Implementing a Typed Javascript and its IDE: a case-study with Xsemantics

Lorenzo Bettini

Dip. Statistica, Informatica, Applicazioni
Università di Firenze, Italy
Email: lorenzo.bettini@unifi.it

Jens von Pilgrim, Mark-Oliver Reiser

NumberFour AG,
Berlin, Germany
Email: {jens.von.pilgrim, mark-oliver.reiser}@numberfour.eu

Abstract—**Developing a compiler and an IDE for a programming language is time consuming and it poses several challenges, even when using language workbenches like Xtext that provides Eclipse integration. A complex type system with powerful type inference mechanisms needs to be implemented efficiently, otherwise its implementation will undermine the effective usability of the IDE: the editor must be responsive even when type inference takes place in the background, otherwise the programmer will experience too many lags. In this paper, we will present a real-world case study: N4JS, a JavaScript dialect with a full-featured Java-like static type system, including generics, and present some evaluation results. We will describe the implementation of its type system and we will focus on techniques to make the type system implementation of N4JS integrate efficiently with Eclipse. For the implementation of the type system of N4JS we use Xsemantics, a DSL for writing type systems, reduction rules and in general relation rules for languages implemented in Xtext. Xsemantics is intended for developers who are familiar with formal type systems and operational semantics since it uses a syntax that resembles rules in a formal setting. This way, the implementation of formally defined type rules can be implemented easier and more directly in Xsemantics than in Java.**

Keywords–DSL; Type System; Implementation; Eclipse.

I. Introduction

In this paper, we present N4JS, a JavaScript dialect implemented with Xtext, with powerful type inference mechanisms (including Java-like generics). In particular, we focus on the implementation of its type system. The type system of N4JS is implemented in Xsemantics, an Xtext DSL to implement type systems and reduction rules for DSLs implemented in Xtext.

The type system of N4JS drove the evolution of Xsemantics: N4JS' complex type inference system and the fact that it has to be used in production with large code bases forced us to enhance Xsemantics in many parts. The implementation of the type system of N4JS focuses both on the performance of the type system and on its integration in the Eclipse IDE.

This paper is the extended version of the conference paper [1]. With respect to the conference version, in this paper we describe more features of Xsemantics, we provide a full description of the main features of N4JS and we describe its type system implementation in more details. Motivations, related work and conclusions have been extended and enhanced accordingly.

The paper is structured as follows. In Section II we introduce the context of our work and we motivate it; we also discuss some related work. We provide a small introduction to Xtext in Section III and we show the main features of Xsemantics in Section IV. In Section V, we present N4JS and its main features. In Section VI, we describe the implementation of the type system of N4JS with Xsemantics, with some performance benchmarks related to the type system. Section VII concludes the paper.

II. Motivations and Related Work

Integrated Development Environments (IDEs) help programmers a lot with features like syntax aware editor, compiler and debugger integration, build automation and code completion, just to mention a few. In an agile [2] and test-driven context [3] the features of an IDE like Eclipse become essential and they dramatically increase productivity.

Developing a compiler and an IDE for a language is usually time consuming, even when relying on a framework like Eclipse. Implementing the parser, the model for the Abstract Syntax Tree (AST), the validation of the model (e.g., type checking), and connecting all the language features to the IDE components require lot of manual programming. Xtext, `http://www.eclipse.org/Xtext`, [4], [5] is a popular Eclipse framework for the development of programming languages and Domain-Specific Languages (DSLs), which eases all these tasks.

A language with a static type system usually features better IDE support. Given an expression and its static type, the editor can provide all the completions that make sense in that program context. For example, in a Java-like method invocation expression, the editor should propose only the methods and fields that are part of the class hierarchy of the receiver expression, and thus, it needs to know the static type of the receiver expression. The same holds for other typical IDE features, like, for example, navigation to declaration and quickfixes.

The type system and the interpreter for a language implemented in Xtext are usually implemented in Java. While this works for languages with a simple type system, it becomes a problem for an advanced type system. Since the latter is often formalized, a DSL enabling the implementation of a type system similar to the formalization would be useful. This would reduce the gap between the formalization of a language and its actual implementation.

Besides functional aspects, implementing a complex type system with powerful type inference mechanisms poses several challenges due to performance issues. Modern IDEs and compilers have defined a high standard for performance of compilation and responsiveness of typical user interactions, such as content assist and immediate error reporting. At the same time, modern statically-typed languages tend to reduce the verbosity of the syntax with respect to types by implementing type inference systems that relieve the programmer from the burden of declaring types when these can be inferred from the context. In order to be able to cope with these high demands on both type inference and performance, efficiently implemented type systems are required.

In [6] Xsemantics, `http://xsemantics.sf.net`, was introduced. Xsemantics is a DSL for writing rules for languages implemented in Xtext, in particular, the *static semantics* (type system), the *dynamic semantics* (operational semantics) and relation rules (subtyping). Given the type system specification, Xsemantics generates Java code that can be used in the Xtext implementation. Xsemantics specifications have a declarative flavor that resembles formal systems (see, e.g., [7], [8]), while keeping the Java-like shape. This makes it usable both by formal theory people and by Java programmers.

Originally, Xsemantics was focused on easy implementation of prototype languages. While the basic principles of Xsemantics were not changed, Xsemantics has been improved a lot in order to make it usable for modern full-featured languages and real-world performance requirements [9]. In that respect, N4JS drove the evolution of Xsemantics. In fact, N4JS' complex type inference system and its usage in production with large code bases forced us to enhance Xsemantics in many parts. The most relevant enhanced parts in Xsemantics dictated by N4JS can be summarized as follows:

- Enhanced handling of the rule environment, simplifying implementation of type systems with generics.
- Fields and imports, simplifying the use of Java utility class libraries from within an Xsemantics system definition.
- The capability of extending an existing Xsemantics system definition, improving the modularization of large systems.
- Improved error reporting customization, in order to provide the user with more information about errors.
- Automatic caching of results of rule computations, increasing performance.

Xsemantics itself is implemented in Xtext, thus it is completely integrated with Eclipse and its tooling. From Xsemantics we can access any existing Java library, and we can even debug Xsemantics code. It is not mandatory to implement the whole type system in Xsemantics: we can still implement parts of the type system directly in Java, in case some tasks are easier to implement in Java. In an existing language implementation, this also allows for an easy incremental or partial transition to Xsemantics. All these features have been used in the implementation of the type system of N4JS.

A. Related work

In this section we discuss some related work concerning both language workbenches and frameworks for specifying type systems.

Xsemantics can be considered the successor of Xtypes [10]. With this respect, Xsemantics provides a much richer syntax for rules that can access any existing Java library. This implies that, while with Xtypes many type computations could not be expressed, this does not happen in Xsemantics. Moreover, Xtypes targets type systems only, while Xsemantics deals with any kind of rules.

XTS [11] (Xtext Type System) is a DSL for specifying type systems for DSLs built with Xtext. The main difference with respect to Xsemantics is that XTS aims at expression based languages, not at general purpose languages. Indeed, it is not straightforward to write the type system for a Java-like language in XTS. Type systems specifications are less verbose in XTS, since it targets type systems only, but XTS does not allow introducing new relations as Xsemantics, and it does not target reductions rules. Xsemantics aims at being similar to standard type inference and semantics rules so that anyone familiar with formalization of languages can easily read a type system specification in Xsemantics.

OCL (Object Constraint Language) [12], [13] allows the developer to specify constraints in metamodels. While OCL is an expression language, Xsemantics is based on rules. Although OCL is suitable for specifying constraints, it might be hard to use to implement type inference.

Neverlang [14] is based on the fact that programming language features can be plugged and unplugged, e.g., you can "plug" exceptions, switch statements or any other linguistic constructs into a language. It also supports composition of specific Java constructs [15]. Similarly, JastAdd [16] supports modular specifications of extensible compiler tools and languages. Eco [17], [18] is a language composition editor for defining composed languages and edit programs of such composed languages. The Spoofax [19] language workbench provides support for language extensions and embeddings. Polyglot [20] is a compiler front end for Java aiming at Java language extensions. However, it does not provide any IDE support for the implemented extension. Xtext only provides single inheritance mechanisms for grammars, so different grammars can be composed only linearly. In Xsemantics a system can extend an existing one (adding and overriding rules). These extensibility and compositionality features are not as powerful as the ones of the systems mentioned above, but we think they should be enough for implementing *pluggable type systems* [21].

There are other tools for implementing DSLs and IDE tooling (we refer to [22], [23], [24] for a wider comparison). Tools like IMP (The IDE Meta-Tooling Platform) [25] and DLTK (Dynamic Languages Toolkit), `http://www.eclipse.org/dltk`, only deal with IDE features. TCS (Textual Concrete Syntax) [26] aims at providing the same mechanisms as Xtext. However, with Xtext it is easier to describe the abstract and concrete syntax at once. Morever, Xtext is completely open to customization of every part of the generated IDE. EMFText [27] is similar to Xtext. Instead of deriving a metamodel from the grammar, the language to be implemented must be defined in an abstract way using an EMF metamodel.

The Spoofax [19], language workbench mentioned above, relies on Stratego [28] for defining rule-based specifications for the type system. In [29], Spoofax is extended with a collection of declarative meta-languages to support all the aspects of language implementation including verification infrastructure and interpreters. These meta-languages include NaBL [30] for name binding and scope rules, TS for the type system and DynSem [31] for the operational semantics. Xsemantics shares with these systems the goal of reducing the gap between the formalization and the implementation. An interesting future investigation is adding the possibility of specifying scoping rules in an Xsemantics specification as well. This way, also the Xtext scope provider could be easily generated automatically by Xsemantics.

EriLex [32] is a software tool for generating support code for embedded domain specific languages and it supports specifying syntax, type rules, and dynamic semantics of such languages but it does not generate any artifact for IDE tooling.

An Xsemantics specification can access any Java type, not only the ones representing the AST. Thus, Xsemantics might also be used to validate any model, independently from Xtext itself, and possibly be used also with other language frameworks like EMFText [27]. Other approaches, such as, e.g., [33], [34], [35], [36], [37], [32], [14], instead require the programmer to use the framework also for defining the syntax of the language.

The importance of targeting IDE tooling when implementing a language was recognized also in older frameworks, such as Synthesizer [38] and Centaur [33]. In both cases, the use of a DSL for the type system was also recognized (the latter was using several formalisms [39], [40], [41]). Thus, Xsemantics enhances the usability of Xtext for developing prototype implementations of languages during the study of the formalization of languages.

We just mention other tools for the implementation of DSLs that are different from Xtext and Xsemantics for the main goal and programming context, such as, e.g., [42], [43], [44], which are based on language specification preprocessors, and [45], [46], which target host language extensions and internal DSLs.

Xsemantics does not aim at providing mechanisms for formal proofs for the language and the type system and it does not produce (like other frameworks do, e.g., [47], [29]), versions of the type system for proof assistants, such as Coq [48], HOL [49] or Isabelle [50]. However, Xsemantics can still help when writing the meta-theory of the language. An example of such a use-case, using the traces of the applied rules, can be found in [9].

We chose Xtext since it is the de-facto standard framework for implementing DSLs in the Eclipse ecosystem, it is continuously supported, and it has a wide community, not to mention many applications in the industry. Xtext is continuously evolving, and the main new features introduced in recent versions include the integration in other IDEs (mainly, IntelliJ), and the support for programming on the Web (i.e., an Xtext DSL can be easily ported on a Web application).

Finally, Xtext provides complete support for typical Java build tools, like Maven and Gradle. Thus, Xtext DSLs also automatically support these build tools. In that respect, Xsemantics provides Maven artifacts so that Xsemantics files can be processed during the Maven build in a Continuous Integration system.

III. XTEXT

In this section we will give a brief introduction to Xtext. In Section III-A we will also briefly describe the main features of Xbase, which is the expression language used in Xsemantics' rules.

It is out of the scope of the paper to describe Xtext and Xbase in details. Here we will provide enough details to make the features of Xsemantics understandable.

Xtext [5] is a *language workbench* (such as MPS [51] and Spoofax [19]): Xtext deals not only with the compiler mechanisms but also with Eclipse-based tooling. Starting from a grammar definition, Xtext generates an ANTLR parser [52]. During parsing, the AST is automatically generated by Xtext as an EMF model (Eclipse Modeling Framework [53]). Besides, Xtext generates many other features for the Eclipse editor for the language that we are implementing: syntax highlighting, background parsing with error markers, outline view, code completion.

Most of the code generated by Xtext can already be used as it is, but other parts, like type checking, have to be customized. The customizations rely on Google-Guice, a *dependency injection* framework [54].

In the following we describe the two complementary mechanisms of Xtext that the programmer has to implement. Xsemantics aims at generating code for both mechanisms.

Scoping is the mechanism for binding the symbols (i.e., references). Xtext supports the customization of binding with the abstract concept of *scope*, i.e., all declarations that are available (visible) in the current context of a reference. The programmer provides a `ScopeProvider` to customize the scoping. In Java-like languages the scoping will have to deal with types and inheritance relations, thus, it is strictly connected with the type system. For example, the scope for methods in the context of a method invocation expression consists of all the members, including the inherited ones, of the class of the *receiver* expression. Thus, in order to compute the scope, we need the type of the receiver expression.

Using the scope, Xtext will automatically resolve cross references or issue an error in case a reference cannot be resolved. If Xtext succeeds in resolving a cross reference, it also takes care of implementing IDE mechanisms like navigating to the declaration of a symbol and content assist.

All the other checks that do not deal with symbol resolutions, have to be implemented through a *validator*. In a Java-like language most validation checks typically consist in checking that the program is correct with respect to types. The validation takes place in background while the user is writing in the editor, so that an immediate feedback is available.

Scoping and validation together implement the mechanism for checking the correctness of a program. This separation into two distinct mechanisms is typical of other approaches, such as [38], [47], [16], [30], [29], [55].

A. *Xbase*

Xbase [56] is a reusable expression language that integrates completely with Java and its type system. Xbase also implements UI mechanisms that mimic the ones of the Eclipse Java Development Tools (JDT).

The syntax of Xbase is similar to Java with less "syntactic noise" (e.g., the terminating semicolon ";" is optional) and some advanced linguistic constructs. Although its syntax is not the same as Java, Xbase should be easily understood by Java programmers.

In this section we briefly describe the main features of Xbase, in order to make Xsemantics rules shown in the paper easily understandable for the Java programmers.

Variable declarations in Xbase are defined using `val` or `var`, for final and non-final variables, respectively. The type is not mandatory if it can be inferred from the initialization expression.

A cast expression in Xbase is written using the infix keyword `as`, thus, instead of writing "`(C) e`" we write "`e as C`".

Xbase provides *extension methods*, a syntactic sugar mechanism: instead of passing the first argument inside the parentheses of a method invocation, the method can be called with the first argument as its receiver. It is as if the method was one of the argument type's members. For example, if `m(E)` is an extension method, and `e` is of type `E`, we can write `e.m()` instead of `m(e)`. With extension methods, calls can be chained instead of nested: e.g., `o.foo().bar()` rather than `bar(foo(o))`.

Xbase also provides *lambda expressions*, which have the shape `[param1, param2, ... | body]`. The types of the parameters can be omitted if they can be inferred from the context. Xbase automatically compiles lambda expressions into Java anonymous classes; if the runtime Java library is version 8, then Xbase automatically compiles its lambda expressions into Java 8 lambda expressions.

All these features of Xbase allow the developer to easily write statements and expressions that are much more readable than in Java, and that are also very close to formal specifications. For example, a formal statement of the shape

"$\exists x \in \mathtt{L} \,.\, x \neq 0$"

can be written in Xbase like

"`L.exists[ x | x != 0 ]`".

This helped us a lot in making Xsemantics close to formal systems.

IV. XSEMANTICS

Xsemantics is a DSL (written in Xtext) for writing type systems, reduction rules and in general relation rules for languages implemented in Xtext. Xsemantics is intended for developers who are familiar with formal type systems and operational semantics since it uses a syntax that resembles rules in a formal setting (e.g., [7], [57], [8]).

```
judgments {
  type |- Expression expression : output Type
    error "cannot type " + expression
  subtype |- Type left <: Type right
    error left + " is not a subtype of " + right
}
```

Figure 1. An example of judgment definitions in Xsemantics.

A system definition in Xsemantics is a set of *judgments*, that is, assertions about the properties of programs, and a set of *rules*. Rules can be seen as implications between judgments, i.e., they assert the validity of certain judgments, possibly on the basis of other judgments [7]. Rules have a conclusion and a set of premises. Typically, rules act on the EMF objects representing the AST, but in general they can refer to any Java class. Starting from the definitions of judgments and rules, Xsemantics generates Java code that can be used in a language implemented in Xtext for scoping and validation.

A. *Judgments*

An Xsemantics judgment consists of a name, a *judgment symbol* (which can be chosen from some predefined symbols) and the *parameters* of the judgment. Parameters are separated by *relation symbol*s (which can be chosen from some predefined symbols).

Currently, Xsemantics only supports a predefined set of symbols, in order to avoid possible ambiguities with expression operators in the premises of rules.

Judgment symbols are

```
||-   |-   ||~   |~   ||=   |=   ||>   |>
```

Relation symbols are

```
<!   !>   <<!   !>>   <~!   !~>
 :   <:   :>    <<    >>    ~~
<|   |>   <~    ~>    \/    /\
```

All these symbols aim at mimicking the symbols that are typically used in formal systems.

Two judgments must differ for the judgment symbol or for at least one relation symbol. The parameters can be either input parameters (using the same syntax for parameter declarations as in Java) or output parameters (using the keyword `output` followed by the Java type). For example, the judgment definitions for an hypothetical Java-like language are shown in Figure 1: the judgment `type` takes an `Expression` as input parameter and provides a `Type` as output parameter. The judgment `subtype` does not have output parameters, thus its output result is implicitly boolean. Judgment definitions can include `error` specifications (described in Section IV-F), which are useful for generating informative error information.

B. *Rules*

Rules implement judgments. Each rule consists of a name, a *rule conclusion* and the *premises* of the rule. The conclusion consists of the name of the *environment* of the rule, a *judgment symbol* and the *parameters* of the rules, which are separated by *relation symbols*. To enable better IDE tooling and a more "programming"-like style, Xsemantics rules are written in the opposite direction of standard deduction rules, i.e., the conclusion comes before the premises (similar to other frameworks, like [29], [31]).

The elements that make a rule belong to a specific judgment are the judgment symbol and the relation symbols that separate the parameters. Moreover, the types of the parameters of a rule must be Java subtypes of the corresponding types of the judgment. Two rules belonging to the same judgment must differ for at least one input parameter's type. This is a sketched example of a rule, for a Java-like method invocation expression, of the judgment `type` shown in Figure 1:

```
rule MyRule
  G |− MethodSelection exp : Type type
from {
  // premises
  type = ... // assignment to output parameter
}
```

The rule *environment* (in formal systems it is usually denoted by Γ and, in the example it is named `G`) is useful for passing additional arguments to rules (e.g., contextual information, bindings for specific keywords, like `this` in a Java-like language). An empty environment can be passed using the keyword `empty`. The environment can be accessed with the predefined function `env`.

Xsemantics uses Xbase to provide a rich Java-like syntax for defining rules. The premises of a rule, which are specified in a `from` block, can be any Xbase expression (described in Section III-A), or a *rule invocation*. If one thinks of a rule declaration as a function declaration, then a rule invocation corresponds to a function invocation, thus one must specify the environment to pass to the rule, as well as the input and output arguments.

In a rule invocation, one can specify additional *environment mappings*, using the syntax `key <- value` (e.g., `'this' <- C`). When an environment is passed to a rule with additional mappings, actually a brand new environment is passed, thus the current rule environment will not be modified. If a mapping with the same key exists in the current environment, then in the brand new environment (and only there) the existing mapping will be overwritten. Thus, the rule environment passed to a rule acts in a stack manner.

The premises of an Xsemantics rule are considered to be in *logical and* relation and are verified in the same order they are specified in the block. If one needs premises in *logical or* relation, the operator `or` must be used to separate blocks of premises.

If a rule does not require any premise, we can use a special kind of rule, called *axiom*, which only has the conclusion.

In the premises, one can assign values to the output parameters, as shown in the previous rule example. When another rule is invoked, upon return, the output arguments will have the values assigned in the invoked rule. Alternatively, an expression can be directly specified instead of the output parameter in the rule conclusion.

If one of the premises fails, then the whole rule will fail, and in turn the stack of rule invocations will fail. In particular, if the premise is a boolean expression, it will fail if the expression evaluates to false. If the premise is a rule invocation, it will fail if the invoked rule fails. An explicit failure can be triggered using the keyword `fail`.

At runtime, upon rule invocation, the generated Java system will select the most appropriate rule according to the runtime types of the passed arguments (using the *polymorphic dispatch* mechanism provided by Xtext, which performs method dispatching according to the runtime type of arguments). Note that, besides this strategy for selecting a specific rule, Xsemantics itself does not implement, neither it defines, any other strategy. It is Xtext that decides when a part of a program has to be validated or a symbol has to be bound. This is consistent with the nature of frameworks, which dictate the overall program's flow of control.

C. Auxiliary Functions

Besides judgments and rules, one can write *auxiliary functions*. In type systems, such functions are typically used as a support for writing rules in a more compact form, delegating some tasks to such functions (for example, see [8]). *Predicates* can be seen as a special form of auxiliary functions.

D. Checkrules

In an Xsemantics system, we can specify some special rules, *checkrules*, which do not belong to any judgment. They are used by Xsemantics to generate a Java validator for the Xtext language. A checkrule has a name, a single parameter (which is the AST object to be validated) and the premises (but no rule environment). The syntax of the premises of a checkrule is the same as in the standard rules.

The Java validator generated by Xsemantics will automatically generate error markers for failed rules. Error markers will be automatically generated according to the error information found in the trace of a failure, which is computed and handled automatically by Xsemantics. When generating error markers the validator will use only the error information related to elements in the AST. The error marker will be generated in correspondence to the innermost failure, because this is usually the most informative error message. A custom error specification can be attached to a judgment or to a single rule, as described in Section IV-F.

E. Fields and Imports

Fields can be defined in an Xsemantics system. Such fields will be available to all the rules, checkrules and auxiliary functions, just like Java fields in a class are available to all methods of the class. This way, it is straightforward to reuse external Java utility classes from an Xsemantics system. This is useful when some mechanisms are easier to implement in Java than in Xsemantics.

Xsemantics also supports Java-like import statements, including Java static imports. This way, external Java classes' static methods can be used from within Xsemantics premises without the fully qualified name. In particular, the Xsemantics Eclipse editor supports automatic insertion of imports during code completion. This mimics what the Eclipse Java editor does. Both fields and static imports can be further decorated with the `extension` specification. This will enable the *extension methods* mechanism of Xbase (described in Section III-A) making Xsemantics code less verbose and more compact.

F. Error Information

Custom error information can be specified on judgments, rules and auxiliary functions. This can be used for providing error information that is useful in specific scenarios.

When specifying a custom error information, using the keyword `error` followed by a string describing the error, the developer can also specify the `source` element in the program that will be marked with error. Additional `data` can be attached to the error information that can be later reused in case of custom error handling.

Moreover, when using the explicit failure keyword `fail`, a custom error information can be specified as well. This use of `fail` is useful together with *or* blocks to provide more information about the error.

For example, consider the boolean premise

```
args.size() == params.size()
```

that checks that the number of arguments is equal to the number of parameters in a Java-like method invocation. If that premise fails, the default error message will show the original expression text, specifying that it failed. This would not be useful for the user (it would show an error with implementation details). To generate a better error message we can write

```
args.size() == params.size()
or
fail error "expected " + params.size() +
  " arguments, but was " + args.size()
```

There might be cases when we want to show errors containing more information about the cause that made a rule invocation fail, especially when the failure took place because of a rule invocation that is deep in the rule invocation stack. For such cases, the implicit variable `previousFailure` is available. This is automatically handled by Xsemantics at run-time: in case of a rule failure, it provides the developer with all the problems that took place when applying the rule. This allows us to build informative error messages as shown in Section VI-A.

G. Caching

In a language implemented with Xtext, types are used in many places by the framework, e.g., in the scope provider, in the validator and in the content assist. Besides that, some type computation rules, some subtyping checks and some auxiliary functions are also used more than once from the type system implementation itself. For example, the subtyping relation between the same two classes can be checked by many checkrules for the same sources.

For the above reasons, the results of type computations should be cached to improve the performance of the compiler and, most of all, the responsiveness of the Eclipse editor. However, caching usually introduces a few levels of complexity in implementations, and, in the context of an IDE that performs background parsing and checking, we also need to keep track of changes that should invalidate the cached values. Xsemantics provides automatic caching mechanisms that can be enabled in a system specification. These mechanisms internally use at run-time a cache that stores its values in the scope of a resource (a resource is an abstraction of a source file). The cached values will be automatically discarded as soon as the contents of the program changes. When caching is enabled in an Xsemantics system specification, then Xsemantics will generate Java code that automatically uses the cache, hiding the details from the programmer.

The programmer can enable caching, using the keyword `cached` on a per-judgment basis. The rationale behind this granularity is that caching should be enabled with care, otherwise it could decrease the performance. In fact, the caching is based on the Java hashing features, thus it makes sense only when used with actual object instances, not with references. In fact, in the AST of a program there might be many different references to the same object, and using such references as cache keys will only lead to many cache misses. Thus, it is responsibility of the programmer to be aware of which judgments and auxiliary functions to cache, depending on the nature of the involved input parameters.

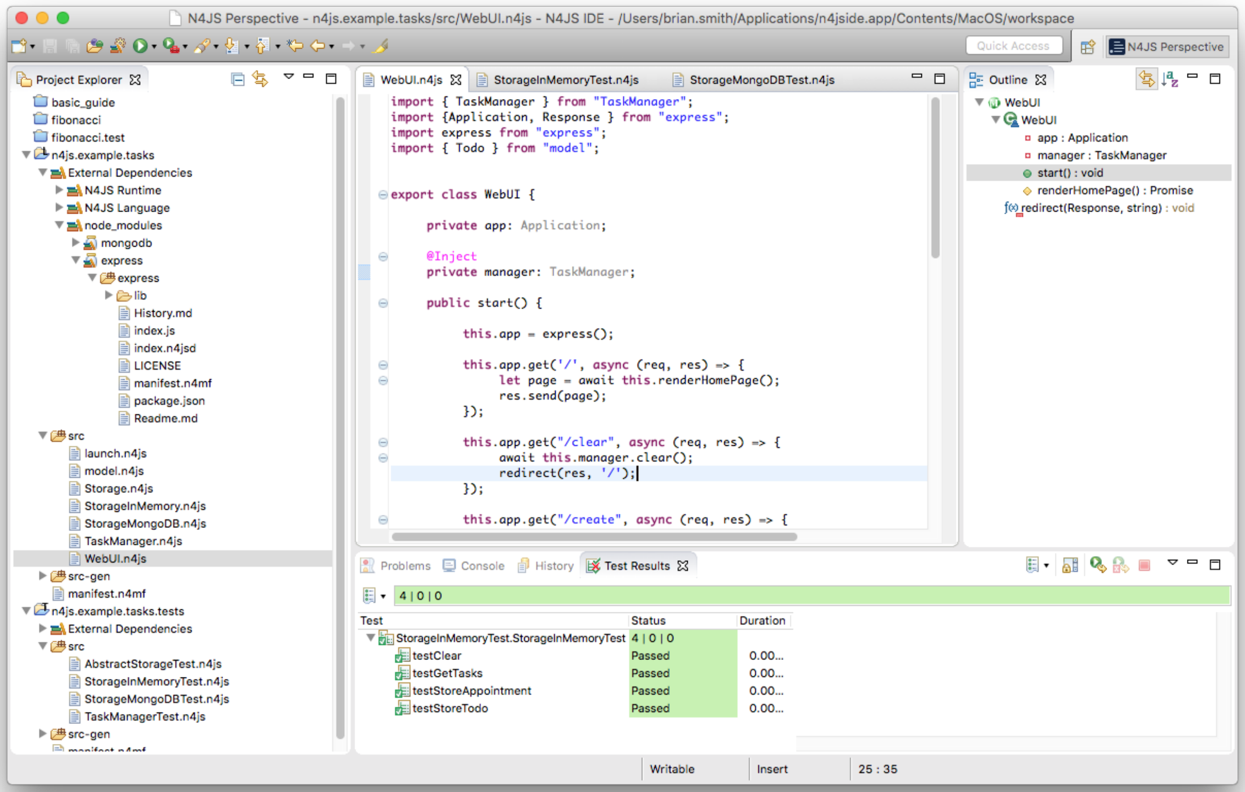
The use of caching for the implementation of the N4JS' type system is described in Section VI-B.

H. Extensions

When defining a system in Xsemantics it is also possible to extend another Xsemantics system, using `extends`. Just like in Java class inheritance, an extended system implicitly inherits from the "super system" all judgments, rules, check rules and auxiliary functions. In the extended system one can override any such element; the overriding follows the same Java overriding rules (e.g., the types of input parameters must be the same and the types of output parameters can be subtypes). For example, an axiom in a super system can be turned into a rule in the extended system and vice versa. Similarly, we can override a judgment of the super system changing the names of parameters and error specifications. Since an Xtext grammar can inherit from another grammar, Xsemantics system extensions can be used when the language we inherit from already implements a type system using Xsemantics. Moreover, we used system extension to quickly test possible modifications/improvements to an existing type system, e.g., for testing that caching features do not break the type system implementation.

V. N4JS—A Typed JavaScript

We have used Xsemantics to implement the type system of a real-world programming language called N4JS. In this section, we give an overview of the syntax and semantics of N4JS, before presenting the Xsemantics-based implementation of the N4JS type system in Section VI.


N4JS Perspective - n4js.example.tasks/src/WebUI.n4js - N4JS IDE - /Users/brian.smith/Applications/n4jside.app/Contents/MacOS/workspace

```
basic_guide
fibonacci
fibonacci.test
n4js.example.tasks
  External Dependencies
    N4JS Runtime
    N4JS Language
    node_modules
      mongodb
      express
        express
          lib
          History.md
          index.js
          index.n4jsd
          LICENSE
          manifest.n4mf
          package.json
          Readme.md
  src
    launch.n4js
    model.n4js
    Storage.n4js
    StorageInMemory.n4js
    StorageMongoDB.n4js
    TaskManager.n4js
    WebUI.n4js
  src-gen
  manifest.n4mf
n4js.example.tasks.tests
  External Dependencies
  src
    AbstractStorageTest.n4js
    StorageInMemoryTest.n4js
    StorageMongoDBTest.n4js
    TaskManagerTest.n4js
  src-gen
```

```
import { TaskManager } from "TaskManager";
import {Application, Response } from "express";
import express from "express";
import { Todo } from "model";


export class WebUI {

    private app: Application;

    @Inject
    private manager: TaskManager;

    public start() {

        this.app = express();

        this.app.get('/', async (req, res) => {
            let page = await this.renderHomePage();
            res.send(page);
        });

        this.app.get("/clear", async (req, res) => {
            await this.manager.clear();
            redirect(res, '/');
        });

        this.app.get("/create", async (req, res) => {
```

```
WebUI
  WebUI
    app : Application
    manager : TaskManager
    start() : void
    renderHomePage() : Promise
  redirect(Response, string) : void
```

4 | 0 | 0

Test	Status	Duration
StorageInMemoryTest.StorageInMemoryTest	4 | 0 | 0	
testClear	Passed	0.00...
testGetTasks	Passed	0.00...
testStoreAppointment	Passed	0.00...
testStoreTodo	Passed	0.00...

Writable | Insert | 25 : 35

Figure 2. The N4JS IDE, showing the editor with syntax highlighting, the outline with type information, and the integrated test support.

A. *Overview and Background*

Before going into technical details of the language itself, let us provide some context. NumberFour AG is developing a platform for applications specifically tailored towards small businesses that will, on the client side, target JavaScript and the browser (among other targets such as iOS). Due to a large overall code base, high reliability requirements, interoperability with statically typed languages and their ecosystems, such as Scala or ObjectiveC, and for maintenance reasons, an explicit declaration of types together with static type checking was deemed a necessity across all targets, in particular JavaScript. This set of requirements led us to the development of the N4JS language.

N4JS is an extension of JavaScript, also referred to as ECMAScript, with modules and classes as specified in the ECMAScript 2015 standard [58]. Most importantly, N4JS adds a full-featured static type system on top of JavaScript, similar to TypeScript [59] or Dart [60]. N4JS compiles to plain JavaScript and provides seamless interoperability with JavaScript. In fact, most valid ECMAScript 2015 code is valid N4JS code (not considering type errors), but some recent features of ECMAScript 2015 are not supported yet. N4JS is still under development, but has been used internally at NumberFour AG for more than two years now, and was released as open source in March 2016, `https://numberfour.github.io/n4js/`.

Roughly speaking, N4JS' type system could be described as a combination of the type systems provided by Java, TypeScript and Dart. Besides primitive types, already present in ECMAScript, it provides declared types such as classes and interfaces, also supporting default methods (i.e., mixins), and combined types such as *union types* [61]. N4JS supports generics similar to Java, that is, it supports generic types and generic methods (which are supported by TypeScript but not by Dart) including wildcards, requiring the notion of existential types (see [62]).

Beyond its type-system-related features, the implementation of N4JS provides a transpiler that transforms N4JS code into ECMAScript 2015 code and a full featured IDE, shown in Figure 2. This IDE provides some advanced editing features such as

- live code validation, i.e., type errors and other code issues are shown and updated in the editor while the programmer is typing,
- a project explorer for managing large, multi-project code bases (left-hand side of the screenshot),
- convenience UI features such as an outline view showing the classes, interfaces, their fields and methods together with their signature and other elements defined in the currently active editor (right-hand side of the screenshot).

Since this IDE is based on the Eclipse framework, it inherits many features from Eclipse and readily available Eclipse plugins, without the need of any N4JS-specific implementation, for example, a seamless integration of the git version management system.

At NumberFour AG, N4JS and its IDE are being developed using agile development methodologies, in particular the Scrum process and test-driven development, aiming for a high test coverage with around 110.000 tests, at the moment, including a comprehensive test suite to ensure compatibility with ECMAScript 2015 syntax. Apache Maven is used as build tool and Jenkins for continuous integration.

The main frameworks being used are Xtext and Xsemantics, as introduced above. Most tests, especially those related to the type system, are written using the Xpect framework, `http://www.xpect-tests.org/`, which allows the developer to formulate test cases in the language being developed, i.e., N4JS in this case. For example, a test case for asserting that a number cannot be assigned to a variable of type `string` could be written as follows:

```
/* XPECT_SETUP N4JSSpecTest END_SETUP */

let x: string;
/* XPECT errors ---
   "int is not a subtype of string." at "123"
--- */
x = 123;
```

All information for testing is provided in ordinary N4JS comments, enclosed in `/* */`. The second to last line in the above code example shows an Xpect test expectation using a so-called *Xpect test method* called `errors` that asserts a particular error message at a particular location in the source (the integer literal `123` in this example) and fails if no error or a different error occurs. There are other test methods for checking for warnings or asserting the correct type of an element or expression.

In addition to N4JS, there exist other JavaScript dialects augmenting the language by static types with compile-time type checking, most notably TypeScript [59]. All these languages are facing the same, fundamental challenge: dynamic, untyped languages and statically typed languages follow two clearly distinct programming paradigms, leading in practice to different programming styles, idioms, and ecosystems of frameworks and libraries. Thus, when adding a static type system on top of an existing dynamic language there is a risk of breaking established programming idioms and conventions of the dynamic language. This is where an important difference between N4JS and, for example, TypeScript lies. While TypeScript aims for as much type safety as possible without sacrificing typical JavaScript idioms and convenience, N4JS would rather risk getting in the way of the typical JavaScript programmer from time to time than sacrificing type safety. N4JS aims at providing type soundness as its sole primary priority without compromises for the sake of programmer familiarity or convenience. In other words, TypeScript is designed primarily with existing JavaScript programmers in mind, whereas N4JS is more tailored to developers accustomed to statically typed languages, mainly Java and C# (though the needs of pure JavaScript developers and support for legacy JavaScript code bases have been considered as far as possible).

```
interface NamedEntity {
    get name(): string;
}
class Person implements NamedEntity { // class
    implementing an interface
    // three field declarations:
    public firstName: string;
    public lastName: string;
    protected age: number;
    // a method
    public marry(spouse: Person) {
        let familyName = this.lastName + '-' +
            spouse.lastName;
        this.lastName = familyName;
        spouse.lastName = familyName;
    }
    // a getter (required by interface)
    @Override get name(): string {
        return this.firstName + ' ' + this.lastName;
    }
}
class Employee extends Person { // a class extending
     another class
    private _salary: number;
    // a getter/setter pair:
    get salary(): number {
        return this._salary;
    }
    set salary(amount: number) {
        if(amount>=0) {
            this.salary = amount;
        }
    }
}
```

Figure 3. Defining classes, interfaces and their members in N4JS.

Language features where this difference in priorities can be observed include TypeScript's flexible yet unsafe default handling of built-in type `any` as well as TypeScript's use of bivariant function subtyping and method overriding, which has been identified by recent studies [63] as a form of unsoundness in the name of convenience and flexibility without clear usability benefits in practice (the latest TypeScript version has an optional compiler flag for activating a more rigid handling of `any`). In comparison, `any`, function subtyping and method overriding are handled in N4JS in a strictly type-safe manner according to how these concepts are commonly defined in the literature on object-oriented programming and type theory in general (e.g., strict observance of the Liskov substitution principle [64]). A more detailed comparison of N4JS with other typed JavaScript dialects is, however, beyond the scope of this article.

After this brief overview of why and how N4JS is being developed, we will now, for the remainder of Section V, focus on the syntax and semantics of the language itself.

B. Basic Language Features

N4JS provides the typical language features expected from an object-oriented language, such as declaration of classifiers (i.e., classes and interfaces), inheritance, polymorphism, etc. We do not aim for a full description of the language here, but Figure 3 is provided to give an impression of the syntax of the most common constructs.

In addition to fields and methods, classifiers may have so-called *getters* and *setters* members, which are similar to methods but are invoked as if the containing classifier had a field of that name. For example, the getter/setter pair salary of class Employee in Figure 3 would be invoked as follows:

```
let e = new Employee();
e.salary = 42;
console.log(e.salary);
```

Line 2 invokes setter salary with a value of 42 as argument and line 3 invokes the getter. Providing only either a getter or setter amounts to a read-only / write-only field from outside the type.

For the remainder of this section we focus on language features of N4JS that are less common and are most relevant from a type system perspective.

C. Union and Intersection Types

As a first advanced feature, N4JS allows the programmer to combine two or more existing types to form a new type that represents the union or intersection of the combined types. Given types $T_1, \dots, T_n$, the union U of these types is a supertype of another type T' if and only if T' is a subtype of at least one of the combined types and is a subtype of T' if and only if T' is a subtype of all the combined types.

More formally, given types $T_1, \dots, T_n$ and the union U of these types, for all types T' we have

$$T' <: U \iff T' <: T_1 \vee \dots \vee T' <: T_n \tag{1}$$

$$U <: T' \iff T_1 <: T' \wedge \dots \wedge T_n <: T' \tag{2}$$

In N4JS source code, the union of two types A and B is written as A|B. The following N4JS code snippet provides an example:

```
function foo(param: string | number) {
    let str: string;
    str = param; // ERROR
}
foo('hello'); // ok
foo(42); // ok
```

Function foo can be invoked with strings and with numbers as argument, because string is a subtype of the union string|number and also number is a subtype of this union. However, the assignment inside function foo fails, because union string|number is not a subtype of string (and also not a subtype of number).

Intersection types are defined accordingly: the intersection is a subtype of all its combined types. The notation for the intersection of two types A and B in N4JS source code is A&B.

Intersection types are actually more common in mainstream languages than union types. Java has support for intersection types, but they can only be used in very few places, for example, when declaring the upper bound of a type parameter:

```
public interface I {}
public interface J {}
public class G<T extends I & J> {
    /* ... */
}
```

Here, type parameter T has the intersection type I & J as its upper bound; Java developers often view this as parameter T having two upper bounds I and J.

From a practical point of view, union types are particularly important in N4JS, because N4JS—just as ECMAScript 2015—does not allow method overloading. So, union types are a means to provide methods than can handle different types of arguments in much the same way as done in Java using method overloading.

D. Nominal and Structural Typing

The fundamental notion for reasoning about types is the *subtype relation*. According to the Liskov substitution principle [64], given two types S, T we call S a subtype of T if and only if every property that can be observed from the outside for T, does also apply to S, and we can thus use instances of S wherever instances of T are expected.

One of the primary responsibilities of a type system is to decide whether a given type is, in the above sense, a subtype of another type. N4JS provides support for different strategies of checking whether two types are subtypes of one another, namely nominal and structural typing [8]. Additionally, it provides certain variations of structural typing to support typical use cases of ECMAScript.

In the context of a programming language, a type S is a subtype of type T if, roughly speaking, a value of type S may be used as if it were a value of type T. Therefore, if type S is a subtype of T, denoted as $S <: T$, a value that is known to be of type S may, for example, be assigned to a variable of type T or may be passed as an argument to a function expecting an argument of type T. There are two major classes of type systems that differ in how they decide on such type compatibility:

- *Nominal type systems*, as known from most object-oriented programming languages, e.g., Java, C#.
- *Structural type systems*, as more common in type theory and functional programming languages.

Since N4JS provides both forms of typing, we briefly introduce each approach in the following sections before we show how they are combined in N4JS.

1) Nominal Typing: In a *nominal*, or *nominative*, type system, two types are deemded to be the same if they have the *same name* and a type S is deemed to be an (immediate) subtype of a type T if and only if S is *explicitly declared* to be a subtype of T.

In the following example, Employee is a subtype of Person because it is declared as such using keyword extends within its class declaration. Conversely, Product is not a subtype of Person because it lacks such an "extends" declaration.

```
class Person {
    public name: string;
}

class Employee extends Person {
    public salary: number;
}
```

```
class Manager extends Employee { }

class Product {
    public name: string;
    public price: number;
}
```

The subtype relation is transitive and thus `Manager` is not only a subtype of `Employee` but also of `Person`. `Product` is not a subtype of `Person`, although it provides the same members.

Most mainstream object-oriented languages use nominal subtyping, for example, C++, C#, Java, Objective-C.

2) Structural Typing: In a *structural* type system, two types are deemed the same if they are of the same *structure*. In other words, if they have the same public fields and methods of compatible type/signature. Similarly, a type S is deemed a subtype of a type T if and only if S has all public members (of compatible type/signature) that T has (but may have more).

In the example from the previous section, we said `Product` is not a (nominal) subtype of `Person`. In a structural type system, however, `Product` would indeed be deemed a (structural) subtype of `Person` because it has all of `Person`'s public members of compatible type (the field `name` in this case). The opposite is, in fact, not true: `Person` is not a subtype of `Product` because it lacks `Product`'s field `price`.

3) Comparison: Both classes of type systems have their own advantages and proponents [65]. Nominal type systems are usually said to provide more type safety and better maintainability whereas structural typing is mostly believed to be more flexible. As a matter of fact, nominal typing *is* structural typing extended with an extra relation explicitly declaring one type a subtype of another, e.g., the `extends` clause in case of N4JS. So the real question is: What are the advantages and disadvantages of such an explicit relation?

Let us assume we want to provide a framework or library with a notion of indentifiable elements, i.e., elements that can be identified by name. We would define an interface as follows:

```
export public interface Identifiable {
    public get name(): string

    static checkNom(identifiable: Identifiable):
        boolean {
        return identifiable.name !== 'anonymous';
    }
    static checkStruct(identifiable: ~Identifiable):
         boolean {
        return identifiable.name !== 'anonymous';
    }
}
```

A nominal implementation of this interface could be defined as

```
import { Identifiable } from 'Identifiable';

class AN implements Identifiable {
    @Override
    public get name(): string { return 'John'; }
}
```

whereas here is a structural implementation of above interface:

```
class AS {
    public get name(): string { return 'John'; }
}
```

A client may use these classes as follows:

```
Identifiable.checkNom(new AN());
Identifiable.checkNom(new AS()); // ERROR "AS is not
      a (nominal) subtype of Identifiable"
Identifiable.checkStruct(new AN());
Identifiable.checkStruct(new AS());
```

Let us now investigate advantages and disadvantages of the two styles of subtyping based on this code; we will mainly focus on maintainability and flexibility.

Maintainability. As a refactoring, consider renaming `name` to `id` in order to highlight that the name is expected to be unique. Assume you have thousands of classes and interfaces. You start by renaming the getter in the interface:

```
export public interface Identifiable {
    public get id(): string
    // ...
}
```

With structural typing, you will not get any errors in your framework. You are satisfied with your code and ship the new version. However, client code outside your framework will no longer work as you have forgotten to accordingly rename the getter in class `AS` and so `AS` is no longer a (structural) subtype of `Identifiable`.

With nominal typing, you would have gotten errors in your framework code already at compile time: “Class AN must implement getter id.” and “The getter name must implement a getter from an interface.” Instead of breaking the code on the client side only, you find the problem in the framework code. In a large code base, this is a huge advantage. Without such a strict validation, you probably would not dare to refactor your framework. Of course, you may still break client code, but even then it is much easier to pinpoint the problem.

Flexibility. Given the same code as in the previous example, assume that some client code also uses another framework providing a class `Person` with the same public members as `AN`, `AS` in the above example. With structural typing, it is no problem to use `Person` with static method `checkStruct()` since `Person` provides a public data field `name` and is thus a structural subtype of `Identifiable`. So, the code inside the method would work as intended when called with an instance of `Person`.

This will not be possible with nominal typing though. Since `Person` does not *explicitly* implement `Identifiable`, there is no chance to call method `checkNom()`. This can be quite cumbersome, particularly if the client can change neither your framework nor the framework providing class `Person`.

4) Combination of Nominal and Structural Typing: Because both classes of type systems have their advantages and because structural typing is particularly useful in the context of a dynamic language ecosystem such as the one of JavaScript, N4JS provides both kinds of typing and aims to combine them in a seamless way.

N4JS uses nominal typing by default, but allows the programmer to switch to structural typing by means of special type constructors using the tilde symbol. The switch can be done with either of the following:

- Globally when defining a type. This then applies to all uses of the type throughout the code, referred to as *definition-site structural typing*
- Locally when referring to an existing nominal type, referred to as *use-site structural typing*.

For the latter we have already seen an example in the signature of static method `checkStruct()`. For its parameter `elem` we used a (use-site) structural type by prefixing the type reference with a ∼ (tilde), which means we are allowed, when invoking `checkStruct()`, to pass in an instance of `AS` or `Person` even though they are not nominal subtypes of `Identifiable`.

This way, N4JS provides the advantages of nominal typing (which is the default form of typing) while granting many of the advantages of structural typing, if the programmer decides to use it. Additionally, if you rename `name` to `id`, the tilde will tell you that there may be client code calling the method with a structural type.

The full flexibility of a purely structurally typed language, however, cannot be achieved with this combination. For example, the client of an existing function or method that is declared to expect an argument of a nominal type N is confined to nominal typing. They cannot choose to invoke this function with an argument that is only a structural subtype of N (it would be a compile time error). This could possibly be exactly the intention of the framework's author in order to enable easier refactoring later.

5) Field Structural Typing: N4JS provides some variants of structural types. Usually two structural types are compatible, if they provide the same properties, or in case of classes, public members. In ECMAScript we often only need to access the fields. In N4JS, we can use ∼∼ to refer to the so-called *field structural type*. Two field structural types are compatible, if they provide the same `public` fields. Methods are ignored in these cases. Actually, N4JS provides even more options. There are several modifiers to further filter the properties or members to be considered:

- `∼r∼` only considers getters and data fields,
- `∼w∼` only considers setters and data fields,
- `∼i∼` is used for initializer parameters: for every setter or (non-optional) data field in the type, the `∼i∼`-type needs to provide a getter or (readable) data field.

E. Parameterized Types

Generics in N4JS are a language feature that allows for generic programming. They enable a function, class, interface, or method to operate on the values of various (possibly unknown) types while preserving compile-time type safety. There are some differences with respect to Java generics, which we shall describe below.

1) Motivation: Several language elements may be declared in a generic form; we will start with focusing on classes, generic methods will be discussed after that.

The standard case, of course, is a non-generic class. Take the following class, for example, that aggregates a pair of two strings:

```
export public class PairOfString {
    first: string;
    second: string;
}
```

This implementation is fine as long as all we ever want to store are strings. If we wanted to store numbers, we would have to add another class:

```
export public class PairOfNumber {
    first: number;
    second: number;
}
```

Following this pattern of adding more classes for new types to be stored obviously has its limitations. We would soon end up with a multitude of classes that are basically serving the same purpose, leading to code duplication, bad maintainability and many other problems.

One solution could be having a class that stores two values of type `any` (in N4JS, `any` is the so-called *top type*, the common supertype of all other types).

```
export public class PairOfWhatEver {
    first: any;
    second: any;
}
```

Now the situation is worse off than before. We have lost the certainty that within a single pair, both values will always be of the same type. When reading a value from a pair, we have no clue what its type might be.

2) Generic Classes and Interfaces: The way to solve our previous conundrum using generics is to introduce a *type variable* for the class. We will then call such a class a *generic class*. A type variable can then be used within the class declaration just as any other ordinary type.

```
export public class Pair<T> {
    first: T;
    second: T;
}
```

The type variable `T`, declared after the class name in angle brackets, now represents the type of the values stored in the `Pair` and can be used as the type of the two fields.

Now, whenever we refer to the class `Pair`, we will provide a *type argument*, in other words a type that will be used wherever the type variable `T` is being used inside the class declaration.

```
import { Pair } from 'Pair';

let myPair = new Pair<string>();
myPair.first = '1st value';
myPair.second = '2nd value';
```

By using a type variable, we have not just allowed any given type to be used as value type, we have also stated that both values, first and second, must always be of the same type. We have also given the type system a chance to track the types of values stored in a `Pair`:

```
import { Pair } from 'Pair';

let myPair2 = new Pair<string>();
myPair2.first = '1st value';
myPair2.second = 42; // error: 'int is not a subtype
      of string.'

console.log(myPair2.first.charAt(2));
// type system will know myPair2.first is of type
     string
```

The error in line 3 shows that the type checker will make sure we will not put any value of incorrect type into the pair. The fact that we can access method `charAt()` (available on strings) in the last line indicates that when we read a value from the pair, the type system knows its type and we can use it accordingly.

Generic interfaces can be declared in exactly the same way.

3) Generic Functions and Methods: With the above, we can now avoid introducing a multitude of classes that are basically serving the same purpose. It is still not possible, however, to write code that manipulates such pairs regardless of the type of its values may have. For example, a function for swapping the two values of a pair and then return the new first value would look like this:

```
import { PairOfString } from 'PairOfString';

function swapStrings1(pair: PairOfString): string {
    let backup = pair.first; // inferred type of '
        backup' will be string
    pair.first = pair.second;
    pair.second = backup;
    return pair.first;
}
```

The above function would have to be copied for every value type to be supported. Using the generic class `Pair<T>` does not help much:

```
import { Pair } from 'Pair';

function swapStrings2(pair: Pair<string>): string {
    let backup = pair.first; // inferred type of '
        backup' will be string
    pair.first = pair.second;
    pair.second = backup;
    return pair.first;
}
```

The solution is not only to make the type being manipulated generic (as we have done with class `Pair<T>` above) but to make the code performing the manipulation generic:

```
import { Pair } from 'Pair';

function <T> swap(pair: Pair<T>): T {
    let backup = pair.first; // inferred type of '
        backup' will be T
    pair.first = pair.second;
    pair.second = backup;
    return pair.first;
}
```

We have introduced a type variable for function `swap()` in much the same way as we have done for class `Pair` in the previous section (we then call such a function a *generic function*). Similarly, we can use the type variable in this function's signature and body.

It is possible to state in the declaration of the function `swap()` above that it will return something of type `T` when having obtained a `Pair<T>` without even knowing what type that might be. This allows the type system to track the type of values passed between functions and methods or put into and taken out of containers, and so on.

Generic methods can be declared just as generic functions. There is one caveat, however: Only if a method introduces its own new type variables it is called a generic method. If it is merely using the type variables of its containing class or interface, it is an ordinary method. The following example illustrates the difference:

```
export public class Pair<T> {

    public foo(): T {      }
    public <S> bar(pair: Pair2<S>): void { /*...*/ }
}
```

The first method `foo` is a non generic method, while the second one `bar` is.

A very interesting application of generic methods is when using them in combination with function type arguments:

```
class Pair<T> {

    <R> merge(merger: {function(T,T): R}): R {
        return merger(this.first, this.second);
    }
}

var p = new Pair<string>();
/* ... */
var i = p.merge( (f,s)=> f.length+s.length )
```

You will notice that N4JS can infer the correct types for the arguments and the return type of the arrow expression. Also, the type for `i` will be automatically computed.

4) Differences to Java: Important differences between generics in Java and N4JS include:

- Primitive types can be used as type arguments in N4JS.
- There are no raw types in N4JS. Whenever a generic class or interface is referenced, a type argument has to be provided - possibly in the form of a wildcard. For generic functions and methods, an explicit definition of type arguments is optional if the type system can infer the type arguments from the context.

F. *Use-site and Definition-Site Variance*

In the context of generic types, the "variance of a generic type $G\langle T_1,...,T_n\rangle$ in T_i, $i \in \{1,...,n\}$," tells how G behaves with respect to subtyping when changing the type argument for type parameter T_i. In other words, knowing $X <: Y$, does this tell us anything about whether either $G\langle X\rangle <: G\langle Y\rangle$ or $G\langle Y\rangle <: G\langle X\rangle$ holds?

More formally, given a type $G\langle T_1,...,T_n\rangle$ and $i \in \{1,...,n\}$, we say

- G is *covariant* in T_i if and only if

$$\forall X,Y : X <: Y \Rightarrow G\langle X\rangle <: G\langle Y\rangle \qquad (3)$$

- G is *contravariant* in T_i if and only if

$$\forall X,Y : X <: Y \Rightarrow G\langle Y\rangle <: G\langle X\rangle \qquad (4)$$

If neither applies, we call G *invariant* in T_i. For the sake of conciseness, the case that both applies is not discussed, here.

In N4JS, the variance of a generic type G can be declared both on use-site, e.g., when referring to G as the type of a formal parameter in a function declaration, or on definition-site, i.e., in the class or interface declaration of G, and these two styles of declaring variance can be combined seamlessly.

For further investigating these two styles and for showing how they are integrated in N4JS, we first introduce an exemplary, single-element container class G as follows:

```
class G<T> {
    private elem: T;

    put(elem: T) { this.elem = elem; }
    take(): T { return this.elem; }
}
```

In addition, for illustration purposes, we need three helper classes $C <: B <: A$:

```
class A {}
class B extends A {}
class C extends B {}
```

1) Use-site Variance: N4JS provides support for *wildcards*, as known from Java [66]. In the source code, a wildcard is represented as `?` and can be used wherever type arguments are provided for the type parameters of a generic type. Furthermore, wildcards can be supplied with upper or lower bounds, written as `? extends U` and `? super L`, with U,L being two types, here used as upper and lower bound, respectively.

Figure 4 shows three functions that all take an argument of type G, but using a different type argument for G's type parameter T.

The effect of the different type arguments becomes apparent when examining invocations of these functions. Using helper variables

```
let ga: G<A> = /* ... */ ;
let gb: G<B> = /* ... */ ;
let gc: G<C> = /* ... */ ;
```

we start with `fun1` by invoking it with each helper variable. We get:

```
fun1(ga); // ERROR: "G<A> is not a subtype of G<B>."
fun1(gb); // ok
fun1(gc); // ERROR: "G<C> is not a subtype of G<B>."
```

```
function fun1(p: G<B>) {
    let b: B = p.take(); // we know we get a B
    p.put(new B()); // we're allowed to put in a B
}
function fun2(p: G<? extends B>) {
    let b: B = p.take(); // we know we get a B
    p.put(new B()); // ERROR: "B is not a subtype of
          ? extends B."
}
function fun3(p: G<? super B>) {
    let b: B = p.take(); // ERROR: "? super B is not
          a subtype of B."
    p.put(new B()); // we're allowed to put in a B
}
```

Figure 4. Three functions illustrating the use of different wildcards.

In the first case, we get an error because the $G\langle A\rangle$ we pass in might contain an instance of A. The second invocation is accepted, of course. The third case, however, often leads to confusion: why are we not allowed to pass in a $G\langle C\rangle$, since all it may contain is an instance of C which is a subclass of B, so `fun1` would be ok with that argument? A glance at the body of `fun1` shows that this would be invalid, because `fun1` is, of course, allowed to invoke method `put()` of G to store an instance of B in G. If passing in an instance gc of $G\langle C\rangle$ were allowed, we would end up with a B being stored in gc after invoking `fun1(gc)`, breaking the contract of G.

Similarly, when invoking `fun2` and `fun3`, we notice that in each case one of the two errors we got in the previous listing will disappear:

```
fun2(ga); // ERROR: "G<A> is not a subtype of G<?
    extends B>."
fun2(gb); // ok
fun2(gc); // ok, G<C> is a subtype of G<? extends B>

fun3(ga); // ok, G<A> is a subtype of G<? super B>
fun3(gb); // ok
fun3(gc); // ERROR: "G<C> is not a subtype of G<?
    super B>."
```

By using a wildcard with an upper bound of B in the signature of `fun2`, we have effectively made G covariant in T, meaning

$$C <: B \Rightarrow G\langle C\rangle <: G\langle ?\ \texttt{extends}\ B\rangle \qquad (5)$$

Checking the body of `fun2`, we see that due to the wildcard in its signature, `fun2` is no longer able to invoke method `put()` of G on its argument `p` and put in a B. Precisely speaking, `fun2` would be allowed to call this method, but only with a value that is a subtype of the unknown type `? extends B`, which is never the case except for values that are a subtype of *all* types. In N4JS this is only the case for the special values `undefined` and `null` (similar to Java's `null`); hence, `fun2` would be allowed to clear the element stored in `p` by calling `p.put(undefined)`.

Accordingly, the above three invocations of `fun3` show that by using a wildcard with a lower bound of B in the signature of `fun3`, we can effectively make G contravariant in T and can thus invoke `fun3` with an instance of $G\langle A\rangle$ (but no longer with an instance of $G\langle C\rangle$, as was the case with fun2). Consequently, while fun3 is now allowed to put an instance of B into p, it can no longer assume getting back a B when calling method take() on p.

Using an unbounded wildcard in the signature of fun1 would leave us, in its body, with a combination of both restrictions we faced in fun2 and fun3, but would make all of the three invocations valid, i.e., both of the errors shown for the invocations of fun1 would disappear.

2) Definition-Site Variance: Many more recent programming languages did not take up the concept of wildcards as introduced by Java, but instead opted for a technique of declaring variance on the definition-site, e.g., C#, Scala.

In N4JS this is also possible, using the keywords out and in when declaring the type parameter of a generic class or interface. As an example, let us create two variations of type *G* introduced above (beginning of Section V-F), first starting with a covariant type *GR*:

```
class GR<out T> {
    private elem: T;
// ERROR "Cannot use covariant (out) type variable
    at contravariant position."
//  put(elem: T) { this.elem = elem; }
    take(): T { return this.elem; }
}
```

We have prefixed the declaration of type parameter T with keyword out, thus declaring *GR* to be covariant in T. Trying to define the exact same members as in *G*, we get an error for method put(), disallowing the use of covariant T as the type of a method parameter. Without going into full detail, we can see that just those cases that had been disallowed in the body of function fun2 (i.e., when using use-site covariance) are now disallowed already within the declaration of *GR*.

Given a modified version of fun1, using the above $GR\langle T\rangle$ as the type of its parameter, defined as

```
function funR(p: GR<B>) {
    let b: B = p.take(); // we know we get a B
//  p.put(new B()); // ERROR "No such member: put."
}
```

and helper variables

```
let gra: GR<A>;
let grb: GR<B>;
let grc: GR<C>;
```

we can invoke funR as follows:

```
funR(gra); // ERROR "GR<A> is not a subtype of GR<B
    >."
funR(grb);
funR(grc);
```

Note how having an error in the first and none in the last case corresponds exactly to what we saw above for use-site covariance through wildcards with upper bounds.

For completeness, let us see what a contravariant version of *G* would look like:

```
class GW<in T> {
    private elem: T;
    put(elem: T) { this.elem = elem; }
// ERROR "Cannot use contravariant (in) type
    variable at covariant position."
//  take(): T { return this.elem; }
}
```

Now, using T as the return type of a method is disallowed, meaning we cannot include method take().

A comparison of *GR* and *GW* shows that in the first case methods with an information flow leading into the class are disallowed while methods reading information from the type are allowed, and vice versa in the second case. Therefore, read-only classes and interfaces are usually covariant, whereas write-only classes and interfaces are usually contravariant (hence the “R” and “W” in the names of types *GR*, *GW*).

3) Comparison: Use-site variance is more flexible, because with the concept of wildcards any type can be used in a covariant or contravariant way if some functionality (e.g., our example functions above) is using instances purely for purposes that do not conflict with the assumptions of co-/contravariance, for example, only reading from a mutable collection (covariance), or only computing its size or only reordering its elements (co- and contravariance). And this is possible even if the implementor of the type in question did not prepare this before-hand.

On the other hand, if a particular type can only ever be used in, for example, a covariant way, e.g., a read-only collection type, declaring this variance on definition-site has the benefit that implementors of functions and methods using this type do not have to take care of the extra declaration of wildcards.

G. Conclusion

We would like to conclude this section by highlighting that we here do not aim to make claims as to whether structural or nominal typing or their combination is ultimately preferable, nor as to whether use- or definition-site variance or its combination is preferable on a general level. This would require an extensive analysis and empirical study, which is outside the scope of this article. We provided the above brief discussions of advantages and disadvantages merely for the sake of understandability of the respective language features. Also, full introduction to N4JS, its syntax and semantics, is not intended.

Our main goal for this brief overview of N4JS and its main typing-related features is to illustrate that we have used Xsemantics to implement a fearure-rich, real-world programming language that requires a comprehensive, complex type system.

VI. Case Study

In this section we will describe our real-world case study: the Xsemantics implementation of the type system of N4JS, a JavaScript dialect with a full-featured static type system (described in Section V). We will also describe some performance benchmarks related to the type system and draw some evaluations.

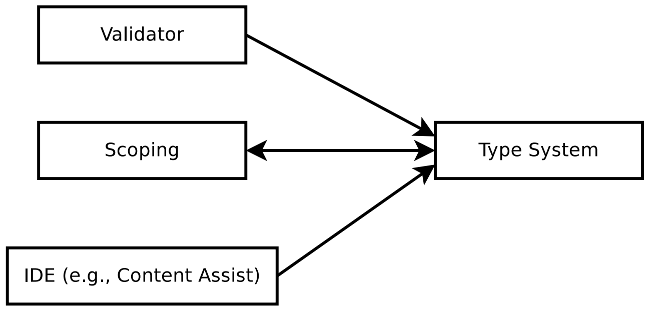


Figure 5. Interactions among the modules of the N4JS implementation.

A. Type System

The Xsemantics-based type system is not only used for validating the source code and detecting compile-time errors, but also for implementing scoping (see Section III), e.g., in order to find the actually referenced member in case of member access expressions:

```
class A {
    public method() {}
}
class B {
    public method() {}
}

let x: B;
x.method();
```

To know which method we are referring to in the last line, the one in class `A` or the one in class `B`, we need to first infer the type of `x`, which is a simple variable in this case, but, in general, it could be an arbitrary expression according to N4JS' syntax.

The relationship and interactions of the different modules of the N4JS implementation can be depicted as in Figure 5. Note that the interaction between scoping and the type system is bidirectional since, during the type inference some symbols may have to be resolved, and for symbol resolution, type inference is needed. Of course, the implementation of the type system takes care of avoiding possible cycles and loops.

The core parts of the N4JS type system are modeled by means of nine Xsemantics judgments, which are declared in Xsemantics as shown in Figure 6. The judgments have the following purpose:

- "type" is used to infer the type of any typable AST node.
- "subtype" tells if one given type is a subtype of another.
- "supertype" and "equaltype" are mostly delegating to "subtype" and are only required for convenience and improved error reporting.
- "expectedTypeIn" is used to infer a type expectation for an expression in a given context (the container).
- "upperBound" and "lowerBound" compute the lower and upper type bound, respectively. For example, given a wildcard `? extends C` (with *C* being a class) the "upperBound" judgment will return `C` and for wildcard `? super C` it will return the top type, i.e., `any`.
- "substTypeVariables" will replace all type variables referred to in a given type reference by a replacement type. The mapping from type variables to replacements (or bindings, substitutions) is defined in the rule environment.
- "thisTypeRef" is a special judgment for the so-called `this`-type of N4JS, which is not covered in detail, here.

This set of judgments does not only reflect the specific requirements of N4JS but arguably provides a good overview of what an Xsemantics-based type system implementation of any comprehensive Object-Oriented programming language would need.

These judgments are implemented by approximately 30 axioms and 80 rules. Since, with Xsemantics, type inference rules can often be implemented as a 1:1 correspondence to inference rules from a given formal specification, many rules are simple adaptations of rules given in the papers cited in Section V. For example, the subtype relation for union and intersection types is implemented with the rules shown in Figure 7. Note that we use many Xbase features, e.g., lambda expressions and extension methods (described in Section III-A).

In the implementation of the N4JS type system in Xsemantics we made heavy use of the rule environment. We are using it not only to pass contextual information and configuration to the rules, but also to store basic types that have to be globally available to all the rules of the type system (e.g., boolean, integer, etc.). This way, we can safely make the assumption that such type instances are singletons in our type system, and can be compared using the Java object identity. Another important use of the rule environment, as briefly mentioned above when introducing judgment "substTypeVariables", is to store type variable mappings and to pass this information from one rule to another. Finally, the rule environment is the key mechanism for guarding against infinite recursion in case of invalid source code such as cyclicly defined inheritance hierarchies.

To make the type system more readable, we implemented some static methods in a separate Java class `RuleEnvironmentExtensions`, and imported such methods as extension methods in the Xsemantics system:

```
import static extension RuleEnvironmentExtensions.*
```

These methods are used to easily access global type instances from the rule environment, as it is shown, for example, in the rule of Figure 8.

Other examples are shown in Figures 9 and 10. In particular, these examples also show how Xsemantics rules are close to the formal specifications. We believe they are also easy to read and thus to maintain.

Since the type system of N4JS is quite involved, creating useful and informative error messages is crucial to make the

```
judgments {

  type |− TypableElement element : output TypeRef
    error "cannot type " + element?.eClass?.name + " " + stringRep(element)
    source element

  subtype |− TypeArgument left <: TypeArgument right
    error stringRep(left) + " is not a subtype of " + stringRep(right)

  supertype |− TypeArgument left :> TypeArgument right
    error stringRep(left) + " is not a super type of " + stringRep(right)

  equaltype |− TypeArgument left ∼∼ TypeArgument right
    error stringRep(left) + " is not equal to " + stringRep(right)

  expectedTypeIn |− EObject container |> Expression expression : output TypeRef

  upperBound |∼ TypeArgument typeArgument /\ output TypeRef

  lowerBound |∼ TypeArgument typeArgument \/ output TypeRef

  substTypeVariables |− TypeArgument typeArg ∼> output TypeArgument

  thisTypeRef |∼ EObject location ∼> output TypeRef
}
```

Figure 6. Declarations of Xsemantics judgments from the N4JS type system.

```
rule subtypeUnion_Left
  G |− UnionTypeExpression U <: TypeRef S
from {
  U.typeRefs.forall[T|
    G |− T <: S
  ]
}

rule subtypeUnion_Right
  G |− TypeRef S <: UnionTypeExpression U
from {
  U.typeRefs.exists[T|
    G |− S <: T
  ]
}

rule subtypeIntersection_Left
  G |− IntersectionTypeExpression I <: TypeRef S
from {
  I.typeRefs.exists[T|
    G |− T <: S
  ]
}

rule subtypeIntersection_Right
  G |− TypeRef S <: IntersectionTypeExpression I
from {
  I.typeRefs.forall[T|
    G |− S <: T
  ]
}
```

Figure 7. N4JS union and intersection types implemented with Xsemantics.

```
rule typeUnaryExpression
  G |− UnaryExpression e: TypeRef T
from {
  switch (e.op) {
    case UnaryOperator.DELETE: T= G.booleanTypeRef()
    case UnaryOperator.VOID: T= G.undefinedTypeRef()
    case UnaryOperator.TYPEOF: T= G.stringTypeRef()
    case UnaryOperator.NOT: T= G.booleanTypeRef()
    default: // INC, DEC, POS, NEG, INV
      T = G.numberTypeRef()
  }
}
```

Figure 8. Typing of unary expression.

```
rule typeConditionalExpression
  G |− ConditionalExpression expr : TypeRef T
from {
  G |− expr.trueExpression : var TypeRef left
  G |− expr.falseExpression : var TypeRef right
  T = G.createUnionType(left, right)
}
```

Figure 9. Typing of conditional expression.

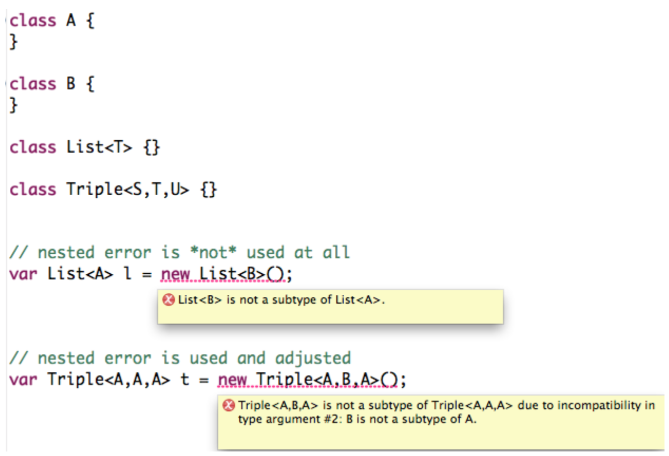
language usable, especially in the IDE. We have 3 main levels of error messages in the implementation:

1) default error messages defined on judgment declaration,

```
rule typeArrayLiteral
  G |− ArrayLiteral al : TypeRef T
from {
  val elementTypes = al.elements.map[
    elem |
    G |− elem : var TypeRef elementType;
    elementType;
  ]

  T = G.arrayType.createTypeRef(
      G.createUnionType(elementTypes))
}
```

Figure 10. Typing of array literal expression.



```
class A {
}

class B {
}

class List<T> {}

class Triple<S,T,U> {}


// nested error is *not* used at all
var List<A> l = new List<B>();
```

List<B> is not a subtype of List<A>.

```
// nested error is used and adjusted
var Triple<A,A,A> t = new Triple<A,B,A>();
```

Triple<A,B,A> is not a subtype of Triple<A,A,A> due to incompatibility in type argument #2: B is not a subtype of A.

Figure 11. The N4JS IDE and error reporting.

2) custom error messages using `fail`,
3) customized error messages due to failed nested judgments using `previousFailure` (described in Section IV-F).

Custom error messages are important especially when checking subtyping relations. For example, consider checking something like `A<string> <: A<number>`. The declared types are identical (i.e., `A`), so the type arguments have to be checked. If we did not catch and change the error message produced by the nested subtype checks `string <: number` and `number <: string`, then the error message would be very confusing for the user, because it only refers to the type arguments. In cases where the type arguments are explicitly given, this might be rather obvious, but that is not the case when the type arguments are only defined through type variable bindings or can change due to considering the upper/lower bound. Some examples of error messages due to subtyping are shown in Figure 11.

Figure 12 shows an excerpt of the subtype rule for parameterized type references, in order to illustrate how such composed error messages can be implemented. The excerpt shows the part of the rule that checks the type arguments given on left and right side for compatibility. If one of these subtype checks fails, it creates an error message composed from the original error message of the failed nested subtype check (obtained via special property "previousFailure") and an additional explanation including the index of the incompatible type argument. Note that such a composed error message is only created in certain cases, in this example only if there are at least two type arguments. Otherwise the default error message of judgment "subtype" (Figure 6) is being issued automatically by using the keyword `fail`.

The Xsemantics code in Figure 12 also shows that whenever some more involved special handling is required and the special, declarative-style syntax provided by Xsemantics is not suitable, all ordinary, imperative programming language constructs provided by Xbase can be integrated seamlessly into an Xsemantics rule.

B. *Performance*

N4JS is used to develop large scale ECMAScript applications. For this purpose, N4JS comes with a compiler, performing all validations and eventually transpiling the code to plain ECMAScript. We have implemented a test suite in order to measure the performance of the type system. Since we want to be able to measure the effect on performance of specific constructs, we use synthetic tests with configured scenarios. In spite of being artifical, these scenarios mimic typical situations in Javascript programming. There are several constructs and features that are performance critical, as they require a lot of type inference (which means a lot of rules are to be called). We want to discuss three scenarios in detail, Figure 13 summarizes the important code snippets used in these scenarios.

Function Expression: Although it is possible to specify the types of the formal parameters and the return type of functions, this is very inconvenient for function expressions. The function definition `f` (Figure 13) is called in the lines below the definition. Function `f` takes a function as argument, which itself requires a parameter of type `C` and returns an `A` element. Both calls (below the definition) use function expressions. The first call uses a fully typed function expression, while the second one relies on type inference. *Generic Method Calls*: As in Java, it is possible to explicitly specify type arguments in a call of a generic function. Similar to type expressions, it is more convenient to let the type system infer the type arguments, which actually is a typical constraint resolution problem. The generic function `g` (Figure 13) is called one time with explicitly specified type argument, and one time without type arguments. *Variable Declarations*: The type of a variable can either be explicitly declared, or it is inferred from the type of the expression used in an assignment. This scenario demonstrates why caching is so important: without caching, the type of `x1` would be inferred three times. Of course, this is not the case if the type of the variable is declared explicitly.

Table I shows some performance measurements, using the described scenarios to set up larger tests. That is, test files are generated with 250 or more usages of function expressions, or with up to 200 variables initialized following the pattern described above. In all cases, we run the tests with and without caching enabled. Also, for all scenarios we used two variants: with and without declared types. We measure the time required to execute the JUnit tests.

There are several conclusions, which could be drawn from the measurement results. First of all, caching is only worth in

```
rule subtypeParameterizedTypeRef
  G |− ParameterizedTypeRef left <: ParameterizedTypeRef right
from {
  // ...
  or
  {
    left.declaredType == right.declaredType
    // so, we have a situation like A<X> <: B<Y> with A==B,
    // continue checking X, Y for compatibility ...

    val len = Math.min(Math.min(left.typeArgs.size, right.typeArgs.size), right.declaredType.typeVars.size);
    for(var i=0;i<len;i++) {

      val leftArg = left.typeArgs.get(i)
      val rightArg = right.typeArgs.get(i)
      val variance = right.declaredType.getVarianceOfTypeVar(i)

      G |~ leftArg /\ var TypeRef leftArgUpper
      G |~ leftArg \/ var TypeRef leftArgLower
      G |~ rightArg /\ var TypeRef rightArgUpper
      G |~ rightArg \/ var TypeRef rightArgLower

      {
        // require leftArg <: rightArg, except we have contravariance
        if(variance!==Variance.CONTRA) {
          G2 |− leftArgUpper <: rightArgUpper
        }
        // require rightArg <: leftArg, except we have covariance
        if(variance!==Variance.CO) {
          G2 |− rightArgLower <: leftArgLower
        }
      }
      or
      {
        if(len>1 && previousFailure.isOrCausedByPriorityError) {
          fail error stringRep(left) + " is not a subtype of " + stringRep(right)
               + " due to incompatibility in type argument #" + (i+1) + ": "
               + previousFailure.compileMessage
            data PRIORITY_ERROR
        } else {
          fail // with default message
        }
      }
    }
  }
  or
  // ...
}
```

Figure 12. Implementing advanced, composed error messages in Xsemantics.

some cases, but these cases can make all the difference. The first two scenarios do not gain much from caching, actually the overhead for managing the cache even slightly decreases performance in case of generic methods calls. In many cases, types are to be computed only once. In our example, the types of the type arguments in the method call are only used for that particular call. Thus, caching the arguments there does not make any sense. Things are different for variable declarations. As described above, caching the type of a variable, which is used many times, makes a lot of sense. Increasing the performance by the factor of more than 100 is not only about speeding up the system a little bit—it is about making it work at all for larger programs. Even if all types are declared, type inference is still required in order to ensure that the inferred type is compatible with the declared type. This is why in some cases the fully typed scenario is even slower than the scenario which uses only inferred types. While in some cases (scenario 1 and 3) the performance increases linearly with the size, this is not true for scenario 2, the generic method call. This demonstrates a general problem with interpreting absolute performance measurements: it is very hard to pinpoint the exact location in case of performance problems, as many parts, such as the parser, the scoping system and the type system are involved. Therefore, we concentrate on relative

```
// Scenario 1: function expression
function f ({function (C): A} func) { ... };
// typed
f( function (C p): A { return p.getA() || new A(); }
    )
// inferred
f( function (p) { return p.getA() || new A(); } )

// Scenario 2: generic method call
function <T> g (T p): T { ... }
// typed
var s1 = <string>g("");
// inferred
var s2 = g("");

// Scenario 3: variable declarations and references
// typed
var number y1 = 1;
var number y2 = y1; ...
// inferred
var x1 = 1;
var x2 = x1; var x3 = x2; ...
```

Figure 13. Scenario snippets used in performance tests

TABLE I. Performance measurements (runtime in ms)

Scenario	**without caching**		**with caching**	
size	**typed**	**inferred**	**typed**	**inferred**
Function Expressions				
250	875	865	772	804
500	1,860	1,797	1,608	1,676
1000	4,046	3,993	3,106	3,222
2000	9,252	9,544	8,143	8,204
Generic Method Calls				
250	219	273	223	280
500	566	644	548	654
1000	1,570	1,751	1,935	1,703
2000	6,143	6,436	6,146	6,427
Variable Declarations				
50	19	580	18	39
100	27	3,848	26	102
200	44	31,143	36	252

performance between slightly modified versions of the type system implementation (while leaving all other subsystems unchanged).

We observe that it is not feasible to compare, on a more global level, the overall performance of N4JS to other languages implemented with more traditional approaches (without the use of Xsemantics), because there are too many factors that should be taken into consideration, starting from the complexity of the type system and its type inference up to the specific programming language and frameworks used for their compiler's implementation.

Summarizing, we learned that different scenarios must be taken into account when working on performance optimization, in order to make the right decision about whether using caching or not. Surely, when type information is reused in other parts of the program over and over again, like in the variable scenario, caching optimization is crucial. Combining the type system with control flow analysis, leading to effect systems, may make caching dispensable in many cases. Further investigation in this direction is ongoing work.

VII. Conclusion

In this paper, we presented the implementation in Xsemantics of the type system of N4JS, a statically typed JavaScript, with powerful type inference mechanisms, focusing both on the performance of the type system and on its integration in the Eclipse IDE. The N4JS case study proved that Xsemantics is mature and powerful enough to implement a complex type system of a real-world language, where types do not need to be declared, thus requiring involved type inference mechanisms.

Thanks to Xtext, Xsemantics offers a rich Eclipse tooling; in particular, thanks to Xbase, Xsemantics is also completely integrated with Java. For example, from the Xsemantics editor we can navigate to Java types and Java method definitions, see Java type hierarchies, and other features that are present in the Eclipse Java editor (see, e.g., Figure 14). This also holds the other way round: from Java code that uses code generated from a Xsemantics definition we can navigate directly to the original Xsemantics method definition.

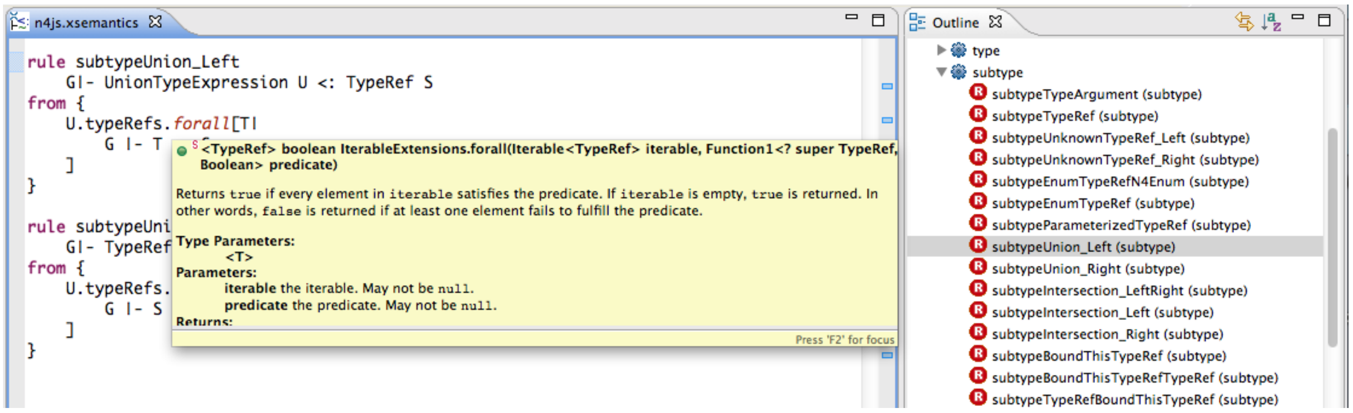
Most importantly, the Xsemantics IDE allows the developer to debug the original Xsemantics system source code, besides the generated Java code. Figure 15 shows a debug session of the N4JS type system: we have set a break point in the Xsemantics file, and when the program hits Xsemantics generated Java code the debugger automatically switches to the original Xsemantics code (note the file names in the thread stack, the "Breakpoint" view and the "Variables" view).

With respect to manual implementations of type systems in Java, Xsemantics specifications are more compact and closer to formal systems. We also refer to [67] for a wider discussion about the importance of having a DSL for type systems in language frameworks. In particular, Xsemantics integration with Java allows the developers to incrementally migrate existing type systems implemented in Java to Xsemantics [68].

Xsemantics has been developed with *Test Driven Development* technologies, with almost 100% code coverage, using *Continuous Integration* systems and code quality tools, such as *SonarQube* (a report can be found at `http://www.lorenzobettini.it/2014/09/dealing-with-technical-debt-with-sonarqube-a-case-study-with-xsemantics`).

Acknowledgment

We want to thank Sebastian Zarnekow, Jörg Reichert and the colleagues at NumberFour, in particular Jakub Siberski and Torsten Krämer, for feedback and for implementing N4JS with us.



```
rule subtypeUnion_Left
    G|- UnionTypeExpression U <: TypeRef S
from {
    U.typeRefs.forall[T|
        G |- T
    ]
}

rule subtypeUni
    G|- TypeRef
from {
    U.typeRefs.
        G |- S
    ]
}
```

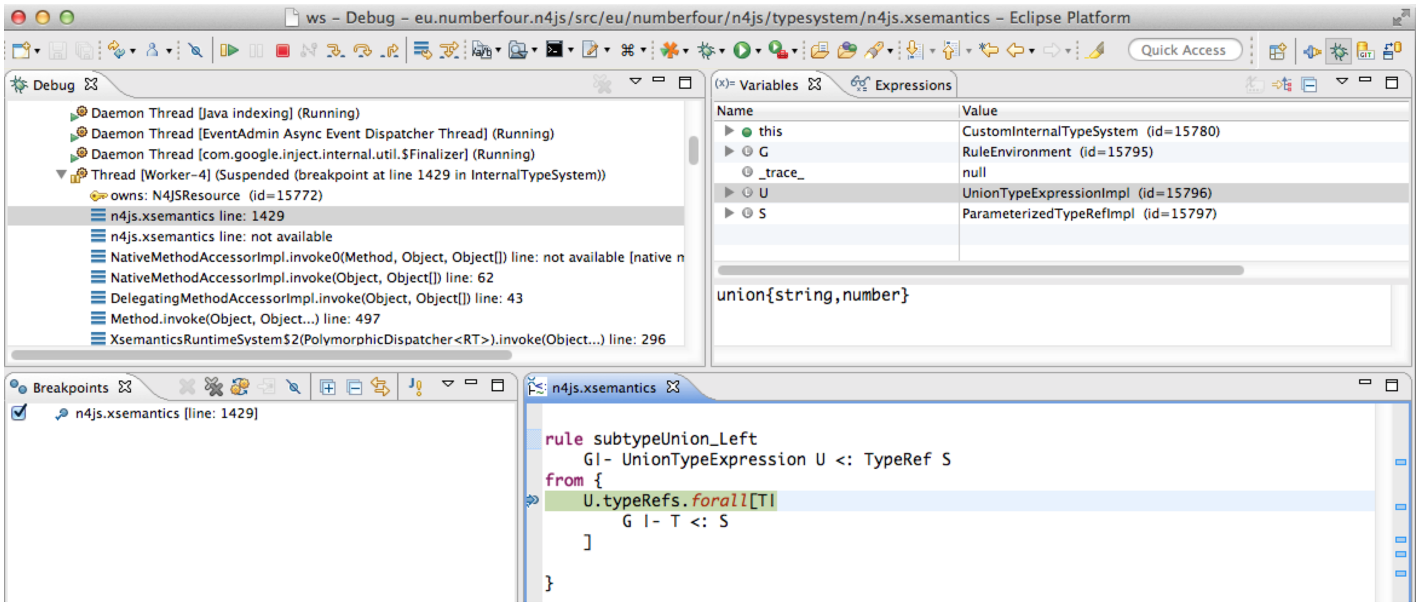
<TypeRef> boolean IterableExtensions.forall(Iterable<TypeRef> iterable, Function1<? super TypeRef, Boolean> predicate)

Returns true if every element in iterable satisfies the predicate. If iterable is empty, true is returned. In other words, false is returned if at least one element fails to fulfill the predicate.

Type Parameters:
<T>
Parameters:
iterable the iterable. May not be null.
predicate the predicate. May not be null.
Returns:

subtypeTypeArgument (subtype)
subtypeTypeRef (subtype)
subtypeUnknownTypeRef_Left (subtype)
subtypeUnknownTypeRef_Right (subtype)
subtypeEnumTypeRefN4Enum (subtype)
subtypeEnumTypeRef (subtype)
subtypeParameterizedTypeRef (subtype)
subtypeUnion_Left (subtype)
subtypeUnion_Right (subtype)
subtypeIntersection_LeftRight (subtype)
subtypeIntersection_Left (subtype)
subtypeIntersection_Right (subtype)
subtypeBoundThisTypeRef (subtype)
subtypeBoundThisTypeRefTypeRef (subtype)
subtypeTypeRefBoundThisTypeRef (subtype)

Figure 14. Accessing Java types from Xsemantics source code.


ws – Debug – eu.numberfour.n4js/src/eu/numberfour/n4js/typesystem/n4js.xsemantics – Eclipse Platform
Quick Access
Debug
Variables
Expressions
Breakpoints
n4js.xsemantics

Daemon Thread [Java indexing] (Running)
Daemon Thread [EventAdmin Async Event Dispatcher Thread] (Running)
Daemon Thread [com.google.inject.internal.util.$Finalizer] (Running)
Thread [Worker-4] (Suspended (breakpoint at line 1429 in InternalTypeSystem))
owns: N4JSResource (id=15772)
n4js.xsemantics line: 1429
n4js.xsemantics line: not available
NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native m
NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
Method.invoke(Object, Object...) line: 497
XsemanticsRuntimeSystem$2(PolymorphicDispatcher<RT>).invoke(Object...) line: 296

Name	Value
this	CustomInternalTypeSystem (id=15780)
G	RuleEnvironment (id=15795)
trace	null
U	UnionTypeExpressionImpl (id=15796)
S	ParameterizedTypeRefImpl (id=15797)

```
union{string,number}
```

n4js.xsemantics [line: 1429]

```
rule subtypeUnion_Left
    G|- UnionTypeExpression U <: TypeRef S
from {
    U.typeRefs.forall[T|
        G |- T <: S
    ]
}
```

Figure 15. Debugging Xsemantics source code: a breakpoint was set in rule `subtypeUnion_Left` inside the Xsemantics editor (bottom right), stack trace and local variables are shown on the top left and right, respectively.

References

[1] L. Bettini, J. von Pilgrim, and M.-O. Reiser, "Implementing the Type System for a Typed Javascript and its IDE," in COMPUTATION TOOLS. IARIA, 2016, pp. 6–11.

[2] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2003.

[3] K. Beck, Test Driven Development: By Example. Addison-Wesley, 2003.

[4] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in SPLASH/OOPSLA Companion. ACM, 2010, pp. 307–309.

[5] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, 2nd ed. Packt Publishing, 2016.

[6] ——, "Implementing Java-like languages in Xtext with Xsemantics," in OOPS (SAC). ACM, 2013, pp. 1559–1564.

[7] L. Cardelli, "Type Systems," ACM Computing Surveys, vol. 28, no. 1, 1996, pp. 263–264.

[8] B. C. Pierce, Types and Programming Languages. Cambridge, MA: The MIT Press, 2002.

[9] L. Bettini, "Implementing Type Systems for the IDE with Xsemantics," Journal of Logical and Algebraic Methods in Programming, vol. 85, no. 5, Part 1, 2016, pp. 655 – 680.

[10] ——, "A DSL for Writing Type Systems for Xtext Languages," in PPPJ. ACM, 2011, pp. 31–40.

[11] M. Voelter, "Xtext/TS - A Typesystem Framework for Xtext," 2011.

[12] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modeling with UML. Addison Wesley, 1999.

[13] Object Management Group, "Object Constraint Language, Version 2.2," Omg document number: formal/2010-02-01 edition, 2010, http://www.omg.org/spec/OCL/2.2, Accessed: 2016-01-07.

[14] E. Vacchi and W. Cazzola, "Neverlang: A Framework for Feature-Oriented Language Development," Computer Languages, Systems & Structures, vol. 43, no. 3, 2015, pp. 1–40.

[15] W. Cazzola and E. Vacchi, "Neverlang 2: Componentised Language Development for the JVM," in Software Composition, ser. LNCS, vol. 8088. Springer, 2013, pp. 17–32.

[16] T. Ekman and G. Hedin, "The JastAdd system – modular extensible compiler construction," Science of Computer Programming, vol. 69, no. 1-3, 2007, pp. 14 – 26.

[17] L. Diekmann and L. Tratt, "Eco: A Language Composition Editor," in SLE, ser. LNCS, vol. 8706. Springer, 2014, pp. 82–101.

[18] E. Barrett, C. F. Bolz, L. Diekmann, and L. Tratt, "Fine-grained Language Composition: A Case Study," in ECOOP, ser. LIPIcs, vol. 56. Dagstuhl LIPIcs, 2016, pp. 3:1–3:27.

[19] L. C. L. Kats and E. Visser, "The Spoofax language workbench. Rules for declarative specification of languages and IDEs," in OOPSLA. ACM, 2010, pp. 444–463.

[20] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An Extensible Compiler Framework for Java," in Compiler Construction, ser. LNCS, vol. 2622. Springer, 2003, pp. 138–152.

[21] G. Bracha, "Pluggable Type Systems," in Workshop on Revival of Dynamic Languages, 2004.

[22] M. Voelter et al, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, 2013.

[23] M. Pfeiffer and J. Pichler, "A comparison of tool support for textual domain-specific languages," in Proc. DSM, 2008, pp. 1–7.

[24] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," Computer Languages, Systems & Structures, vol. 44, Part A, 2015, pp. 24–47.

[25] P. Charles, R. Fuhrer, S. Sutton Jr., E. Duesterwald, and J. Vinju, "Accelerating the creation of customized, language-Specific IDEs in Eclipse," in OOPSLA. ACM, 2009, pp. 191–206.

[26] F. Jouault, J. Bézivin, and I. Kurtev, "TCS: a DSL for the specification of textual concrete syntaxes in model engineering," in GPCE. ACM, 2006, pp. 249–254.

[27] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and Refinement of Textual Syntax for Models," in ECMDA-FA, ser. LNCS, vol. 5562. Springer, 2009, pp. 114–129.

[28] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," Science of Computer Programming, vol. 72, no. 1–2, 2008, pp. 52–70.

[29] E. Visser et al, "A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs," in Onward! ACM, 2014, pp. 95–111.

[30] G. Konat, L. Kats, G. Wachsmuth, and E. Visser, "Declarative Name Binding and Scope Rules," in SLE, ser. LNCS, vol. 7745. Springer, 2012, pp. 311–331.

[31] V. A. Vergu, P. Neron, and E. Visser, "DynSem: A DSL for Dynamic Semantics Specification," in RTA, ser. LIPIcs, vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378.

[32] H. Xu, "EriLex: An Embedded Domain Specific Language Generator," in TOOLS, ser. LNCS, vol. 6141. Springer, 2010, pp. 192–212.

[33] P. Borras et al, "CENTAUR: the system," in Software Engineering Symposium on Practical Software Development Environments, ser. SIGPLAN. ACM, 1988, vol. 24, no. 2, pp. 14–24.

[34] M. Fowler, "A Language Workbench in Action - MPS," `http://martinfowler.com/articles/mpsAgree.html`, 2008, accessed: 2016-01-07.

[35] M. G. J. V. D. Brand, J. Heering, P. Klint, and P. A. Olivier, "Compiling language definitions: the ASF+SDF compiler," ACM TOPLAS, vol. 24, no. 4, 2002, pp. 334–368.

[36] A. Dijkstra and S. D. Swierstra, "Ruler: Programming Type Rules," in FLOPS, ser. LNCS, vol. 3945. Springer, 2006, pp. 30–46.

[37] M. Felleisen, R. B. Findler, and M. Flatt, Semantics Engineering with PLT Redex. The MIT Press, 2009.

[38] T. Reps and T. Teitelbaum, "The Synthesizer Generator," in Software Engineering Symposium on Practical Software Development Environments. ACM, 1984, pp. 42–48.

[39] G. Kahn, B. Lang, B. Melese, and E. Morcos, "Metal: A formalism to specify formalisms," Science of Computer Programming, vol. 3, no. 2, 1983, pp. 151–188.

[40] E. Morcos-Chounet and A. Conchon, "PPML: A general formalism to specify prettyprinting," in IFIP Congress, 1986, pp. 583–590.

[41] T. Despeyroux, "Typol: a formalism to implement natural semantics," INRIA, Tech. Rep. 94, Mar. 1988.

[42] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," in ICSR. IEEE, 1998, pp. 143–153.

[43] M. Bravenboer, R. de Groot, and E. Visser, "MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT," in GTTSE, ser. LNCS, vol. 4143. Springer, 2006, pp. 297–311.

[44] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," STTT, vol. 12, no. 5, 2010, pp. 353–372.

[45] T. Clark, P. Sammut, and J. Willans, Superlanguages, Developing Languages and Applications with XMF, 1st ed. Ceteva, 2008.

[46] L. Renggli, M. Denker, and O. Nierstrasz, "Language Boxes: Bending the Host Language with Modular Language Changes," in SLE, ser. LNCS, vol. 5969. Springer, 2009, pp. 274–293.

[47] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa, "Ott: Effective tool support for the working semanticist," J. Funct. Program, vol. 20, no. 1, 2010, pp. 71–122.

[48] Y. Bertot and P. P. Castéran, Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions, ser. Texts in theoretical computer science. Springer, 2004.

[49] M. Gordon, "From LCF to HOL: a short history," in Proof, Language, and Interaction: Essays in Honour of Robin Milner. The MIT Press, 2000, pp. 169–186.

[50] L. C. Paulson, Isabelle: A Generic Theorem Prover, ser. LNCS. Springer, 1994, vol. 828.

[51] M. Voelter, "Language and IDE Modularization and Composition with MPS," in GTTSE, ser. LNCS, vol. 7680. Springer, 2011, pp. 383–430.

[52] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers, 2007.

[53] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework, 2nd ed. Addison-Wesley, 2008.

[54] D. R. Prasanna, Dependency Injection: Design Patterns Using Spring and Guice, 1st ed. Manning, 2009.

[55] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth, "A Theory of Name Resolution," in ESOP, ser. LNCS, vol. 9032. Springer, 2015, pp. 205–231.

[56] S. Efftinge et al, "Xbase: Implementing Domain-Specific Languages for Java," in GPCE. ACM, 2012, pp. 112–121.

[57] J. R. Hindley, Basic Simple Type Theory. Cambridge University Press, 1987.

[58] ECMA, "ECMAScript 2015 Language Specification," ISO/IEC, International Standard ECMA-262, 6th Edition, Jun. 2015, accessed: 2016-01-07. [Online]. Available: `http://www.ecma-international.org/publications/-files/ECMA-ST/Ecma-262.pdf`

[59] A. Hejlsberg and S. Lucco, TypeScript Language Specification, 1st ed., Microsoft, Apr. 2014.

[60] Dart Team, Dart Programming Language Specification, 1st ed., Mar. 2014.

[61] A. Igarashi and H. Nagira, "Union types for object-oriented programming," Journal of Object Technology, vol. 6, no. 2, 2007, pp. 47–68.

[62] N. Cameron, E. Ernst, and S. Drossopoulou, "Towards an Existential Types Model for Java Wildcards," in Formal Techniques for Java-like Programs (FTfJP), July 2007, pp. 1–17.

[63] G. Mezzetti, A. Møller, and F. Strocco, "Type Unsoundness in Practice: An Empirical Study of Dart," in DLS. ACM, 2016, pp. 13–24.

[64] B. Liskov, "Keynote address – data abstraction and hierarchy," ACM SIGPLAN Notices, vol. 23, no. 5, 1987, pp. 17–34.

[65] D. Malayeri and J. Aldrich, "Integrating Nominal and Structural Subtyping," in ECOOP, ser. LNCS, vol. 5142. Springer, 2008, pp. 260–284.

[66] A. Igarashi and M. Viroli, "On Variance-Based Subtyping for Parametric Types," in ECOOP, ser. LNCS, vol. 2374. Springer, 2002, pp. 441–469.

[67] L. Bettini, D. Stoll, M. Völter, and S. Colameo, "Approaches and Tools for Implementing Type Systems in Xtext," in SLE, ser. LNCS, vol. 7745. Springer, 2012, pp. 392–412.

[68] A. Heiduk and S. Skatulla, "From Spaghetti to Xsemantics - Practical experiences migrating typesystems for 12 languages," XtextCon, 2015.