ // get video details
	id: id,
	part: 'snippet, player, statistics'
	});
		yes

	gapi.client.youtube.commentThreads.list({ // get comments
	videoId: id,
	part: 'snippet'
	});
		yes

	gapi.client.youtube.search.list({ // search videos
	q: search.keyword,
	part: 'snippet',
	location: lat + ',' + long,
	locationRadius: '100mi',
	type: 'video',
	maxResults: 50,
	pageToken: nextPage
	});
		yes

	gapi.client.youtube.videos.list({ // get video details
	id: id,
	part: 'snippet, player, statistics'
	});
		yes

	gapi.client.youtube.commentThreads.list({ // get comments
	videoId: id,
	part: 'snippet'
	});
		yes

https://github.com/langostinko/torrent-parser	gapi.client.youtube.search.list({ q: "<?=html_entity_decode($title)?> <?=$desc['Year']?> трейлер", part: 'id', type: 'video', publishedAfter: "<?=date(DateTime::RFC3339, $movie['Release'] - 3600*24*365)?>" });	yes

https://github.com/treant-prime/yt-crawler	requestParameters = {
	part: 'snippet',
	chanelId: 'UCuQZ-VMez8stUmNvNDfpV7A',
	maxResults: 50,
	mine: true
	};
	request = gapi.client.youtube.playlists.list(requestParameters);
		no, right above

	requestParameters = {
	part: 'snippet',
	playlistId: id,
	maxResults: 50
	};
	request = gapi.client.youtube.playlistItems.list(requestParameters);
		no, right above

Appendix B

Type Preservation

B.1 Type Preservation of Expressions

Theorem 1 (Type Preservation for Expressions)**.** If $\Sigma \models \langle H, L, \mathtt{e}\rangle : \mathtt{T}$ and $\langle H, L, \mathtt{e}\rangle \Downarrow \langle H', r\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', r\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \mathtt{T}$.

Proof. By case analysis on the evaluation rules. We start from the givens and deconstruct it into parts, notably we reconstruct the typing environment from the *context*, which is a combination of the heap type and the scope chain. The combination of typing environment and typing rule for the given expression yields a number of typing derivations on the components of said rule. We apply the induction hypothesis to these components and focus on the remaining proof obligations.

Case E-Id

Assume: $\Sigma \models \langle H, L, \mathtt{x}\rangle : \mathtt{T}$ (1)

Prove: $\Sigma \models \langle H, (l, \mathtt{x})\rangle : \mathtt{T}$ (2)

From (1):

$$\dfrac{\dfrac{(3)}{\Sigma \models H} \quad \dfrac{(4)}{H, L \models \diamond} \quad \dfrac{\dfrac{\dfrac{(5)}{\mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}}}{\Gamma', \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}}}{\Gamma \vdash \mathtt{x} : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{x}\rangle : \mathtt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (2):

where $\sigma(H, L, \mathtt{x}) = l$ (6)

$$\dfrac{\dfrac{}{\Sigma \models H}\,(3) \qquad \dfrac{\dfrac{}{\mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}}\,(5)}{\Sigma(l) \vdash \mathtt{x} : \mathtt{T}}\,(7)}{\Sigma \models \langle H, (l, \mathtt{x}) \rangle : \mathtt{T}}$$

We need to show that (5) follows from (7). The construction of *context* guarantees that for every variable the binding of the most inner scoped is used, which coincides with they way σ looks up variables in the scope chain. Therefore, (7) follows from (6).

Note that (4) is of key importance to the correct working of σ, as it guarantees that the scope chain is well-formed: every element is an object map in the heap. This is elided in further proofs.

Case E-Lit

Assume: $\Sigma \models \langle H, L, \mathtt{l} \rangle : \mathtt{T}$ (8)

Prove: $\Sigma \models \langle H, \mathtt{l} \rangle : \mathtt{T}$ (9)

From (8):

$$\dfrac{\dfrac{(10)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(11)}{\vdash \mathtt{l} : \mathtt{T}}}{\mathit{context}(\Sigma, L) \vdash \mathtt{l} : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{l} \rangle : \mathtt{T}}$$

To prove (9) :

$$\dfrac{\dfrac{}{\Sigma \models H}\,(10) \qquad \dfrac{}{\vdash \mathtt{l} : \mathtt{T}}\,(11)}{\Sigma \models \langle H, \mathtt{l} \rangle : \mathtt{T}}$$

Case E-This

Assume: $\Sigma \models \langle H, L, \mathtt{this} \rangle : \mathtt{T}$ (12)

Prove: $\Sigma \models \langle H, l \rangle : \mathtt{T}$ (13)

From (12):

$$\dfrac{\dfrac{(14)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\Gamma, \mathtt{this} : \mathtt{T} \vdash \mathtt{this} : \mathtt{T}}{\mathit{context}(\Sigma, L) \vdash \mathtt{this} : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{this} \rangle : \mathtt{T}}$$

with $\sigma(H, L, \mathtt{x}) = l$ (15)

To prove (13) :

$$\dfrac{\dfrac{}{\Sigma \models H}\,(14) \qquad \dfrac{(16)}{\Sigma(l) = \mathtt{T}}}{\Sigma \models \langle H, l\rangle : \mathtt{T}}$$

(16) follows from (15) because of the correspondance between σ and Σ (see proof of preservation for E-Id).

Case E-Op

Assume: $\Sigma \models \langle H, L, \mathtt{e}_1 \otimes \mathtt{e}_2\rangle : \mathtt{T}$ (17)

Prove: $\Sigma' \models \langle H', \mathtt{l}_1 \otimes \mathtt{l}_2\rangle : \mathtt{T}$ (18)

$\Sigma \subseteq \Sigma'$ (19)

From (17):

$$\dfrac{\Sigma \models H \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(20)}{\Gamma \vdash \mathtt{e}_1 : \mathtt{S}_0} \quad \dfrac{(21)}{\Gamma \vdash \mathtt{e}_1 : \mathtt{S}_0} \quad \dfrac{(22)}{\mathtt{S}_0 \otimes \mathtt{S}_1 = \mathtt{T}}}{\Gamma \vdash \mathtt{e}_1 \otimes \mathtt{e}_2 : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{e}_1 \otimes \mathtt{e}_2\rangle : \mathtt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (18):

where $\langle H_0, L, \mathtt{e}_1\rangle \Downarrow_v \langle H_1, \mathtt{l}_1\rangle$ (23)

and $\langle H_1, L, \mathtt{e}_2\rangle \Downarrow_v \langle H_2, \mathtt{l}_2\rangle$ (24)

$$\dfrac{\dfrac{\text{IH on (23) and (24)}}{\Sigma' \models H'} \qquad \dfrac{\dfrac{\text{IH on (23)}}{\vdash \mathtt{l}_1 : \mathtt{S}_0} \quad \dfrac{\text{IH on (24)}}{\vdash \mathtt{l}_2 : \mathtt{S}_1} \quad \dfrac{}{\mathtt{S}_0 \otimes \mathtt{S}_1 = \mathtt{T}}\,(22)}{\vdash \mathtt{l}_1 \otimes \mathtt{l}_2 : \mathtt{T}}}{\Sigma' \models \langle H', \mathtt{l}_1 \otimes \mathtt{l}_2\rangle : \mathtt{T}}$$

(19) follows from the induction hypothesis on (23) and (24).

Case E-ObLit

Assume: $\Sigma \models \langle H, L, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}\rangle : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}$ (25)

Prove: $\Sigma' \models \langle H', l\rangle : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}$ (26)

$\Sigma \subseteq \Sigma'$ (27)

From (25):

$$\dfrac{\Sigma \models H \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(28)}{\mathit{context}(\Sigma, L) \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}}{\mathit{context}(\Sigma, L) \vdash \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\} : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}}{\Sigma \models \langle H, L, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}\rangle : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

To prove (26) :

where $H' = H[l \mapsto new(l)]$

$\langle H_1, L, \overline{\mathtt{e}} \rangle \Downarrow_v \langle H'_m, \overline{v} \rangle$ (29)

$$\frac{\dfrac{(30)}{\Sigma' \models H'} \qquad \dfrac{(31)}{\Sigma'(l) = \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}}{\Sigma' \models \langle H', l \rangle : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

From E-ObLit, we observe the following about changes to the heap and the heap type. The evaluation rule E-ObLit extends the heap with a new location. This location is fresh and contains an `@this` property, which points to itself. Thus, this is a well-formed extension of the heap. Moreover, the property `@this` is not taken into account by Σ. For the evaluation of properties, the induction hypothesis on (29) guarantees that H was extended (or remained identical) and that Σ remained compatible. This proves (27) and (30).

To prove (31), we observe the following about the new object stored at l. First, its type is the type of its properties, for which preservation is proved by the induction hypothesis on (29). Second, the internal property `@this` is not taken into account by Σ.

Case E-Assign

Assume: $\Sigma \models \langle H, L, \mathtt{e}_1 = \mathtt{e}_2 \rangle : \mathtt{T}$ (32)

Prove: $\Sigma' \models \langle H', v \rangle : \mathtt{T}'$ (33)

$\Sigma \subseteq \Sigma'$ (34)

$\mathtt{T}' \leqq \mathtt{T}$ (35)

From (32):

$$\frac{\Sigma \models H \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(36)}{\Gamma \vdash \mathtt{e}_1 : \mathtt{S}} \quad \dfrac{(37)}{\Gamma \vdash \mathtt{e}_2 : \mathtt{T}} \quad \dfrac{(38)}{\mathtt{T} \leqq \mathtt{S}}}{\Gamma \vdash \mathtt{e}_1 = \mathtt{e}_2 : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{e}_1 = \mathtt{e}_2 \rangle : \mathtt{T}}$$

with $\Gamma = context(\Sigma, L)$

To prove (33):

where $\langle H, L, \mathtt{e}_1 \rangle \Downarrow \langle H_1, (l, \mathtt{x}) \rangle$ (39)

$\langle H_1, L, \mathtt{e}_2 \rangle \Downarrow_v \langle H_2, v \rangle$ (40)

$$\frac{\dfrac{(41)}{\Sigma' \models H'} \qquad \dfrac{(42)}{\vdash \mathtt{l} : \mathtt{T}' \text{ or } \Sigma'(l') = \mathtt{T}'}}{\Sigma' \models \langle H', v \rangle : \mathtt{T}'}$$

From the induction hypothesis on (39), we know that $(l, \mathtt{x})$ is of type S' which is assignable to S. In TIPC, the left-hand side of an assignment is either a variable or a property access. Note that the preservation proof for identifiers and property accesses prove a stronger preservation where the types remain identical. Thus, we know that even after evaluation the type of $(l, \mathtt{x})$ remains S. The induction hypothesis on (40) gives us that v is of type $\mathtt{T}''$ which is assignable to T. Moreover, the induction hypothesis on (39) and (40) also ensures heap–heap type compatibility up until H_2. Finally, E-Assign points the reference $(l, \mathtt{x})$ to v. It does not extend the heap, thus proving (34). From the induction hypothesis, we know that its type ($\mathtt{T}''$) is assignable to T and from (38) we know that T is assignable to S, the type of the reference. This proves (41).

To prove (42), there are two cases. In case that v is a literal, the type obtained from the induction hypothesis on (40) is invariant with respect to the heap H_2 and thus H', which proves (35) and (42) for literals. In case that v is a location, Σ_2 (compatible with H_2 from (40)) will not be affected by the heap update in E-Assign (as this does not extend the heap). Therefore, the induction hypothesis on (39) and (40) suffice to prove (35) and (42).

Note on inter-property constraints The definition of assignment compatibility ensures that when $\Sigma(l) = \mathtt{I}$, the updated object in the heap still satisfies the constraints of I. This is explained in detail in Corollary 1 (cases 1b (assignment of an interface instance to an interface variable) and 2a (assignment to an interface property)) and Lemma 1 (case E-Assign).

Case E-Update

Assume: $\Sigma \models \langle H, L, \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\})\rangle : \mathtt{T}$ (43)

Prove: $\Sigma' \models \langle H', l\rangle : \mathtt{T}'$ (44)

$\Sigma \subseteq \Sigma'$ (45)

$\mathtt{T}' \leqq \mathtt{T}$ (46)

As can be seen in the typing rules I-UpdateObj and I-UpdateInf, the type of the first argument of `assign` has either an object literal type or an interface type. We first cover the case where the first argument has an object literal type.

From (43):

$$\dfrac{\dfrac{(47)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(48)}{\Gamma \vdash \mathtt{e} : \{\overline{\mathtt{M}}\}} \qquad \dfrac{(49)}{\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{N}}\}} \qquad \dfrac{(50)}{\mathtt{T} = \{\overline{\mathtt{M}}\} \Subset \{\overline{\mathtt{N}}\}}}{\Gamma \vdash \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}) : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\})\rangle : \mathtt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (44):

where $\langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H_0', l\rangle$ (51)

$H_1 = H_0' * [l_r \mapsto \mathit{clone}(H_0'(l), l_r)]$

and every $\mathtt{e}_i$ is evaluated and updated in l_r (52)

$$\frac{\dfrac{(53)}{\Sigma' \models H'} \quad \dfrac{(54)}{\Sigma'(l_r) = \mathtt{T}'}}{\Sigma' \models \langle H', l_r\rangle : \mathtt{T}'}$$

To prove (45) and (53), we start from (47). Rule E-Update extends the heap with a new object that is a clone of an existing location l on the heap. Correspondingly, Σ is extended with an extra location of which the type is identical to the type of l. Changes caused to the heap by evaluating the object literal $\mathtt{e}$ and the properties $\mathtt{n}_i$ remain consistent with a corresponding heap store, given the induction hypothesis on (52). For every change of H_i' by adding or updating each $\mathtt{n}_i$, the corresponding Σ is also updated accordingly. This leaves us with a Σ' that remains compatible with the final heap H'.

From (53), it follows that the type of l_r in H' is found in Σ'. This proves (54). Moreover, $\mathtt{T}$ is the set of properties of $\{\overline{\mathtt{M}}\}$ and $\{\overline{\mathtt{N}}\}$. To prove (46), there are three cases to cover. Properties that only reside in $\{\overline{\mathtt{M}}\}$ are covered by the induction hypothesis on (51). Similarly, properties that only reside in $\{\overline{\mathtt{N}}\}$ are covered by the induction hypothesis on (52). For properties that are common between $\{\overline{\mathtt{M}}\}$ and $\{\overline{\mathtt{N}}\}$, the heap contains the value of the second argument. The corresponding Σ' matches the exact behaviour performed by the $\Subset$ operator.

Next, we cover the case where the first argument has an interface type. From (43):

$$\frac{\dfrac{(55)}{\Sigma \models H} \quad H, L \models \diamond \quad \dfrac{(56)}{\Gamma \vdash \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\}) : \mathtt{I}}}{\Sigma \models \langle H, L, \mathtt{assign}(\mathtt{e}, \{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_m : \mathtt{e}_m\})\rangle : \mathtt{I}}$$

$$\frac{\dfrac{(57)}{\Gamma \vdash \mathtt{e} : \mathtt{I}} \quad \dfrac{(58)}{\mathtt{I}' = \mathit{slice}(...)} \quad \dfrac{(59)}{\Gamma \vdash \texttt{<I'>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}'} \quad (60)}{(56)}$$

$$\frac{\dfrac{(61)}{\overline{\mathtt{n}} \in \mathit{dom}(\mathit{properties}(\mathtt{I}))} \quad \dfrac{(62)}{\overline{\mathtt{n}} = \mathit{dom}(\mathit{properties}(\mathtt{I}'))}}{(60)}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (44):

where $\langle H_0, L, \mathtt{e}\rangle \Downarrow_v \langle H_0', l\rangle$
$H_1 = H_0' * [l_r \mapsto \mathit{clone}(H_0'(l), l_r)]$
and every $\mathtt{e}_i$ is evaluated and updated in l_r

$$\frac{\dfrac{}{\Sigma' \models H'}\,(63) \qquad \dfrac{}{\Sigma'(l_r) = \mathtt{T}'}\,(64)}{\Sigma' \models \langle H', l_r\rangle : \mathtt{T}'}$$

At first, the prove for (63) looks very similar to the prove for (53): the same changes to the heap are performed. However, caution is required because the location l has an interface $\mathtt{I}$ in Σ. As a consequence, the cloned object that resides at l_r also has type $\mathtt{I}$ in Σ' (this already proves (64), as $\mathtt{T}' = \mathtt{I}$). Therefore, to prove (63), we need to prove that the updated object at l_r still satisfies the constraints of $\mathtt{I}$. This is covered by (58) to (62), and is explained in detail in Corollary 1 (case 2b) and Lemma 1 (case E-Update).

Case E-Prop

Assume: $\Sigma \models \langle H, L, \mathtt{e.n}\rangle : \mathtt{T}$ (65)

Prove: $\Sigma' \models \langle H', (l, \mathtt{n})\rangle : \mathtt{T}'$ (66)

$\Sigma \subseteq \Sigma'$ (67)

From (65):

$$\frac{\dfrac{}{\Sigma \models H}\,(68) \qquad H, L \models \diamond \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathtt{e} : \mathtt{S}}\,(69) \qquad \dfrac{}{\mathit{lookup}(\mathtt{S}, \mathtt{n}) = \mathtt{T}}\,(70)}{\Gamma \vdash \mathtt{e.n} : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{e.n}\rangle : \mathtt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (66):

where $\langle H, L, \mathtt{e}\rangle \Downarrow_v \langle H', l\rangle$ (71)
$l \neq \mathtt{null}$

$$\frac{\dfrac{}{\Sigma' \models H'}\,\text{IH on (69)} \qquad \dfrac{}{\Sigma'(l) \vdash \mathtt{n} : \mathtt{T}}\,(72)}{\Sigma' \models \langle H', (l, \mathtt{n})\rangle : \mathtt{T}}$$

The induction hypothesis on (71) gives us (67).

(70) ensures that the property $\mathtt{n}$ is present in $\mathtt{S}$. The induction hypothesis on (71) gives us that the location l will have a type $\mathtt{S}' \leqq \mathtt{S}$ in Σ'. The assignment

compatibility ensures us that $\mathtt{S}'$ contains at least all properties of $\mathtt{S}$. Thus, $\mathtt{n}$ will be a property of $\Sigma'(l)$. Moreover, the assginability compatiblity rule is invariant for properties, ensuring that property $\mathtt{n}$ will be of type $\mathtt{T}$ in $\mathtt{S}'$. This suffices to prove (72).

Note on inter-property constraints The *lookup* function ensures that the property is certainly present or absent in an object of an interface. In combination with I-Assign, this ensures that the presence of properties cannot change in an assignment.

Case E-Prop'

Assume: $\Sigma \models \langle H, L, \mathtt{e.n} \rangle : \mathtt{T}$ (73)

Prove: $\Sigma' \models \langle H', (l, \mathtt{n}) \rangle : \mathtt{T}'$ (74)

$\Sigma \subseteq \Sigma'$ (75)

$\mathtt{T}' \leqq \mathtt{T}$ (76)

From (73):

$$\dfrac{\Sigma \models H \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(77)}{\Gamma \vdash \mathtt{e} : \mathtt{S}} \qquad \dfrac{(78)}{\mathit{lookup}(\mathtt{S}, \mathtt{n}) = \mathtt{T}}}{\Gamma \vdash \mathtt{e.n} : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{e.n} \rangle : \mathtt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (74):

where $\langle H, L, \mathtt{e} \rangle \Downarrow_v \langle H_1, \mathtt{l} \rangle$ (79)

$H_2 = H_1 * [l_{boxed} \mapsto \mathit{box}(\mathtt{l}, l_{boxed})]$

$$\dfrac{\dfrac{(80)}{\Sigma' \models H'} \qquad \dfrac{(81)}{\Sigma'(l_{boxed}) \vdash \mathtt{n} : \mathtt{T}'}}{\Sigma' \models \langle H', (l_{boxed}, \mathtt{n}) \rangle : \mathtt{T}'}$$

To prove (80) it suffices to combine the induction hypothesis on (79) with the knowledge that *box* is a built-in function that constructs a new object and returns its location.

Recall that *box* produces a new object mapping which (at a minimum) contains all methods and properties n for which (78) succeeds, with the correct type. From this immediately follows (81).

Case E-Call

Assume: $\Sigma \models \langle H, L, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \rangle : \mathtt{T}$ (82)

Prove: $\Sigma'' \models \langle H'', v\rangle : \mathtt{T}'$ (83)

$\Sigma \subseteq \Sigma''$ (84)

$\mathtt{T}' \leqq \mathtt{T}$ (85)

From (82):

$$\frac{\Sigma \models H \qquad H, L \models \diamond \qquad \dfrac{(86) \qquad \dfrac{(87)}{\mathit{context}(\Sigma, L) \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}} \qquad \dfrac{(88)}{\overline{\mathtt{T}} \leqq \overline{\mathtt{S}}}}{\mathit{context}(\Sigma, L) \vdash \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{T}}}{\Sigma \models \langle H, L, \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\rangle : \mathtt{T}}$$

$$\text{canonical forms}\ \frac{\dfrac{\dfrac{(89)}{\mathit{context}(\Sigma, L), \mathtt{this} : \mathtt{any}, \overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}} \qquad \dfrac{(90)}{\overline{\mathtt{R}} \leqq \mathtt{T}}}{\mathit{context}(\Sigma, L) \vdash \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\} : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\}}}{\mathit{context}(\Sigma, L) \vdash \mathtt{e} : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\} \quad (91)}}{(86)}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (83):

where $\langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle$ (92)

$\gamma(H_1, r) = l_1$

$H_1(l_1) = \langle \lambda\overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L_1\rangle$

$\mathit{This}(H_1, r) = l_2$

$\langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_v \langle H_2, v_1\rangle \ldots \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_v \langle H_{n+1}, v_n\rangle$ (93)

$H' = H_{n+1} * \mathit{act}(l, \overline{\mathtt{x}}, \overline{v}, l_2)$ (94)

$\langle H', l : L_1, \overline{\mathtt{s}}\rangle \Downarrow \langle H'', \mathtt{return}\ v;\rangle$

$$\frac{\dfrac{(95)}{\Sigma' \models H''} \qquad \dfrac{(96)}{\Sigma(l) = \mathtt{T}' \text{ or } \vdash \mathtt{l} : \mathtt{T}'}}{\Sigma' \models \langle H'', v\rangle : \mathtt{T}'}$$

To prove (95), we look at the changes to the heap. The change from H to H_1 ((92)) is covered by the induction hypothesis. The transformations from H_1 to H_{n+1} ((93)) are also covered by the induction hypothesis.For the adaptions from H_{n+1} to H', we take a look at the definition of *act*. From *act*, we get a new location l which maps the parameters onto their values and binds `@this`: $l \mapsto (\{\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{@this} \mapsto l_2\})$. If we add location l to the scope chain L, we get the following:

$\mathit{context}(\Sigma, l : L) \vdash \mathtt{this} : \mathtt{any}, \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathit{context}(\Sigma, L)$ (97)

This matches with the environment in (89) against which the body of the function is evaluated.

Finally, we use preservation of statements for the body of the function. The heap type corresponding to H' stems from the heap type resulted from (93), and is extended with `this` and $\overline{\mathtt{x}}$. This proves (95). From the induction hypothesis on (93) and as can be seen in (94), Σ' is a superset of Σ. Thus, we can safely use the preservation of statements as follows:

$$\Sigma' \models \langle H', l : L, \overline{\mathtt{s}} \rangle : \mathtt{T} \text{ and } \langle H', l : L, \overline{\mathtt{s}} \rangle \Downarrow \langle H'', \mathtt{return}\ v; \rangle$$

$$\text{then } \exists \Sigma'', \mathtt{T}' \text{ such that } \Sigma' \subseteq \Sigma'', \Sigma'' \models \langle H'', \mathtt{return}\ v; \rangle : \mathtt{T}' \text{ and } \mathtt{T}' \leqq \mathtt{T}$$

Note that the type of `return` v; is identical to the type of v, according to I-ReturnVal. (84), (85) and (96) follow from the preservation for statements.

Case E-CallUndef

Same as the previous case, with alternate ending for the return.

Case E-Func

Assume: $\Sigma \models \langle H, L, \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}'\ \{\overline{\mathtt{s}}\} \rangle : \mathtt{T}$ (98)

Prove: $\Sigma' \models \langle H', l \rangle : \mathtt{T}'$ (99)

$\Sigma \subseteq \Sigma'$ (100)

$\mathtt{T}' \leqq \mathtt{T}$ (101)

From (98):

$$\frac{\dfrac{(102)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \Gamma \vdash \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\} : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\}}{\Sigma \models \langle H, L, \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\ \{\overline{\mathtt{s}}\} \rangle : \{(\overline{\mathtt{x}} : \overline{\mathtt{S}}) : \mathtt{T}\}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (99):

where $H' = H * [l \mapsto \langle \lambda \overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}, L \rangle]$ (103)

$$\frac{\dfrac{(104)}{\Sigma' \models H'} \qquad \dfrac{(105)}{\Sigma'(l) = \mathtt{T}'}}{\Sigma' \models \langle H', l \rangle : \mathtt{T}'}$$

The heap type Σ' that corresponds to the heap H' is extended with a new location that maps onto the type of of the function. Thus, this proves (104) and (105).

Case E-TypeAssert

Assume: $\Sigma \models \langle H, L, \texttt{<T>e} \rangle : \texttt{T}$ (106)

Prove: $\Sigma' \models \langle H', r \rangle : \texttt{T}'$ (107)

$\Sigma \subseteq \Sigma'$ (108)

$\texttt{T}' \leqq \texttt{T}$ (109)

From (106):

$$\dfrac{\dfrac{(110)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\dfrac{(111)}{\Gamma \vdash \texttt{e} : \texttt{S}} \qquad \dfrac{(112)}{\texttt{S} \leqq \texttt{T}}}{\Gamma \vdash \texttt{<T>e} : \texttt{T}}}{\Sigma \models \langle H, L, \texttt{<T>e} \rangle : \texttt{T}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (107):

where $\langle H, L, \texttt{e} \rangle \Downarrow \langle H', r \rangle$ (113)

$$\dfrac{\dfrac{(114)}{\Sigma' \models H'} \qquad \dfrac{(115)}{\Sigma(l) = \texttt{T} \text{ or } \vdash \texttt{l} : \texttt{T} \text{ or } \Sigma(l) \vdash x : \texttt{T}}}{\Sigma' \models \langle H', r \rangle : \texttt{T}'}$$

(110) and the induction hypothesis on (113) prove (108) and (114).

From (111), (112) and the induction hypothesis, we get that r is of type $\texttt{S}' \leqq \texttt{S} \leqq \texttt{T}$. This proves (109) and (115).

Note on inter-property constraints TIPC only allows upcasts, where an expression is cast to a more general type. This is also explained in Corollary 1 (case 1c).

Case E-TypeAssertInf

Assume: $\Sigma \models \langle H, L, \texttt{<I>}\{\overline{\texttt{n}} : \overline{\texttt{e}}\} \rangle : \texttt{I}$ (116)

Prove: $\Sigma' \models \langle H', l \rangle : \texttt{I}$ (117)

$\Sigma \subseteq \Sigma'$ (118)

From (116):

$$\dfrac{\dfrac{(119)}{\Sigma \models H} \qquad H, L \models \diamond \qquad (120)}{\Sigma \models \langle H, L, \texttt{<I>}\{\overline{\texttt{n}} : \overline{\texttt{e}}\} \rangle : \texttt{I}}$$

[B] Type Preservation

$$\dfrac{\dfrac{(121)}{\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{M}}\}} \quad \ldots \quad v = c_p \cup c_{np} \quad \dfrac{(122)}{\hat{v}(\mathit{constraints}(\mathtt{I}))}}{\dfrac{\Gamma \vdash \texttt{<I>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}}{(120)}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (117):

where $\langle H, L, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}\rangle \Downarrow \langle H_1, l\rangle$ (123)

$H' = H_1[(l, \texttt{@interface}) \mapsto \mathtt{I}]]$ (124)

$$\dfrac{\dfrac{(125)}{\Sigma' \models H'} \quad \dfrac{(126)}{\Sigma'(l) = \mathtt{I}}}{\Sigma' \models \langle H', l\rangle : \mathtt{I}}$$

Changes made to the heap in (123) are covered by the induction hypothesis. Next, in (124) the location of the evaluated object literal is extended with an interface tag that points to the interface to which the interface is casted. Thus, Σ' returns the interface to which the tag points. This proves (126). Moreover, (122) ensures that the interface constraints are satisfied for the object literal: $\Sigma', H' \models l\ \mathtt{ok} \implies$ (125).

This was also explained in Corollary 1 (case 1a) and Lemma 1 (case E-TypeAssertInf). □

B.2 Type Preservation of Statements

Theorem 2 (Type Preservation for Statements)**.** If $\Sigma \models \langle H, L, \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}}$ and $\langle H, L, \overline{\mathtt{s}}\rangle \Downarrow \langle H', s\rangle$ then $\exists \Sigma', \mathtt{T}'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', s\rangle : \mathtt{T}'$ and $\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$.

Proof. By case analysis on the evaluation rules for statements. Our strategy is analogous to the proof for preservation of expressions.

Case E-EmptySeq

Assume: $\Sigma \models \langle H, L, \bullet\rangle : \bullet$ (127)

Prove: $\Sigma \models \langle H, \mathtt{;}\rangle : \bullet$ (128)

From (127):

$$\frac{\dfrac{(129)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \mathit{context}(\Sigma, L) \vdash \bullet : \bullet}{\Sigma \models \langle H, L, \bullet\rangle : \bullet}$$

To prove (128):

$$\frac{\dfrac{}{\Sigma \models H}\,(129)}{\Sigma \models \langle H, \mathtt{;}\rangle : \bullet}$$

This evaluation rule does not change the heap.

Case E-Return

Assume: $\Sigma \models \langle H, L, \mathtt{return}; \overline{\mathtt{s}}\rangle : \mathtt{void}, \overline{\mathtt{R}}$ (130)

Prove: $\Sigma \models \langle H, \mathtt{return;}\rangle : \mathtt{void}$ (131)

$\mathtt{void} \leqq \mathtt{void} \cup (\overline{\mathtt{R}})$ (132)

From (130):

$$\frac{\dfrac{(133)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{return}; \overline{\mathtt{s}} : \mathtt{void}, \overline{\mathtt{R}}}}{\Sigma \models \langle H, L, \mathtt{return}; \overline{\mathtt{s}}\rangle : \mathtt{void}, \overline{\mathtt{R}}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (131):

$$\frac{\dfrac{}{\Sigma \models H}\,(133)}{\Sigma \models \langle H, \mathtt{return;}\rangle : \mathtt{void}}$$

(132) follows from the definition of assignment compatibility for union types.

[**B**] Type Preservation

Case E-ReturnVal

Assume: $\Sigma \models \langle H, L, \texttt{return e}; \overline{\texttt{s}}\rangle : \texttt{T}, \overline{\texttt{R}}$ (134)

Prove: $\Sigma' \models \langle H', \texttt{return } v\texttt{;}\rangle : \texttt{T}'$ (135)

$\Sigma \subseteq \Sigma'$ (136)

$\texttt{T}' \leqq \texttt{T} \cup \overline{\texttt{R}}$ (137)

From (134):

$$\frac{\dfrac{(138)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\Gamma \vdash \texttt{e} : \texttt{T} \qquad \Gamma \vdash \overline{\texttt{s}} : \overline{\texttt{R}}}{\Gamma \vdash \texttt{return e}; \overline{\texttt{s}} : \texttt{T}, \overline{\texttt{R}}}}{\Sigma \models \langle H, L, \texttt{return e}; \overline{\texttt{s}}\rangle : \texttt{T}, \overline{\texttt{R}}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (135):

where $\langle H, L, \texttt{e}\rangle \Downarrow_v \langle H', v\rangle$ (139)

$$\frac{\dfrac{\text{IH on (139)}}{\Sigma' \models H'} \qquad \dfrac{\text{IH on (139)}}{\Sigma' \models \langle H, v\rangle : \texttt{T}'}}{\Sigma' \models \langle H', \texttt{return } v\texttt{;}\rangle : \texttt{T}'}$$

From preservation on expressions for (139), we get (136) and that the type for v in H' is $\texttt{T}'$ $\leqq$ T. Thus, $\texttt{T}' \leqq \texttt{T} \cup \overline{\texttt{R}}$ succeeds, proving (137).

Case E-ExpSt

Assume: $\Sigma \models \langle H, L, \texttt{e}; \overline{\texttt{s}}\rangle : \overline{\texttt{T}}$ (140)

Prove: $\Sigma' \models \langle H', s\rangle : \texttt{T}'$ (141)

$\Sigma \subseteq \Sigma'$ (142)

$\texttt{T}' \leqq \cup(\overline{\texttt{T}})$ (143)

From (140):

$$\frac{\dfrac{(144)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\Gamma \vdash \texttt{e} : \texttt{S} \qquad \Gamma \vdash \overline{\texttt{s}} : \overline{\texttt{T}}}{\Gamma \vdash \texttt{e}; \overline{\texttt{s}} : \overline{\texttt{T}} \;\; (145)}}{\Sigma \models \langle H, L, \texttt{e}; \overline{\texttt{s}}\rangle : \overline{\texttt{T}}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (141):

where $\langle H, L, \texttt{e}\rangle \Downarrow \langle H_1, r\rangle$ (146)

$\langle H_1, L, \overline{\texttt{s}}\rangle \Downarrow \langle H', s\rangle$ (147)

$$\frac{\text{IH on (146) and (147)}}{\text{if } s = \;\texttt{;}\; : \texttt{T}' = \bullet, \text{ if } s = \;\texttt{return;}\; : \texttt{T}' = \texttt{void}, \text{ else } \Sigma' \models \langle H', s\rangle : \texttt{T}'}{(148)}$$

$$\dfrac{\dfrac{\text{IH on (146) and (147)}}{\Sigma' \models H'} \qquad (148)}{\Sigma' \models \langle H', s\rangle : \mathtt{T}'}$$

The induction hypothesis on (146) and (147) suffices to prove (142). From (145) and the induction hypothesis on (147) follows that $\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$, proving (143).

Case E-IfTrue

Assume: $\Sigma \models \langle H, L, \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}}$ (149)

Prove: $\Sigma' \models \langle H', s\rangle : \mathtt{T}'$ (150)

$\Sigma \subseteq \Sigma'$ (151)

$\mathtt{T}' \leqq \cup(\overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}})$ (152)

The type of an `if` statement depends on its condition: presence tests on interface properties are treated differently. First, we cover the general case (rule I-IfGeneral in Figure 6.11).

From (149):

$$\dfrac{\dfrac{(153)}{\Sigma \models H} \qquad H, L \models \diamond \qquad \dfrac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \Gamma \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1 \qquad \Gamma \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2 \qquad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}} : \overline{\mathtt{T}_1}, \overline{\mathtt{T}_2}, \overline{\mathtt{R}}}}{\Sigma \models \langle H, L, \mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}_1}, \overline{\mathtt{T}_2}, \overline{\mathtt{R}}}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (150):

where $\langle H, L, \mathtt{e}\rangle \Downarrow \langle H_1, \mathtt{true}\rangle$ (154)

$H_2 = H_1 * [l \mapsto ()]$ (155)

$\langle H_2, l : L, \overline{\mathtt{t}}_1\rangle \Downarrow \langle H_3, s\rangle$ (156)

$\langle H_3, L, s; \overline{\mathtt{s}}\rangle \Downarrow \langle H', s_r\rangle$ (157)

$$\dfrac{\dfrac{(158)}{\text{if } s_r = \ ;\ \ : \mathtt{T}' = \bullet, \text{if } s_r = \ \mathtt{return};\ \ : \mathtt{T}' = \mathtt{void}, \text{else } \Sigma' \models \langle H', s_r\rangle : \mathtt{T}'}}{(160)}$$

$$\dfrac{\dfrac{(159)}{\Sigma' \models H'} \qquad (160)}{\Sigma' \models \langle H', s_r\rangle : \mathtt{T}'}$$

The preservation of expressions is applicable to(154) and the changes to the heap made by (156) are covered by the induction hypothesis. Note that (155) adds a new empty scope object, but this does not affect evaluation (as can be seen in the definition of σ, looking up identifiers in the heap loop through all scope objects in the scope chain). For (157), there are two cases. In the case that

s is a `return` statement, $\overline{\mathtt{s}}$ is not taken into account, which results in $H' = H_3$ and $s_r = s$. Thus, the type of s_r is the type of s: $\overline{\mathtt{T}}'_1$ (induction hypothesis on (156)). This proves (151), (152) ($\mathtt{T}'_1 \leqq \cup(\overline{\mathtt{T}}_1) \leqq \cup(\overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}})$), (159) and (158).

In the other case, s is the empty statement. As a consequence, (157) equals evaluating only $\overline{\mathtt{s}}$ ($\langle H_3, L, \overline{\mathtt{s}}\rangle \Downarrow \langle H', s_r\rangle$)), as evaluating an empty statement has no impact on the heap. We use the induction hypothesis to prove (151),(152) ($\mathtt{R}' \leqq \cup(\overline{\mathtt{R}}) \leqq \cup(\overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}})$), (159) and (158).

Now, we cover the case where the `if` statement tests the presence of an interface instance (I-IfPresenceInterface in Figure 6.11).

From (149):

$$\dfrac{\dfrac{(161)}{\Sigma \models H} \qquad H, L \models \diamond \qquad (162)}{\Sigma \models \langle H, L, \mathtt{if}\ (\mathtt{x.n} \equiv \mathtt{undefined})\ \{\overline{\mathtt{t}}_1\}\ \mathtt{else}\ \{\overline{\mathtt{t}}_2\}; \overline{\mathtt{s}}\rangle : \overline{\mathtt{T}}}$$

$$\dfrac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}} \qquad \ldots \qquad \dfrac{(163)}{\Gamma \uplus \mathtt{x} : \mathtt{I}^- \vdash \overline{\mathtt{t}}_1 : \overline{\mathtt{T}}_1} \qquad \dfrac{(164)}{\Gamma \uplus \mathtt{x} : \mathtt{I}^+ \vdash \overline{\mathtt{t}}_2 : \overline{\mathtt{T}}_2}}{(162)}$$

with $\Gamma = \mathit{context}(\Sigma, L)$

To prove (150):

where
$$\langle H, L, \mathtt{e}\rangle \Downarrow \langle H_1, \mathtt{true}\rangle \tag{165}$$
$$H_2 = H_1 * [l \mapsto ()] \tag{166}$$
$$\langle H_2, l : L, \overline{\mathtt{t}}_1\rangle \Downarrow \langle H_3, s\rangle \tag{167}$$
$$\langle H_3, L, s; \overline{\mathtt{s}}\rangle \Downarrow \langle H', s_r\rangle \tag{168}$$

$$\dfrac{\dfrac{(169)}{\Sigma' \models H'} \qquad \dfrac{(170)}{\text{if } s = \ ;\ : \mathtt{T}' = \bullet, \text{ if } s = \ \mathtt{return};\ : \mathtt{T}' = \mathtt{void}, \text{ else } \Sigma' \models \langle H', s\rangle : \mathtt{T}'}}{\Sigma' \models \langle H', s\rangle : \mathtt{T}'}$$

The reasoning about $\mathtt{e}$ and $s; \overline{\mathtt{s}}$ are the same as for I-IfGeneral. This proof differs for $\overline{\mathtt{t}}_1$ and $\overline{\mathtt{t}}_2$. (163) gives us that $\overline{\mathtt{t}}_1$ is of type $\overline{\mathtt{T}}_1$ in an extended environment, not Γ. This extension entails an update of the interface tested in the conditional of the `if` statement. When type checking the true branch, the interface contains an extra constraint to indicate the presence of the tested property.

The key insight here concerns Σ_2 (where $\Sigma_2 \models H_2$). $\Sigma_2(l_o)$ results in $\mathtt{I}$, with l_o as location for the interface instance being tested. However, the extra knowledge on the presence of properties can be added to $\Sigma_2(l_o)$ (resulting in Σ'_2) with the

guarantee that $\Sigma_2' \models H_2$. Given the extended heap type Σ_2', we can use the induction hypothesis on (167) where the type of s is $\mathtt{T}_1' \leqq \overline{\mathtt{T}}_1 \leqq \cup(\overline{\mathtt{T}}_1, \overline{\mathtt{T}}_2, \overline{\mathtt{R}})$. This is also explained in Corollary 1 (case 3).

Given this knowledge, we can use the same reasoning as for I-IfGeneral to prove (151),(152), (169) and (170).

Case E-IfFalse
Similar to E-IfTrue.

Case E-ITVarDec

Assume: $\Sigma \models \langle H, L, \mathtt{let\ x : S = e}; \overline{\mathtt{s}} \rangle : \overline{\mathtt{T}}$ (171)
Prove: $\Sigma' \models \langle H', s \rangle : \mathtt{T}'$ (172)
$\Sigma \subseteq \Sigma'$ (173)
$\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$ (174)
From (171):

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \dfrac{(175)}{\mathtt{T} \leqq \mathtt{S}} \qquad noDup(\Gamma, \mathtt{x : S}) \qquad \dfrac{(176)}{\mathtt{T} \uplus \mathtt{x : S} \vdash \overline{\mathtt{s}} : \overline{\mathtt{T}}}}{\Gamma \vdash \mathtt{let\ x : S = e}; \overline{\mathtt{s}} : \overline{\mathtt{R}}} \qquad \dfrac{\dfrac{(177)}{\Sigma \models H} \qquad H, L \models \diamond \qquad (178)}{\Sigma \models \langle H, L, \mathtt{let\ x : S = e}; \overline{\mathtt{s}} \rangle : \overline{\mathtt{R}}}}{\Sigma \models \langle H, L, \mathtt{let\ x : S = e}; \overline{\mathtt{s}} \rangle : \overline{\mathtt{R}}}$$

with $\Gamma = context(\Sigma, L)$
To prove (172):
where $\langle H, L, \mathtt{e} \rangle \Downarrow_v \langle H_1, v \rangle$ (179)
$H_2 = H_1 * [l \mapsto (\{\mathtt{x} \mapsto v\})]$ (180)
$\langle H_2, l : L, \overline{\mathtt{s}} \rangle \Downarrow \langle H_3, s \rangle$ (181)

$$\dfrac{\dfrac{(182)}{\Sigma' \models H'} \qquad \dfrac{(183)}{\text{if } s_r = \ ;\ : \mathtt{T}' = \bullet, \text{ if } s_r = \ \mathtt{return};\ : \mathtt{T}' = \mathtt{void}, \text{ else } \Sigma' \models \langle H', s_r \rangle : \mathtt{T}'}}{\Sigma' \models \langle H', s \rangle : \mathtt{T}'}$$

The type preservation of expressions covers the heap change in (179). In (180), the heap is extended with an object map that maps the variable onto its evaluated value. The corresponding heap store Σ_2 ($\Sigma_2 \models H_2$) corresponds with the environment in (176). In combination with the induction hypothesis on (181), this gives us: $\mathtt{T}' \leqq \cup(\overline{\mathtt{T}})$. This suffices to prove (173), (174), (182) and (183).

[**B**] Type Preservation

Note on inter-property constraints The assignment compatibility rule in I-ITVarDec ensures that the left-hand side of the variable declaration does not have an interface type. Therefore, E-ITVarDec does not have logic for adding `@interface` tags to certain objects.

□

Appendix C

Specification of the Twitter API

The following listing contains a snippet of the Twitter API, written in the specification language OAS-IP, introduced in Section 9.3.

```
{
    "swagger": "2.0",
    "info": {
        "version": "1.1",
        "title": "Twitter REST API"
    },
    "host": "api.twitter.com",
    "basePath": "/1.1",
    "schemes": [
        "http",
        "https"
    ],
    "consumes": [
        "application/json"
    ],
    "produces": [
        "application/json"
    ],
    "securityDefinitions": {
        "oauth": {
            "type": "oauth2",
            "flow": "implicit",
            "authorizationUrl": "https://twitter.com/oauth/authorize/?client_id=CLIENT-ID",
            "scopes": {
                "basic": "to read any and all data related to twitter\n"
            }
        }
    },
    "security": [
        {
            "oauth": [
                "basic"
            ]
        }
    ],
    "x-constraint-definitions": [
        "minimum(f, v)          := value(f) >= v",
        "exclusiveMinimum(f, v) := value(f) > v",
        "maximum(f, v)          := value(f) <= v",
        "exclusiveMaximum(f, v) := value(f) < v",
        "minLength(f, v)        := string-length(f) > v",
        "maxLength(f, v)        := string-length(f) < v",
```

```
43          "minItems(f, v)"          := "array-length(f) < v",
44          "maxItems(f, v)"          := "array-length(f) > v",
45          "enum(f, v_1, v_2)"       := "value(f) == v_1 OR value(f) == v_2",
46          "required(f)"             := "present(f)",
47          "string?(f)"              := "type(f) == String",
48          "number?(f)"              := "type(f) == Number",
49          "boolean?(f)"             := "type(f) == Boolean",
50          "xor(f1, f2)"             := "(present(f1) AND NOT(present(f2))) OR
51                                        (NOT(present(f1)) AND present(f2))",
52          "dependent(f, f_1)"       := "present(f_1) -> present(f)",
53          "group(f1, f2)"           := "(present(f1) -> present(f2)) AND
54                                        (present(f2) -> present(f1))",
55          "and(f1, f2)"             := "present(f1) AND present(f2)"
56      ],
57      "paths": {
58        "/direct_messages/new": {
59            "post": {
60                "description": "Sends a new direct message to specified user",
61                "security": [
62                    {
63                        "oauth": [
64                            "basic"
65                        ]
66                    }
67                ],
68                "parameters": [
69                    {
70                        "name": "user_id",
71                        "in": "query",
72                        "description": "User ID of user receiving message",
73                        "type": "number",
74                        "required": false
75                    },
76                    {
77                        "name": "screen_name",
78                        "in": "query",
79                        "description": "Screen name of user receiving message",
80                        "type": "string",
81                        "required": false
82                    },
83                    {
84                        "name": "text",
85                        "in": "query",
86                        "description": "Text of your direct message",
87                        "type": "string",
88                        "required": true
89                    }
90                ],
91                "x-constraints": [ "xor(user_id, screen_name)"
92                ],
93                "responses": {
94                    "200": {
95                        "description": "OK",
96                        "schema": {
97                            "$ref": "#/definitions/Messages"
98                        }
99                    }
100               }
101           }
102       },
103
104       "/statuses/update": {
105           "post": {
106               "description": "Updates the authenticating user's status",
107               "security": [
108                   {
109                       "oauth": [
110                           "basic"
111                       ]
112                   }
113               ],
114               "parameters": [
115                   {
116                       "name": "status",
117                       "in": "query",
118                       "description": "The text of your status update",
```

```
                        "required": true,
                        "type": "string"
                    },
                    {
                        "name": "in_reply_to_status_id",
                        "in": "query",
                        "description": "The ID of an existing status",
                        "required": false,
                        "type": "number"
                    },
                    {   "name": "possibly_sensitive",
                        "in": "query",
                        "description": "If you upload media that might be considered sensitive",
                        "type": "string",
                        "required": false,
                        "default": "false"
                    },
                    {
                        "name": "lat",
                        "in": "query",
                        "description": "The latitude of the location",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "long",
                        "in": "query",
                        "description": "The longitude of the location",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "place_id",
                        "in": "query",
                        "description": "A place in the world",
                        "required": false,
                        "type": "number"
                    },
                    {
                        "name": "display_coordinates",
                        "in": "query",
                        "description": "Whether or not to put a pin on the exact coordinates a tweet",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "trim_user",
                        "in": "query",
                        "description": "Set to either true, t or 1",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "media_ids",
                        "in": "query",
                        "description": "A list of media ids to associate with the tweet",
                        "required": false,
                        "type": "string"
                    }
                ],
                "x-constraints": ["group(lat,long)"],
                "responses": {
                    "200": {
                        "description": "Success",
                        "schema": {
                            "$ref": "#/definitions/Tweets"
                        }
                    },
                    "403": {
                        "description": "Error"
                    }
                }
            }
        },
        "/list/members/create": {
            "post": {
```

```
                "description": "Adds a new member to a list of the authenticated user",
                "security": [
                    {
                        "oauth": [
                            "basic"
                        ]
                    }
                ],
                "parameters": [
                    {
                        "name": "list_id",
                        "in": "query",
                        "description": "The numerical id of the list",
                        "required": false,
                        "type": "number"
                    },
                    {
                        "name": "slug",
                        "in": "query",
                        "description": "You can identify a list being requested by a slug",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "screen_name",
                        "in": "query",
                        "description": "The screen name of the user to add to the list",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "user_id",
                        "in": "query",
                        "description": "The user ID of the user to add to the list",
                        "type": "number",
                        "required": false
                    },
                    {
                        "name": "owner_screen_name",
                        "in": "query",
                        "description": "The screen name of the owner",
                        "required": false,
                        "type": "string"
                    },
                    {
                        "name": "owner_id",
                        "in": "query",
                        "description": "The user ID of the owner",
                        "required": false,
                        "type": "number"
                    }
                ],
                "x-constraints": [
                  "xor(slug, list_id)",
                  "xor(user_id, screen_name)",
                  "present(slug) -> xor(owner_screen_name, owner_id)",
                  "dependent(slug, owner_screen_name)",
                  "dependent(slug, owner_id)"
                ],
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                }
            }
        }
    },
    "definitions": {
       //omitted
    }
}
```

Listing C.1: Snippet of specification for the Twitter API

Bibliography

Martin Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag New York, 1996. ISBN 978-1-4612-6445-3. doi: 10.1007/978-1-4419-8598-9.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP 2005, pages 428–452. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31725-8.

Lennart Augustsson. Cayenne: a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250. ACM, 1998. ISBN 1-58113-024-4. doi: 10.1145/289423.289451.

SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. SAFEWAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 507–517. ACM, 2014. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635916.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):8:1–8:45, January 2011. ISSN 0164-0925. doi: 10.1145/1890028.1890031.

Gavin Bierman, MJ Parkinson, and AM Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP 2014, pages 257–281. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44202-9.

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical Optional Types for Clojure. In *Proceedings of the 25th European Symposium on Programming Languages and Systems*, ESOP 2016, pages 68–94. Springer Berlin Heidelberg, 2016. ISBN 978-3-662-49498-1.

Ana Bove and Peter Dybjer. Dependent Types at Work. In *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pages 57–99. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03153-3. doi: 10.1007/978-3-642-03153-3_2.

John Boyland, James Noble, and William Retert. Capabilities for Sharing. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP 2001, pages 2–27. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-45337-6.

Edwin C. Brady. IDRIS: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, pages 43–54. ACM, 2011. ISBN 978-1-4503-0487-0. doi: 10.1145/1929529.1929536.

Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 410–429. ACM, 2016. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984017.

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and Precise Type Checking for JavaScript. *Proceedings of the ACM Programming Languages*, 1(OOPSLA):48:1–48:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133872.

Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. SJS: A Type System for JavaScript with Fixed Object Layout. In *Proceedings of the 22nd International Static Analysis Symposium*, SAS 2015, pages 181–198. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-48288-9.

Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web Services Description Language (WSDL) 1.1, 2001.

Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606. ACM, 2012a. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384659.

Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 231–244. ACM, 2012b. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103686.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1985.

Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

Peter J Danielsen and Alan Jeffrey. Validation and Interactivity of Web API Documentation. In *Proceedings of the 2013 IEEE 20th International Conference on Web Services*, ICWS 2013, pages 523–530. IEEE Computer Society, 2013. ISBN 978-0-7695-5025-1. doi: 10.1109/ICWS.2013.76.

Nicolaas Govert De Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970. ISBN 978-3-540-36262-3.

Hamza Ed-douibi, Javier Luis Canovas Izquierdo, and Jordi Cabot. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference*, EDOC 2018, pages 181–190. IEEE, 2018. ISBN 978-1-5386-4139-2. doi: 10.1109/EDOC.2018.00031.

Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016. URL `http://arxiv.org/abs/1610.07978`.

Linus Ek, Ola Holmström, and Stevan Andjelkovic. Formalizing Arne Andersson trees and Left-leaning Red-Black trees in Agda. Technical report, Chalmers University of Technology, 2009. Bachelor thesis.

Facebook Inc. Flow, a. URL `https://flow.org`. Accessed: 2018-11-16.

Facebook Inc. Hack, b. URL `https://hacklang.org`. Accessed: 2018-11-16.

Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 302–312. ACM, 2003. ISBN 1-58113-712-5. doi: 10.1145/949305.949332.

Manuel Fahndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 337–350. ACM, 2007. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297052.

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59. ACM, 2002. ISBN 1-58113-487-8. doi: 10.1145/581478.581484.

Cormac Flanagan, Stephen N Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. *Presented at the FOOL/WOOD workshop*, 2006.

Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277. ACM, 1991. ISBN 0-89791-428-7. doi: 10.1145/113445.113468.

Jean H Gallier. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. Elements of reusable object-oriented software. Addison-Wesley, 1995. ISBN 0-201-63361-2.

Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44. ACM, 2012. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103663.

Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

GitHub. Octoverse 2018: Fastest Growing Languages. https://octoverse.github.com/projects#languages. Accessed: 2019-04-06.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 21–40. ACM, 2012. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384619.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts Made Manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364. ACM, 2010. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706341.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State using Flow Analysis. In *Proceedings of the European Symposium on Programming*, ESOP 2011, pages 256–275. Springer, 2011. ISBN 978-3-642-19718-5.

Marc J Hadley. Web Application Description Language (WADL). 2006.

Phillip Heidegger and Peter Thiemann. Recency Types for Analyzing Scripting Languages. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP 2010, pages 200–224. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14107-2.

William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

Dongseok Jang and Kwang-Moo Choe. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1930–1937. ACM, 2009. ISBN 978-1-60558-166-8. doi: 10.1145/1529282.1529711.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium*, SAS 2009, pages 238–255. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03237-0.

Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type Refinement for Static Analysis of JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 17–26. ACM, 2013. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508175.

Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based Data Structure Verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 304–315. ACM, 2009. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542510.

Andrew Kent and Sam Tobin-Hochstadt. Adding Practical Dependent Types to Typed Racket. STOP 2015 Scripts to Programs, 2015. URL `https://2015.ecoop.org/event/stop2015-adding-practical-dependent-types-to-typed-racket`.

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 296–309. ACM, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908091.

Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '08, pages 619–625. IEEE Computer Society, 2008. ISBN 978-0-7695-3496-1. doi: 10.1109/WIIAT.2008.379.

Nico Lehmann and Éric Tanter. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 775–788. ACM, 2017. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009856.

Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 1–16. ACM, 2013. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508170.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984.

Conor McBride. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming*, AFP 2004, pages 130–170. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31872-9.

Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee Protocols. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP 2014, pages 334–359. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44202-9.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained Types for Object-oriented Languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 457–474. ACM, 2008. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449800.

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter Constraints in Contemporary Web APIs. In *Proceedings of the International Conference on Web Engineering*, ICWE 2017, pages 323–335. Springer International Publishing, 2017. ISBN 978-3-319-60131-1.

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Static Typing of Complex Presence Constraints in Interfaces (Artifact). *Dagstuhl Artifacts Series*, 4(3):3:1–3:2, 2018a. ISSN 2509-8195. doi: 10.4230/DARTS.4.3.3.

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Static Typing of Complex Presence Constraints in Interfaces. In *Proceedings of the 32nd European Conference on Object-Oriented Programming*, ECOOP 2018, pages 14:1–14:27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018b. ISBN 978-3-95977-079-8. doi: 10.4230/LIPIcs.ECOOP.2018.14.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Exploring New Frontiers of Theoretical Informatics*, pages 437–450. Springer US, 2004. ISBN 978-1-4020-8141-5.

S. Owre, J. Rushby, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24:709–720, September 1998. ISSN 0098-5589. doi: 10.1109/32.713327.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212. ACM, 2008. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390656.

Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002.

Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and types for objects with first-class member names. In *Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages*, FOOL 2012, page 37, 2012.

Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Typed-based Verification of Web Sandboxes. *Journal of Computer Security*, 22(4):511–565, 2014.

Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180. ACM, 2015. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676971.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, ECOOP 2015, pages 76–100, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.76.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169. ACM, 2008. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602.

Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Report on the Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP 2007, pages 2–27. Springer, 2007. ISBN 978-3-540-73589-2.

Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *Proceedings of the European Symposium on Programming*, ESOP 2000, pages 366–381. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-46425-9.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034811.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270. ACM, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837655.

Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *Proceedings of the European Symposium on Programming*, ESOP 2005, pages 408–422. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31987-0.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406. ACM, 2008. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328486.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128. ACM, 2010. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863561.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282. ACM, 2014. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628161.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 310–325. ACM, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908110.

Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257. ACM, 1998. ISBN 0-89791-987-4. doi: 10.1145/277650.277732.

Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227. ACM, 1999. ISBN 1-58113-095-3. doi: 10.1145/292540.292560.

Tian Zhao. Type inference for scripting languages with implicit extension. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages*, FOOL 2010, 2010.