Static typing of complex presence constraints in interfaces

Nathalie Oostvogels[1]
Vrije Universiteit Brussel, Brussels, Belgium
noostvog@vub.ac.be

Joeri De Koster
Vrije Universiteit Brussel, Brussels, Belgium
jdekoste@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel, Brussels, Belgium
wdmeuter@vub.ac.be

Abstract

Many functions in libraries and APIs have the notion of optional parameters, which can be mapped onto optional properties of an object representing those parameters. The fact that properties are optional opens up the possibility for APIs and libraries to design a complex "dependency logic" between properties: for example, some properties may be mutually exclusive, some properties may depend on others, etc. Existing type systems are not strong enough to express such dependency logic, which can lead to the creation of invalid objects and accidental usage of absent properties. In this paper we propose TypeScript$_{\text{IPC}}$: a variant of TypeScript with a novel type system that enables programmers to express complex presence constraints on properties. We prove that it is sound with respect to enforcing complex dependency logic defined by the programmer when an object is created, modified or accessed.

2012 ACM Subject Classification Software and its engineering → Object oriented languages, Theory of computation → Type theory, Software and its engineering → Data types and structures

Keywords and phrases type checking, interfaces, dependency logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.14

1 Introduction

Static type checking enables the compile-time detection of type errors in programs, which would otherwise occur at run-time. To enable static type checking, developers have to include *type declarations* in their code. These type declarations also serve as documentation, which facilitates reasoning over code. Early type systems only describe the basic type of the values that could be stored in a variable, but throughout the years more complex types have been introduced, such as intersection types [26], union types, linear types [16] and dependent types [22]. Using these more expressive types, developers can express more sophisticated programs while retaining the compile-time guarantee that their code is correct.

Dynamically typed languages have given rise to new challenges in type systems, such as flow-sensitivity and optional types. One such challenge in particular is using the absence

---

[1] Funded by a PhD Fellowship of the Research Foundation - Flanders (FWO)

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 14; pp. 14:1–14:27
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

or presence of parameters to encode information. For example, a search function might require that at least one filter is specified, or objects might only be considered valid if a group of properties are all present or all absent. For singular properties, optional types can already express this. However, in order to fully resolve this challenge using static type systems, these *inter-property constraints* must be made explicit.

These types of constraints are common for Web APIs [24], where the presence of a 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. However, inter-property constraints also exist in programming languages and libraries. We show several examples of inter-property constraints, classified into three categories:

- **Exclusive constraints:** exactly one of a set of properties must be present. In the Twitter API, users can be identified by either their `user_id` or their `screen_name`. Another example is found in the Python standard library, where the function `os.utime`[2] 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`*"*.
- **Dependent constraints:** constraints on a property depend on the presence or the value of another property. For example, properties explaining details of a picture (name, description) should not be present if the picture property itself is not present either. In Chart.js, a library for designing charts in JavaScript, the documentation for `lines` in a chart states that *"If the* `steppedLine` *value is set to anything other than* `false`*,* `lineTension` *will be ignored"*.[3]
- **Group constraints:** a group of properties should either all be present or not present in an object. For example, latitude and longitude properties of a GPS location should always occur (or be omitted) together.

We will use a running example from the Twitter API specification to demonstrate that state-of-the-art interfaces do not suffice to describe inter-property constraints. Table 1 shows the specification for sending a private message, with a typical translation to a TypeScript interface in Listing 1. Every object that contains the input data for sending a private message should adhere to the `PrivateMessage` interface.

Property name	Optional?	Description
`text`	required	The text of your direct message.
`user_id`	optional	ID of the user who should receive the direct message.
`screen_name`	optional	Screen name of the user who should receive the direct message.
Note: One of `user_id` or `screen_name` are required.[4]		

Table 1 Twitter API documentation for sending private messages[5]

The accompanying note in Table 1 indicates that there is an *exclusive* constraint imposed on the user properties. However, in TypeScript (and also in other languages) it is impossible to express that *exactly one* of `user_id` and `screen_name` is required. The question marks

---

[2] `https://docs.python.org/3/library/os.html#os.utime`

[3] `http://www.chartjs.org/docs/latest/charts/line.html#stepped-line`

[4] At the time of writing, the note below the table was explicitly mentioned in the API. Recently, the description has changed — omitting the note — but the constraint still holds.

[5] `https://developer.twitter.com/en/docs/direct-messages/sending-and-receiving/api-reference/new-message`

after `user_id` and `screen_name` in Listing 1 denote that these properties are *optional*, but this means that the type system accepts objects containing none or both of the user properties. Similarly, a group constraint with latitude and longitude properties 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.

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

Listing 1 TypeScript interface for the specification in Table 1

The lack of support for inter-property constraints in existing programming languages causes errors to be delegated to the runtime. In the best case, the API or library provides a detailed error message, stating which properties were incompatible. Sometimes no error message is returned at all, and a silent choice is made instead: if both user properties are provided, Twitter silently chooses the screen name over the user ID.

Existing type systems are incapable of expressing inter-property constraints and statically checking these constraints both at construction time and during updates. In this paper we describe a type system that can express such complex presence constraints over multiple properties of an object. We show how interfaces with support for inter-property constraints can be incorporated in programming languages in Section 2, and describe the key features of the type system in Section 3. Sections 4 and 5 present the formalisations of the language, as a variant of TypeScript. We prove that the type system enforces both type safety and constraint integrity (Section 6). Sections 7 and 8 discuss related work and future work, respectively. Section 9 contains concluding remarks.

2 Programming with Inter-property Constraints

In this section, we propose a syntax for expressing inter-property constraints and explain intuitively how they can be used. Unless otherwise noted, every code snippet in the rest of this paper is written in TypeScript$_{\text{IPC}}$, our version of TypeScript with support for inter-property constraints. The syntax of TypeScript$_{\text{IPC}}$ differs little from the syntax of TypeScript. Instead, the type system makes optimal use of the information provided by the program about the structure of objects.

2.1 Definition of interfaces with constraints

To handle inter-property constraints, the interface declaration syntax needs to be extended. Listing 2 shows an example of an interface declaration, revisiting the Twitter specification we showed in Table 1. Interfaces now consist of two parts: next to the traditional property name–type declarations, they also contain a list of constraints over the presence and absence of those properties. The syntax of constraints is as follows:

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

As opposed to TypeScript and many other languages — where properties are required by default and can be made optional with a `?` annotation — properties in TypeScript$_{\text{IPC}}$ are optional by default and are made required by adding a `present(n)` constraint.

Lines 2–4 list the three properties for `PrivateMessage`, and their types in TypeScript$_{\text{IPC}}$. Lines 6 and 7 denote the constraints on the presence of those three properties. To improve the expressiveness of interfaces, constraints on the presence of a property can be combined with logical operators. The `PrivateMessage` interface lists two presence constraints: line 6 requires the presence of the `text` property and line 7 is the inter-property constraint from our running example. Objects can only be of an interface type if all its constraints are satisfied.

```
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 2 Twitter private messaging API data expressed as interface with constraints

The constraint definition language does not list optional properties as an explicit constraint operation, as this can be expressed by the following constraint: $\texttt{present}(n) \lor \neg\texttt{present}(n)$, which is a tautology.

Listing 3 shows another example of inter-property constraints, describing an interface of a picture object with required caption (line 7) and optional geolocation. However, the `lat` and `long` properties are dependent on the `picture` property: if the picture itself is not provided, the location should 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. The fourth constraint on line 10 requires that the latitude and longitude properties are present or absent *together*.

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

Listing 3 Interface with dependent and group inter-property constraints

Interfaces with inter-property constraints can also benefit from interface inheritance. For example, let us consider the case where we want a stricter version of the `PrivateMessage` interface in which only the screen name is allowed. Instead of creating a new interface, the existing interface can also be extended with extra constraints. Listing 4 shows an interface in which all properties and constraints of `PrivateMessage` are inherited, with an additional `present(screen_name)` constraint. As the `xor` constraint from `PrivateMessage` is still applicable, this interface implicitly forbids the presence of a `user_id` property.

```
interface PrivateMessageStrict extends PrivateMessage {
  // reuse properties from PrivateMessage
} constraining {
  present(screen_name);
}
```

Listing 4 Extending PrivateMessage to require the screen name property

2.2 Object creation

Listing 5 shows how three objects are created and assigned to three variables of type `PrivateMessage`. Even though the interface contains inter-property constraints, nothing changes 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, including the exclusive constraint: only `user_id` is passed along as identification for the user. However, the type system has to generate errors for `msg2` and `msg3`, as they both violate the exclusive constraint.

```
var msg1: PrivateMessage = {text: "Hello", user_id: 42}; // correct
var msg2: PrivateMessage = {text: "Hello"}; // error: none present
var msg3: PrivateMessage = {text: "Hello",
                            user_id: 42,
                            screen_name: "Alice"}; // error: both present
```

Listing 5 Creating objects with inter-property constraints

The type system also needs to ensure that no constraints are violated when expressions with different interface types are assigned to each other, or when an instance of an interface is assigned to a variable with a regular object literal type.

2.3 Property access

When inter-property constraints are involved, reading object properties requires extra caution. The type system should only allow the access of a property when that property is guaranteed to be present. For example, 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`.

By contrast, the type system should reject programs where other properties of a `PrivateMessage` object are accessed. The exclusive constraint guarantees that exactly one of `user_id` and `screen_name` will be present, but it is not known *which* property actually is. The function `getUserId` (defined in Listing 6) tries to read the `user_id` of a `PrivateMessage`, which generates a type error as this property access is unsafe.

To prevent errors from accessing undefined properties, programmers must verify whether properties are present before using them. For example, the function `getUser` first performs a test to check whether `user_id` is present. Inside the true branch, access to the user ID (line 6) must be allowed. Additionally, because there is an inter-property constraint between `user_id` and `screen_name`, the `screen_name` property is guaranteed to be absent even though we did not explicitly test for it. The inverse holds in the false branch.

Similarly, in the function `getLocation` (which retrieves the longitude and latitude of a picture), the type system has to allow the access of `long`, which follows directly from the if statement. On top of that, the type system should also accept accessing the properties `lat` and `picture`, which are both guaranteed to be present if `long` is present.

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

```
  }
}
function getLocation(picture: Picture) {
  if (picture.long !== undefined) {
    picture.long;     // :: number  (present due to if statement
    picture.lat;      // :: number  (present due to group constraint)
    picture.picture;  // :: string  (present due to dependent constraint)
  }
}
```

Listing 6 Accessing properties

2.4 Property updates

As with every object-oriented type system, the assignment of a new value to a property of an object should only succeed when the value is of the correct type. Inter-property constraints add an extra complication: assigning to a property might invalidate an inter-property constraint.

Updating a property that was already guaranteed to be present is safe: the previous section showed that the type system will only assign the intended type to properties that are known to be present. Line 2 in Listing 7 illustrates this with the `text` property. The update of the `user_id` property on line 4 will fail, however: the type system disallows the property access, as explained in the previous section.

Note that it is not allowed to assign the value `undefined` to properties of any type except `Undefined`, as this would make a required property absent (line 3). This principle is known as the *strict null-checking* mode of TypeScript. In Listing 7, it is only allowed to assign `undefined` to `screen_name` (line 8), as this property is known to be absent inside the consequent of the if statement.

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

Listing 7 Updating properties

The examples of Listing 7 only modify one property at a time. However, an inter-property constraint often requires the modification of several properties at once, as the object could be in a type-incorrect state inbetween several assignments. Let us consider the case in Listing 8 where a programmer wants to switch from user ID to screen name. The type system rejects this program, as it breaks the rules imposed by the strict-null checking mode. This behaviour is desirable: inbetween lines 3 and 4, the inter-property constraint of `msg` is violated: it contains neither user ID nor screen name.

```
1  var 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 8 Changing an inter-property constraint is not possible with separate assignments

Our solution is to enable updating of multiple properties simultaneously, such that the object is never in an invalid state between consecutive assignment statements. We propose an `assign`(i, o) operator[6] that returns a *copy* of object i, in which the properties from the object o are added or updated. Listing 9 shows how the `assign` operator switches from `user_id` to `screen_name`. Note that `assign` is functional: instead of modifying its first arguments, it returns a new object.

```
var msg:  PrivateMessage = {text: "Hello", user_id: 42};
var msg2: PrivateMessage =
      assign(msg, {user_id: undefined, screen_name: "Alice"}); // correct
var msg3: PrivateMessage =
      assign(msg, {user_id: undefined}); // incorrect
```

Listing 9 Using multi-assign to switch from user ID to screen name

While programmers can update any subset of the properties of an object, not all combinations are correct, as the `msg3` example above shows. Intuitively, if an inter-property constraint exists between two or more properties, they should all appear together in the call to `assign`. The properties of an object can thus be divided into one or more "clusters". For example a `Picture` object has a trivial cluster for `caption`, and a separate cluster for the `long`, `lat` and `picture` properties.

3 Verifying Constraints in TypeScript

The addition of constraints to interfaces has consequences on several facets of the type system. In the following sections, we explain how the type system of TypeScript$_{\text{IPC}}$ deals with the creation, modification, and access of properties of interfaces with constraints. Because the constraint language expresses constraints with logical connectives, the type system uses several concepts from propositional logic to guarantee correctness.

3.1 Object literals have to satisfy constraints

The type system only accepts the assignment of an object literal to a variable with an interface type when that object satisfies the interface constraints. Using terminology from propositional logic, the type system requires that the object literal is a *valuation* [15] that satisfies the logical formulas of the interface (constraints). More specifically, an object literal defines a valuation, assigning truth values (presence and absence of properties) to proposition symbols (property names). Moreover, for every valuation v there exists a unique function $\hat{v}$ which takes a proposition (here: the constraints) and returns true or false.

3.2 Constraints dictate property presence

As with other type systems, interface declarations contain a list of properties with their types. However, looking up a property of an interface may only succeed when the interface contains a constraint indicating that property is present. Of course, with complex inter-property constraints, these constraints may not be *directly* present in the constraint set. Instead, the type system relies on *logical entailment* (denoted $\vDash_\ell$) to verify whether a `present`(n) constraint follows from a set of constraints. Calculating logical entailments

---

[6] `assign` resembles the `Object.assign` function in JavaScript, but does not modify its input object.

can be efficiently automated using deductive systems such as the Gentzen system [15]. Returning to the `PrivateMessage` example, the type system verifies the following logical entailment for accessing the `text` property:

$$\{\texttt{present(text);present(user\_id) xor present(screen\_name)}\} \vDash_\ell \texttt{present(text)}$$

Similarly, inter-property constraints can also guarantee the *absence* of a property. In the case where neither the presence or 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 `getUserId` of Listing 6, because neither the presence nor the absence of `user_id` is a logical consequence of the interface constraints:

$$\{\texttt{present(text);present(user\_id) xor present(screen\_name)}\} \nvDash_\ell \texttt{present(user\_id)}$$

$$\{\texttt{present(text);present(user\_id) xor present(screen\_name)}\} \nvDash_\ell \neg\texttt{present(user\_id)}$$

3.3 Explicit property presence tests

In dynamic languages, it is common to perform runtime property presence tests. These presence tests can provide the type system with more information about the object being tested: in one branch it is certain that the property is present, while it is guaranteed to be absent in the other. For the `true` branch in the function `getUser` of Listing 6, the type system simply adds the new information ($\texttt{present(user\_id)}$) to the set of constraints, to allow the access of the `user_id` property.

That extra information can trigger other inter-property constraints, thus guaranteeing the presence or absence of other properties. Using logical entailment, the type system can prove that `screen_name` will not be present:

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

Similarly, the presence check on longitude in `getLocation` guarantees that the longitude is present, but also suffices to safely access latitude (by combining the constraint $\texttt{present(long)} \leftrightarrow \texttt{present(lat)}$ with $\texttt{present(long)}$) and the picture itself (combining constraints $\texttt{present(long)} \rightarrow \texttt{present(picture)}$ and $\texttt{present(long)}$).

3.4 Interface–interface compatibility

Normally, an instance of interface $\texttt{I}_0$ is considered assignable to a variable with as type another interface $\texttt{I}_1$ if $\texttt{I}_0$ contains at least every property and method in the other interface. However, with the addition of constraints we must also take care that no instance of $\texttt{I}_0$ violates the constraints in $\texttt{I}_1$. To guarantee that all constraints of $\texttt{I}_1$ are satisfied, every constraint from $\texttt{I}_1$ must be a *logical entailment* of the constraints in $\texttt{I}_0$. Properties which are absent from $\texttt{I}_0$ result in extra $\neg\texttt{present(n)}$ constraints at the left-hand side of the entailment.

For example, assigning a variable with a more strict interface type `PrivateMessage2` (defined in Figure 1) to a variable of type `PrivateMessage`, gives rise to the following logical entailment. Next to the constraints of `PrivateMessage`, the left side of the logical entailment

```
interface PrivateMessage1 {
  text: string;
  user_id: number;
  screen_name: string;
} constraining {
  present(text);
  present(user_id);
  present(screen_name);
}
```

```
interface PrivateMessage2 {
  text: string;
  user_id: number;

} constraining {
  present(text);
  present(user_id);

}
```

Figure 1 Other versions of the PrivateMessage interface

contains an extra constraint due the absence of the screen name in `PrivateMessage2`. Without the third constraint, the logical entailment would not be valid.

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

As for properties, one might expect that $\texttt{I}_0$ may contain a superset of the properties in $\texttt{I}_1$. However, this can lead to constraint violations: consider the following example, with two variations on the `PrivateMessage` interface (defined in Figure 1).

```
1 var msg1: PrivateMessage1 = {text:"Hello",user_id:42,screen_name:"Alice"};
2 var msg2: PrivateMessage2 = msg1;
3 var msg3: PrivateMessage  = msg2;
```

On line 2, a variable of type `PrivateMessage1` is assigned to a variable of type `PrivateMessage2` and line 3 assigns a variable of type `PrivateMessage2` to a variable of the default `PrivateMessage` interface: both assignments would be allowed, as no constraints are violated. However, line 3 would result in an object of type `PrivateMessage` that contains both `user_id` and `screen_name`, violating its constraints.

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 interface $\texttt{I}_0$ only contains properties other than those in $\texttt{I}_1$ when those properties are guaranteed to be absent. This is not the case for the second assignment (line 2) in the example:

$$\{\texttt{present(text); present(user\_id); present(screen\_name)}\} \not\models_\ell \neg\texttt{present}(\texttt{screen\_name})$$

3.5 Updated objects have to satisfy constraints

To verify that all constraints are still satisfied after a simultaneous update of multiple properties, the type system again uses valuations. However, as the update only affects a subset of the properties, the object literal in the second argument only serves as a valuation for a subset of the constraints.

Consider the following example of an interface that indicates both the sender (with the `s_*` properties) and the receiver (`r_*`). Logically, these properties form separate clusters that are not affected by each other.

```
 1  interface PrivateMessage3 {
 2    text: string;
 3    r_user_id: number;
 4    r_screen_name: string;
 5    s_user_id: number
 6    s_screen_name: string;
 7  } constraining {
 8    present(text);
 9    r_user_id xor r_screen_name;
10    s_user_id xor s_screen_name;
11  }
```

```
var msg:PrivateMessage3 =
                  {text: "Hello",
                   r_user_id: 42,
                   s_user_id: 43};


var msg2 = assign(msg,
         {r_user_id: undefined,
          r_screen_name: "Alice"});
```

The `assign` at the right side 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 is a valid valuation of the constraint on line 9. This is the case, as `undefined` is interpreted as an absent property. Of course, 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 cluster are updated.

4 TypeScript$_{\mathrm{IPC}}$: A Variant of TypeScript with Constraints

Section 2 showed how constraints on the presence of properties can be added to TypeScript's interfaces and Section 3 gave an informal idea of how the type system statically enforces that constraints stay satisfied throughout the program. In this section, we formalise these ideas in TypeScript$_{\mathrm{IPC}}$, a variant of TypeScript.

TypeScript is an extension of JavaScript which adds optional static typing. It provides extra features over JavaScript such as structural typing and named interfaces. To ensure compatibility with existing JavaScript code, type annotations in TypeScript are optional which enables developers to gradually convert existing JavaScript code to TypeScript.

This section introduces TypeScript$_{\mathrm{IPC}}$. The syntax, semantics and type rules presented in this section build upon those presented by Bierman et al. [7]. They present the type system in two parts: the first is a safe calculus (called safeFTS) which contains the core features of TypeScript, including structural typing, contextual types and the lack of block scoping in JavaScript. The second part expands safeFTS to a production-ready calculus, which is unsafe.

TypeScript$_{\mathrm{IPC}}$ reuses most of safeFTS's features, which are based upon TypeScript 0.9.5. However, as checking the presence or absence of properties is a key feature of TypeScript$_{\mathrm{IPC}}$, we use the subtyping rules from the strict null checking mode in TypeScript 2.0. These make it illegal to assign `null` and `undefined` to variables of any other type, unless explicitly allowed.

Our variant of TypeScript with constraints will focus on objects and interfaces. Contextual typing and constructs to deal with the lack of block scoping are omitted for clarity. As they are orthogonal to object creation and interfaces, they can be trivially added to the language presented in this paper.

4.1 Syntax

Figure 2 presents the syntax of TypeScript$_{\mathrm{IPC}}$, which is based on the syntax presented in [7]. It features basic language expressions such as identifiers, literals, assignment and binary operators. Literals can be numbers n, strings s, or one of the following constants:

$\mathtt{e},\mathtt{f} \in$ Expressions	::=	$\mathtt{x}$	(Identifier)
		$\mathtt{l}$	(Literal)
		$\{\overline{\mathtt{a}}\}$	(Object literal)
		$\mathtt{e} = \mathtt{f}$	(Assignment operator)
		$\mathtt{assign}(\mathtt{e}, \{\overline{\mathtt{a}}\})$	(Assign operator)
		$\mathtt{e} \otimes \mathtt{f}$	(Binary operator)
		$\mathtt{e.n}$	(Property access)
		$\mathtt{e}(\overline{\mathtt{f}})$	(Function call)
		$\mathtt{<T>e}$	(Type assertion)
		$\mathtt{function}\ (\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{S}\ \{\overline{\mathtt{s}}\}$	(Function expression)
$\mathtt{a} \in$ Property assignments	::=	$\mathtt{n} : \mathtt{e}$	(Property assignment)
$\mathtt{s},\mathtt{t} \in$ Statements	::=	$\mathtt{e;}$	(Expression statement)
		$\mathtt{if}\ (\mathtt{e})\ \{\overline{\mathtt{s}}\}\ \mathtt{else}\ \{\overline{\mathtt{t}}\}$	(If statement)
		$\mathtt{return;}$	(Return statement)
		$\mathtt{return\ e;}$	(Return value statement)
		$\mathtt{var}\ \mathtt{x}:\mathtt{T} = \mathtt{e}$	(Variable declaration)

Figure 2 Syntax of TypeScript$_{\mathrm{IPC}}$

`true`, `false`, `null` and `undefined`, where `null` indicates the empty object and `undefined` is returned when accessing a property that is not present in an object.

Objects are defined using object literals, which map property names to values. Multiple properties of an object can be updated at once using `assign`. This function returns a new object that contains all properties of the first argument. Properties from the second argument are either updated (when already present in the first argument) or added (otherwise). Function expressions are similar to those in JavaScript, but with type annotations for the parameters. Expressions can be cast to a type, but only when the cast is known to be correct. Statements and variable declarations are straightforward. TypeScript$_{\mathrm{IPC}}$ only features variable declarations where the type and the value for the variable are provided.

The empty sequence is denoted with $\bullet$, a concatenation is denoted using a comma, and a sequence of expressions is written as $\overline{\mathtt{e}}$. A sequence of property assignments $\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}$ is an abbreviation for $\{\mathtt{n}_1 : \mathtt{e}_1, \ldots, \mathtt{n}_n : \mathtt{e}_n\}$, with n the length of the sequence. Similarly, $(\overline{\mathtt{x}} : \overline{\mathtt{T}})$ is a sequence of function arguments $(\mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n)$.

To reduce the size and complexity of our formalisation, we omit parts of safeFTS that do not contribute to the necessary adaptations for inter-property constraints. More specifically, TypeScript$_{\mathrm{IPC}}$ does not support computed property accesses, untyped identifiers, call signatures without parameter types or return types, and untyped and uninitialised variable declarations.

Figure 3 shows that TypeScript$_{\mathrm{IPC}}$ has three kinds of types: the top type `any`, primitive types and object types. An object type is represented by either a literal type or an interface type. Note that functions are represented as callable objects that contain one field with its type of the form $(\overline{\mathtt{x}} : \overline{\mathtt{S}})\!:\!\mathtt{T}$. A sequence of types is denoted as $\overline{\mathtt{T}}$, and the sequence of properties and call signatures is analogous to their corresponding value sequences.

Interfaces play a key role in expressing inter-property constraints, and their declaration

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

Figure 3 Types of TypeScript$_{\text{IPC}}$

in TypeScript$_{\text{IPC}}$ is different from other languages:

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

TypeScript$_{\text{IPC}}$ interfaces first list the property (field or method) names, together with their types as usual. However, constraints on the presence of a property are specified in the `constraining` section, using the syntax presented in Section 2.1. By default, all properties are optional unless marked as `present`. In addition, the `constraining` section can impose inter-property constraints on properties of the interface. Interfaces can inherit properties and constraints from other interfaces. TypeScript$_{\text{IPC}}$ does not allow interfaces to define properties with the same name as any of their superinterfaces. Furthermore, all properties are public.

To retrieve the properties and constraints from a given interface, we define two auxiliary functions *properties* and *constraints*. Analogous to the inheritance of properties, constraints from the superinterfaces are simply accumulated.

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

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

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

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

Before analysis starts, all interface declarations are gathered and stored in a mapping Σ_i of interface names `I` to their respective declaration `D`. As in safeFTS, a program is a pair

$(\Sigma_i, \overline{\mathtt{s}})$ containing an interface table and a sequence of statements. TypeScript$_{\text{IPC}}$ requires every interface to satisfy a set of sanity conditions:

1. For every $\mathtt{I} \in \text{dom}(\Sigma_i)$, $\Sigma_i(\mathtt{I}) = \mathtt{interface\ I}\ \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}\ \mathtt{constraining}\ \{\overline{\mathtt{c}}\}$ or $\Sigma_i(\mathtt{I}) = \mathtt{interface\ I\ extends}\ \overline{\mathtt{I}}\ \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}\ \mathtt{constraining}\ \{\overline{\mathtt{c}}\}$;
2. for every interface name `I` appearing anywhere in Σ_i, it is the case that $\mathtt{I} \in \text{dom}(\Sigma_i)$;
3. there are no cycles in the dependency graph induced by the `extends` clauses of the interface declarations defined in Σ_i;
4. for every interface name `I` in $\text{dom}(\Sigma_i)$, there exists at least one valuation (that assigns truth values (indicating presence or absence) to proposition symbols (property names)) that satisfies the constraints ($\mathit{constraints}(\mathtt{I})$);
5. for every interface name `I` in $\text{dom}(\Sigma_i)$, none of the properties of `I` is allowed to be of type `any` or `Undefined`.

The first three sanity conditions are common, and almost identical to those in safeFTS, the latter two are specifically for interfaces with inter-property constraints. The fourth condition prevents the declaration of interfaces with inherent contradictions, and the fifth condition prevents the assignment of `undefined` to an object property, which — at runtime — is equal to an absent property.

4.2 Type System

In this section we present the type system of TypeScript$_{\text{IPC}}$. Figure 4 shows the type rules of TypeScript$_{\text{IPC}}$, which are based on those of safeFTS. For clarity, we omit contextual typing and JavaScript's lack of block scoping from the typing rules, which are orthogonal extensions to the contribution in this paper. The typing judgement is written as follows: $\Gamma \vdash \mathtt{e} : \mathtt{T}$, where given an environment Γ the expression `e` is of type `T`. Γ maps variables to types $(\overline{\mathtt{x}} : \overline{\mathtt{T}})$ and is extended as follows: $\Gamma, \mathtt{x} : \mathtt{T}$. For sequences, we write $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ as shorthand for $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_\mathtt{n} : \mathtt{T}_\mathtt{n}$, with n the length of the sequence. $\overline{\mathtt{S}} \leqq \mathtt{T}$ 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 $\mathtt{S}_1 \leqq \mathtt{T}_1, \ldots, \mathtt{S}_\mathtt{n} \leqq \mathtt{T}_\mathtt{n}$.

The rules that do not (directly) deal with interfaces are standard: I-Id looks up a variable in the environment. I-Number, I-String, I-Bool, I-Null and I-Undefined all type check a constant. The type of an object literal is a mapping of all property names onto the type of their expression (I-ObLit). In I-Op, the type system checks that the parameters have the expected type.

4.2.1 Property lookup

I-Prop first retrieves the type of the object, and then determines the type of the property using the *lookup* function:

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

$$\text{I-Id}\ \frac{}{\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x}:\mathtt{T}} \qquad \text{I-Number}\ \frac{}{\Gamma \vdash n : \mathtt{number}} \qquad \text{I-String}\ \frac{}{\Gamma \vdash s : \mathtt{string}}$$

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

$$\text{I-Undefined}\ \frac{}{\Gamma \vdash \mathtt{undefined} : \mathtt{Undefined}} \qquad \text{I-ObLit}\ \frac{\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\Gamma \vdash \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \{\overline{\mathtt{n}} : \overline{\mathtt{T}}\}}$$

$$\text{I-Op}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S}_0 \qquad \Gamma \vdash \mathtt{f} : \mathtt{S}_1 \qquad \mathtt{S}_0 \otimes \mathtt{S}_1 = \mathtt{T}}{\Gamma \vdash \mathtt{e} \otimes \mathtt{f} : \mathtt{T}} \qquad \text{I-Prop}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{S} \\ \mathit{lookup}(\mathtt{S}, \mathtt{n}) = \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{e.n} : \mathtt{T}}$$

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

$$\text{I-Func}\ \frac{\Gamma, \mathtt{this} : \mathtt{any}, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}} \qquad \overline{\mathtt{R}} \leqq \mathtt{S}}{\Gamma \vdash \mathtt{function}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{S}\ \{\overline{\mathtt{s}}\} : \{(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{S}\}} \qquad \text{I-Assert}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{S} \\ \mathtt{S} \leqq \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{<T>e} : \mathtt{T}}$$

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

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

$$\text{I-UpdateInf}\ \frac{\begin{array}{ccc}\Gamma \vdash \mathtt{e} : \mathtt{I} & \multicolumn{2}{c}{\mathtt{I}' = \mathit{slice}(\mathtt{I}, \overline{\mathtt{n}}, \mathit{constraints}(\mathtt{I}))} \\ \Gamma \vdash \mathtt{<I'>}\{\overline{\mathtt{n}} : \overline{\mathtt{e}}\} : \mathtt{I}' & \overline{\mathtt{n}} \in \mathit{dom}(\mathit{properties}(\mathtt{I})) & \overline{\mathtt{n}} = \mathit{dom}(\mathit{properties}(\mathtt{I}'))\end{array}}{\Gamma \vdash \mathtt{assign}(\mathtt{e}, \{\overline{\mathtt{n}} : \overline{\mathtt{e}}\}) : \mathtt{I}}$$

Figure 4 Type rules of TypeScript$_{\text{IPC}}$

Properties of primitive types are looked up in their associated interface type (lines 1–3). Looking up a property in an object literal type is as expected (line 4). When the property is not found in the object literal type, the *lookup* function searches the property in the Object type (line 5). The last two lines show how a property is looked up in a TypeScript$_{\text{IPC}}$ interface. Simply looking up the property in the list of interface properties does not suffice: as shown in Section 3.2, the *constraints* on an interface type dictate the presence of its properties. If the property is guaranteed to be present, *lookup* returns its type, otherwise it returns `Undefined`. If neither the presence nor the absence of a property can be guaranteed, the *lookup* function is not defined.

4.2.2 Assignment Compatibility

In I-Assign, a new expression may only be assigned to an expression when the new expression has a type that is *assignable to* the type of the original expression. Similarly, I-Call uses the assignment compatibility relationship to check that the parameters of the function call have the correct type. When type checking a function definition, I-Func extends the environment as usual with the type declarations for the parameters, and type `any` for the `this` variable. The return types of the function body must all be assignable to the declared return type. As only safe casts are allowed in TypeScript$_{\text{IPC}}$, casting an expression to another type is only allowed when the original type is assignable to the cast type (I-Assert).

The assignment compatibility relation is defined in Figure 5, and is based on the rules of safeFTS. 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 TypeScript$_{\text{IPC}}$, 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 TypeScript$_{\text{IPC}}$ must be checked against the interface definition instead of via a coinductive relation.

First, assignment compatibility is transitive (A-Trans) and reflexive (A-Refl). Any type can be assigned to `any` (A-AnyR). `null` can only be assigned to itself or `any`, and `undefined` can only be assigned to itself, `any` or `void` (A-Undefined). For assigning primitive types, A-Prim looks up their interface type. An object literal type can be assigned to another object literal type when all the properties of the source object are also present on the target object, and properties are assignable pairwise (A-Object). A-Prop defines that assigning properties to each other is invariant. Assigning call signatures is contra-/co-variant (A-CS and A-CS-Void). A-Interface is as discussed in Section 3.4: interfaces must be at least as strict as the target interface to be considered assignment-compatible, and common properties should have the same type. Extra properties on $\mathtt{I}_0$ are not allowed, unless their absence can be proven from the contraints. A-IntObj allows assigning an interface to an object when the constraints on the interface guarantee that all properties are present.

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). However, width subtyping conflicts with inter-property constraints, that may require properties to be absent: the assignment of an object to an interface could possibly invalidate the interface constraints at runtime. Therefore, there is no assignment compatibility rule for assigning an object to an interface: TypeScript$_{\text{IPC}}$ only allows the casting of a *literal* object to an interface. This is covered by the rule I-AssertInf (covered in Section 4.2.3). By only allowing object literals (instead of all object literal types), the type system has an exact view of the properties that are present and can thus guarantee that the interface constraints are satisfied.

A small study[7] on web APIs indicates that this 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.

Note that, as a consequence, the examples in Section 2 that create objects with inter-property constraints (Listing 5) are only accepted by the type checker if they are first typecast to `PrivateMessage`.

---

[7] `http://soft.vub.ac.be/~noostvog/typescriptipc/olrestriction.pdf`

$$\text{A-Trans}\,\frac{\mathtt{R} \leqq \mathtt{S} \qquad \mathtt{S} \leqq \mathtt{T}}{\mathtt{R} \leqq \mathtt{T}} \qquad \text{A-Refl}\,\frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{S}} \qquad \text{A-AnyR}\,\frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{any}}$$

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

$$\text{A-Object}\,\frac{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \vdash \diamond \qquad \overline{\mathtt{M}}_1 \leqq \overline{\mathtt{M}}_2}{\{\overline{\mathtt{M}}_0, \overline{\mathtt{M}}_1\} \leqq \{\overline{\mathtt{M}}_2\}} \qquad \text{A-Prop}\,\frac{}{\mathtt{n} : \mathtt{T} \leqq \mathtt{n} : \mathtt{T}}$$

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

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

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

Figure 5 Assignment compatibility for types in TypeScript$_{\text{IPC}}$

4.2.3 Creating and updating

The rule I-AssertInf covers the case where an object literal is cast to an interface. As explained in Section 3.1, the cast only succeeds when the properties of the object have the correct type *and* the presence and absence of properties form a valid valuation of the constraints. A property is considered absent when it is not in the object literal, or when its type is `Undefined`.

I-UpdateInf and I-UpdateObj cover updating multiple properties of an object at once, using the functional `assign` function (see Section 3.5). When the type of the first argument of `assign` is an object literal type, I-UpdateObj simply combines (updates or adds, when the property is already present resp. not present in the first argument) the properties of the second argument with the first, using $\uplus$. More caution is required when the type of `e` is an interface, as updating properties could invalidate the constraints. 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. To solve this, I-UpdateInf uses the *slice* function (defined below) to generate an interface that only contains constraints concerning the properties that are being updated. Given this generated interface, rule I-AssertInf is reused to verify whether the updated properties satisfy the applicable subset of constraints. An `assign` fails if any of the updated properties are not declared in the interface `I`, or when not all properties of `I'` are part of the second argument of `assign`.

To preserve soundness, `assign` does not 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 [10] and RSC [34]), or requires tracking aliases (such as in DJS [11]).

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. It uses an auxiliary function fv which takes a constraint and returns all referenced properties.

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

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

4.2.4 Sequence typing

Finally, Figure 6 shows the type 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 $\overline{\mathtt{R}}$. These return types are collected from all `return` statements in the sequence. This is used by the type system to verify whether the types of all return statements in a function are assignable to the declared return type.

All rules are default and identical to those in safeFTS, except for the type rules for `if` statements. As with latent predicates in occurrence typing [33], the type system uses the presence tests inside conditions of `if` statements to refine interface types in the branches. I-IfPresenceInterface shows the case where the condition contains a property presence test (cfr. Section 3.3) for a property of an object with an interface type.

The function *addConstraint* adds the constraints to the interface, and performs a satisfiability check to verify that there are no inconsistent constraints in the extended constraint set. 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. The definition of *addConstraint* is straightforward and omitted for lack of space. Note that the type assignment for `e` is *overwritten* in both branches using $\uplus$, leaving type assignments for other variables as-is. Although Figure 6 only defines rules for a single pattern of conditional expressions, the type rule can be generalised to inequalities and combined logical expressions, like in [33]. If statements without presence tests are covered by I-IfGeneral.

5 Operational Semantics of TypeScript$_{\text{IPC}}$

TypeScript is a superset of JavaScript that adds typing. However, after compilation, TypeScript emits JavaScript code in which all types are erased, which means that the semantics of TypeScript (and TypeScript$_{\text{IPC}}$) are the same of those of JavaScript. However, we provide the operational semantics of TypeScript$_{\text{IPC}}$, which will be used in Section 6 to prove its soundness.

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 $\mathtt{function}(\overline{\mathtt{x}})\{\overline{\mathtt{s}}\}$ is shortened to $\lambda\overline{\mathtt{x}}.\{\overline{\mathtt{s}}\}$) and a scope chain L. 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 element x. $H(l,x)\downarrow$ is true

$$\text{I-EmpSeq}\,\frac{}{\Gamma \vdash \bullet : \bullet} \qquad\qquad \text{I-ExpSt}\,\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \Gamma \vdash \overline{\mathtt{s}} : \overline{\mathtt{R}}}{\Gamma \vdash \mathtt{e};\overline{\mathtt{s}} : \overline{\mathtt{R}}}$$

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

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

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

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

Figure 6 Sequence type rules in TypeScript$_{\text{IPC}}$

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 may not be in the domain of the heap.

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. A program is evaluated with a scope chain containing only the global JavaScript object l_g. 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 variable lookup function σ is defined as follows:

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

The evaluation of an expression `e` is written as follows: $\langle H_1, L, \mathtt{e}\rangle \Downarrow \langle H_2, r\rangle$, 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 $\langle H_1, L, \mathtt{e}\rangle \Downarrow_v \langle H_2, v\rangle$ as the combination $\langle H_1, L, \mathtt{e}\rangle \Downarrow \langle H_2, r\rangle$ and $\gamma(H_2, r) = v$.

Figure 7 shows the semantics for evaluating expressions in TypeScript$_{\text{IPC}}$. The evaluation rules of TypeScript$_{\text{IPC}}$ are almost identical to those in safeFTS, but omit block scoping. E-Oblit uses an auxiliary function *new* to create a new location in the object map, E-Update uses the auxiliary function *clone* to duplicate an object, and E-Prop′ uses the auxiliary function *box* to box primitive values. Note that we do not create bindings for all local variables up front: they are added to the local scope as they are declared and initialised. E-Update and E-TypeAssertInf are new. E-Update evaluates the functional update of multiple properties at once, and E-TypeAssertInf covers the casting of an object literal to an interface. Next to evaluating the object literal (as in E-ObLit), the internal property `@interface` 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

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

$$\text{E-this}\,\frac{\begin{array}{c}\sigma(H, L, \texttt{@this}) = l_1 \\ H(l_1, \texttt{@this}) = l\end{array}}{\langle H, L, \texttt{this}\rangle \Downarrow \langle H, l\rangle} \qquad\qquad \text{E-Op}\,\frac{\begin{array}{c}\langle H_0, L, \texttt{e}_1\rangle \Downarrow_v \langle H_1, \texttt{l}_1\rangle \\ \langle H_1, L, \texttt{e}_2\rangle \Downarrow_v \langle H_2, \texttt{l}_2\rangle\end{array}}{\langle H_0, L, \texttt{e}_1 \otimes \texttt{e}_2\rangle \Downarrow \langle H_2, \texttt{l}_1 \otimes \texttt{l}_2\rangle}$$

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

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

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

$$\text{E-Prop}\,\frac{\begin{array}{c}\langle H_0, L, \texttt{e}\rangle \Downarrow_v \langle H_1, l\rangle \\ l \neq \texttt{null}\end{array}}{\langle H_0, L, \texttt{e.n}\rangle \Downarrow \langle H_1, (l, \texttt{n})\rangle} \qquad\qquad \text{E-Prop}'\,\frac{\begin{array}{c}\langle H_0, L, \texttt{e}\rangle \Downarrow_v \langle H_1, \texttt{l}\rangle \\ H_2 = H_1 * [l \mapsto \mathit{box}(\texttt{l})]\end{array}}{\langle H_0, L, \texttt{e.n}\rangle \Downarrow \langle H_2, (l, \texttt{n})\rangle}$$

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

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

$$\text{E-Func}\,\frac{H_1 = H_0 * [l \mapsto \langle \lambda \overline{\texttt{x}}.\{\overline{\texttt{s}}\}, L\rangle]}{\langle H_0, L, \texttt{function}(\overline{\texttt{x}})\{\overline{\texttt{s}}\}\rangle \Downarrow \langle H_1, l\rangle} \qquad\qquad \text{E-TypeAssert}\,\frac{\langle H_0, L, \texttt{e}\rangle \Downarrow \langle H_1, r_1\rangle}{\langle H_0, L, \texttt{<T>e}\rangle \Downarrow \langle H_1, r_1\rangle}$$

$$\text{E-TypeAssertInf}\,\frac{\begin{array}{c}H_1 = H_0 * [l \mapsto \{\texttt{@interface} \mapsto \texttt{I}\}] \\ \langle H_1, L, \texttt{e}_1\rangle \Downarrow_v \langle H_1', v_1\rangle \qquad H_2 = H_1'[(l, \texttt{n}_1) \mapsto v_1] \\ \ldots \\ \langle H_m, L, \texttt{e}_m\rangle \Downarrow_v \langle H_m', v_m\rangle \qquad H = H_m'[(l, \texttt{n}_m) \mapsto v_m]\end{array}}{\langle H_0, L, \texttt{<I>}\{\overline{\texttt{n}} : \overline{\texttt{e}}\}\rangle \Downarrow \langle H, l\rangle}$$

Figure 7 Operational semantics of $\text{TypeScript}_{\text{IPC}}$

program. In E-Call, the auxiliary functions This and *act* are used:

$$\mathrm{This}(H,(l,x)) = \begin{cases} l & \text{if } H(l,\texttt{@this})\downarrow \\ l_g & \text{otherwise} \end{cases}$$

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

The evaluation relation for statement sequences is written as $\langle H_1, L, \overline{\texttt{s}}_1\rangle \Downarrow \langle H_2, s\rangle$, where s is a statement result (i.e. either `return;`, `return` v`;` or `;`). These rules are omitted for brevity. Unlike safeFTS, the branches of if statements introduce a new scope, so variables declared there are not visible outside.

6 Soundness

The novelty of the TypeScript$_{\mathrm{IPC}}$ type system 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. This property is captured in Lemma 1.

Our proof of type soundness is structured identically to [7], albeit without support for block typing and contextual typing. We define a heap type Σ as a partial function from heap locations to types [3, 8] (either function types, object literal types, or interface types). Next, we introduce a number of judgments. First, we define a well-formedness judgment for heaps $H \models \diamond$ and a judgment that a heap H and scope chain L are compatible, written $H, L \models \diamond$. This judgment requires that all scope objects in the scope chain exist on the heap. We use a judgment $\Sigma \models H$ to denote that the heap H is compatible with the heap type Σ. This compatibility also requires that the constraints of interface types are satisfied, which we prove in Lemma 2. Finally, we depend on a function $context(\Sigma, L)$ which 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 instance is returned. Because E-TypeAssertInf attaches an `@interface` label to all interface variables in the heap, Σ can reconstruct interface types as well as function types and object literal types.

$$\begin{aligned} context(\Sigma, []) &= \{\} \\ context(\Sigma, l : L) &= context(\Sigma, L) \uplus \Sigma(l) \end{aligned}$$

We combine the judgments above to write $\Sigma \models \langle H, L, \texttt{e}\rangle : \texttt{T}$ to mean $\Sigma \models H$; $H, L \models \diamond$; and $context(\Sigma, L) \vdash \texttt{e} : \texttt{T}$. We define an analogous judgment for statements, as $\Sigma \models \langle H, L, \overline{\texttt{s}}\rangle : \overline{\texttt{T}}$. Finally, we add a judgment on the result of evaluation of expressions, written $\Sigma \models \langle H, r\rangle : \texttt{T}$.

Before we can prove the safety properties of our type system with respect to evaluation, we first show that the constraints of an interface type accurately predict the presence or absence of its properties at runtime.

▶ **Lemma 1** (Constraint–presence correlation)**.** *The type system of TypeScript*$_{\mathrm{IPC}}$ *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* `not(present(n))`*, it is certain that the property* `n` *will not be present.*

Proof. 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 $\mathtt{I}_0$ is assigned to a variable of type interface $\mathtt{I}_1$, the type system requires that the constraints are satisfied via the assignment compatibility rule A-Interface. The correlation holds for the source object (with type $\mathtt{I}_0$) and the compatibility rule asserts that the properties of $\mathtt{I}_1$ 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 the type system and thus the correlation. ◀

From Lemma 1, we can prove that a well-typed program does not violate constraints at runtime. We add an additional condition to the heap–heap type compatibility rule stated above as $\Sigma \models H$: (the *fields* function returns field names of an object at runtime)

► **Lemma 2** (Correctness of interface types at runtime)**.** *For heap locations tagged as interface types, i.e. those where $\Sigma(l) = \mathtt{I}$, the following is required:*

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

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

This lemma is not only unaffected by explicit property presence tests, it guarantees it because of property 3. Assuming an object (with interface type `I`) is well-formed before the presence test, then the strengthened interface type `I′` in the taken branch must more closely resemble the state of the runtime object.

Proof. By induction on the evaluation rules. Most rules do not directly modify the heap, so we only focus on the rules that potentially invalidate this condition.
E-TypeAssertInf This evaluation rule is responsible for instantiating interface types on the heap, given an object literal. Property 1 follows from the evaluation rule. Properties 2 and 3 follow directly from the type system.
E-Assign There are three sub-cases: $\mathtt{e}_1$ 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 $\mathtt{e}_2$.
- In case of a property belonging to an object: the three properties 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, we prevent users from modifying the `@interface` property (preserving property 1). Properties 2 and 3 are guaranteed by assignment compatibilty.

E-Update This rule first clones the source object (for which all properties are already satisfied) before assigning the new fields. Property 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′`, the properties 2 and 3 are guaranteed by the I-UpdateInf rule. For fields *not* in `I′`, properties 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, but cannot invalidate existing interface types on the heap.

E-Prop', E-Func These rules create a heap location for respectively properties of literal objects and a closure, but neither can affect existing interface types 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 types on the heap. ◀

Finally, we can combine Lemma 2 with the existing proof of safeFTS to obtain proof of type safety in the presence of constraints.

▶ **Theorem 3** (Subject reduction)**.**

- *If* $\Sigma \models \langle H, L, e\rangle : T$ *and* $\langle H, L, e\rangle \Downarrow \langle H', r\rangle$
  then $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', r\rangle : T'$ *and* $T' \leqq T$.
- *If* $\Sigma \models \langle H, L, \overline{s}\rangle : \overline{T}$ *and* $\langle H, L, \overline{s}\rangle \Downarrow \langle H', s\rangle$
  then $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', s\rangle : T'$ *and* $T' \leqq \texttt{return}(\overline{T})$.

7 Related Work

To the best of our knowledge, TypeScript$_{\text{IPC}}$ is the first language that statically verifies *all* aspects of programming with inter-property constraints: defining, initialising, accessing *and* updating objects with inter-property constraints. In this section, we give an overview of existing work related to various aspects of the type system presented in this paper.

Dependent and refinement types

Dependently typed languages [5, 36] allow programmers to write more expressive types, by parametrising types on values. There are no restrictions on what dependent types can express, which comes at the cost of decidability. Refinement types are a restricted form of dependent types where types are "refined" with predicates that are statically decidable, for example through SMT solvers. Refinement types have been used to verify many different properties [35, 14, 29, 23, 6, 11, 34]. We limit our discussion of refinement types to the applications that are close to our work: refinement types for dynamic programming languages and object-oriented programming languages.

DJS [11] extends a subset of JavaScript with dependent types, which allows (with some modifications) the expression of inter-property constraints over object properties. However, DJS requires extensive knowledge on heap typing from the developer. This significant annotation overhead is acknowledged in the paper. Contrast this to TypeScript$_{\text{IPC}}$, which proposes a lightweight extension to regular TypeScript interfaces.

In [34], Vekris et al. introduce RSC, a lightweight refinement system for TypeScript. RSC allows invariants to be imposed in classes and objects, including inter-property constraints on properties. However, the soundness of these invariants is guaranteed by restricting invariants to be imposed on *immutable* properties. Flanagan et al. introduce Hoop [13], a hybrid object-oriented programming language with refinement types and object invariants. Hoop requires refinements and variants to be pure and therefore refinements can only be placed on immutable data. In [23], Nystrom et al. introduce a form of dependent types for objects in X10. Again, constraints can only be imposed on immutable fields. To conclude, although refinement type systems are often able to express inter-property constraints, none of them support inter-property constraints after the initialisation phase: updating properties that are part of inter-property constraints is impossible. In contrast, TypeScript$_{\text{IPC}}$ allows single-property updates of objects, *and* guarantees that the constraints remain satisfied.

Type refinements

The type system of TypeScript$_{\text{IPC}}$ extracts property presence information from conditional expressions. This concept is known as occurrence typing [32, 33] or type refinement, which narrows (or *strengthens*) variable types based on predicates in conditional expressions. Several static type systems for dynamic languages such as TypeScript [2], Hack [1], Flow [10], λ_S [17] and [20] support refining types using tests on the type of a value. Recently, a hybrid occurrence-refinement type system was proposed in [21]. As this paper demonstrates, occurrence typing can also be applied to objects with inter-property constraints.

Constraint-based programming

The constraint-centric interfaces introduced in this paper should not be confused with constraint-based programming [30]. Constraint-based programming is a discipline that finds solutions for a number of variables given constraints over these variables. By contrast, TypeScript$_{\text{IPC}}$ uses constraints and flow information to determine the most specific presence information for properties of objects.

Type systems for dynamic languages

In recent years, several formalisations for TypeScript have been proposed. As already mentioned earlier, TypeScript$_{\text{IPC}}$ is based on earlier work [7] by Bierman et al., who formalised both sound and unsound features of TypeScript, including features such as contextual typing and the lack of block scoping in JavaScript. There exist several other approaches for adding gradual typing to dynamic languages such as TypeScript [27, 28] and Dart [19]. These approaches focus on improving the combination between sound and unsound parts of type systems for dynamic languages, which is orthogonal to the goal of our paper: enabling programmers to express inter-property constraints and statically enforcing them.

There already exist several research efforts that focus on the dynamic nature of objects in JavaScript [4, 31, 18, 9], providing a static type system that verifies the usage of objects, such as property additions, accesses and updates. The focus of this paper is not on supporting JavaScript's object types, but on extending object types with inter-property constraints. Accessing and updating object properties with inter-property constraints is allowed, but only when it does not invalidate the object constraints.

Optional object properties

TypeScript$_{\text{IPC}}$ is not the first language to impose constraints on the presence of an object property. In TypeScript, objects (and methods) can contain optional properties (and parameters). In strict null checking mode, the type of an optional property in TypeScript is automatically transformed to a union type, combining the original type with `Undefined`. Similarly, programmers can only assign `null` to value types in C# if that type is indicated as a nullable type. To support the notion of required and optional properties in Java, there also exist Java frameworks that provide support for `@NonNull` annotations (such as

[12, 25]). However, all of these languages and frameworks are restricted to single-property constraints (types and presence) and cannot express inter-property constraints.

8 Future Work

This paper introduces the concept of constraints in programming languages. Going forward, we would like to further expand the expressiveness of constraint-centric interfaces. So far, TypeScript$_{\text{IPC}}$ only supports inter-property constraints on the presence of properties. In the future, we plan to add support for *value-dependent constraints*, where the presence of a property depends on the value of another property. The introduction already listed an example of a value-dependent constraint in the Chart.js library: "*If the* `steppedLine` *value is set to anything other than* `false`, `lineTension` *will be ignored*". Another example can be found in the Google Maps API for rendering directions[8], where "*the* `infoWindow` *property is ignored when the property* `suppressInfoWindows` *is set to* `true`". To enable value-dependent constraints, we plan on using TypeScript's *literal types* that limit types to a set of predefined values.

In this paper we only considered constraints as applied to interfaces, but constraints could also be imposed on the parameters of a function definition. Listing 10 shows the (simplified) function `utime` from the Python standard library, which imposes a NAND constraint on two of its parameters.

```
function utime(path: string, times: array, ns: array) {
  //...
} constraining {
  present(path);
  ¬(present(ns) ∧ present(times));
}
```

Listing 10 Hypothetical example of a function with inter-parameter constraints

Finally, this paper highlighted the need for updating multiple properties at once. In the future, we plan on updating multiple object properties in place without increasing the annotation burden, by means of alias tracking or stronger heap types.

9 Conclusion

This paper shows how complex constraints on the presence of interface properties can be statically enforced in programming languages. We introduced a type system with *constraint-centric interfaces*, which express constraints on the presence of properties in the desired pattern.

To achieve this, the type system is extended with four new features: 1) Interfaces carry constraints on their properties; 2) The type system uses if statements to enrich variable types of interfaces used in the condition with extra information about property presence; 3) Accessing and updating a property of an object is only allowed when the constraints can statically guarantee its presence; 4) Finally, a novel procedure `assign` allows the (functional) updating of multiple properties at once, which is necessary to safely update properties that are part of an inter-property constraint.

Implementation The implementation of TypeScript$_{\text{IPC}}$ is available at `https://github.com/noostvog/TypeScriptIPC`.

---

[8] `https://developers.google.com/maps/documentation/javascript/reference/3/directions`

References

1 Hack. `http://hacklang.org/`.

2 TypeScript 2.0 Release Notes. `https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html`.

3 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.

4 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.

5 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, New York, NY, USA, 1998. ACM.

6 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, 2011.

7 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

8 Gavin M 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.

9 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*, pages 410–429. ACM, 2016.

10 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):48, 2017.

11 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, New York, NY, USA, 2012. ACM.

12 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, New York, NY, USA, 2003. ACM.

13 Cormac Flanagan, Stephen N Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. *FOOL/WOOD*, 6, 2006.

14 Tim Freeman and Frank Pfenning. Refinement Types for ML. In *In Proceedings of the SIGPLAN'91 Symposium on Language Design and Implementation*. Citeseer, 1991.

15 Jean H Gallier. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

16 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

17 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, pages 256–275. Springer, 2011.

18 Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *European conference on Object-oriented programming*, pages 200–224. Springer, 2010.

19 Thomas S Heinze, Anders Møller, and Fabio Strocco. Type safety analysis for Dart. In *Proceedings of the 12th Symposium on Dynamic Languages*, pages 1–12. ACM, 2016.

20 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, New York, NY, USA, 2013. ACM.

21 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, New York, NY, USA, 2016. ACM.

22 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984.

23 Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *In OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. Citeseer, 2008.

24 Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter Constraints in Contemporary Web APIs. In *International Conference on Web Engineering*, pages 323–335. Springer, 2017.

25 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*, pages 201–212. ACM, 2008.

26 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.

27 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, New York, NY, USA, 2015. ACM.

28 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

29 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, New York, NY, USA, 2008. ACM.

30 Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

31 Peter Thiemann. Towards a type system for analyzing javascript programs. In *European Symposium On Programming*, pages 408–422. Springer, 2005.

32 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, New York, NY, USA, 2008. ACM.

33 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, New York, NY, USA, 2010. ACM.

34 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, New York, NY, USA, 2016. ACM.

35 Hongwei Xi and Frank Pfennig. Eliminating Array Bound Checking Through Dependent Types. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

36 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.