Implementing the Type System for a Typed Javascript and its IDE

Lorenzo Bettini

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

Jens von Pilgrim, Mark-Oliver Reiser

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

***Abstract*—Implementing a programming language with IDE tooling features poses several challenges even when using language workbenches like Xtext that provides Eclipse integration. A complex type system with powerful type inference mechanisms requires focusing carefully on performance issues that might 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 concentrate on techniques to make the type system implementation of N4JS integrate efficiently with Eclipse. For the implementation of such a type system we use Xsemantics, a DSL for writing type systems, reduction rules and in general relation rules for languages implemented in Xtext. Xsemantics uses a syntax that resembles formal type system specifications, so that the implementation of formally defined type rules can be implemented easier and more directly than in Java.**

Keywords–DSL; Type System; Implementation; Eclipse.

I. Introduction

Xtext [1] is a popular Eclipse framework for the development of Domain-Specific Languages (DSLs) and their Integrated Development Environments (IDEs). The type system and 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. Besides functional aspects, implementing a complex type system with powerful type inference mechanisms poses several challenges due to performance issues. 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 [2], Xsemantics [3] was introduced. Xsemantics is a DSL for writing rules for languages implemented in Xtext, e.g., the type system, the operational semantics and the 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, while keeping the Java-like shape. This makes it usable both by formal theory people and by Java programmers. Xsemantics has improved a lot in order to make it usable for modern full-featured languages and real-world performance requirements. The new and advanced features of Xsemantics are presented in [4].

In this paper, we present the implementation in Xsemantics of the type system of N4JS, a version of JavaScript implemented with Xtext, with powerful type inference mechanisms (including Java-like generics). 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 is the first real-world example of the applicability of Xsemantics for a complex type system with involved type inference.

The paper is structured as follows. We provide a small introduction to Xtext and Xsemantics in Section II. In Section III, we present our main case study: the implementation of the type system of N4JS with Xsemantics, with some performance benchmarks related to the type system. Section IV concludes the paper and discusses some related works.

II. Xtext and Xsemantics

In this section, we will briefly recall the main features of Xtext and Xsemantics.

Xtext [1] is a *language workbench* and it deals not only with the compiler mechanisms but also with Eclipse-based tooling: Xtext generates the Eclipse editor for the language that we are implementing with syntax highlighting, background parsing with error markers, outline view and code completion. In the following we describe the two complementary mechanisms of Xtext that the programmer has to implement for the type checking. 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. 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.

A system definition in Xsemantics is a set of *judgments* (formally, assertions about the properties of programs) and a set of *rules* (formally, implications between judgments). Rules have a conclusion and a set of premises. Rules can act on any

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-466-4

```
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. Judgment definitions in Xsemantics.

Java object. Typically, rules act on the objects representing the Abstract Syntax Tree (AST). 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.

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). 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 which are useful for generating informative error information.

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 [5], [6]).

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 [7] to provide a rich Java-like syntax for defining rules. Xbase is a reusable expression language that integrates tightly with Java, its type system and Eclipse Java Development Tools (JDT). The syntax of Xbase is similar to Java with less "syntactic noise" and some advanced linguistic constructs. 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. Xbase also provides *lambda expressions*, which have the shape `[ param1, param2, ... | body ]`. Xbase's lambda expressions together with extension methods allow to easily write statements and expressions which are not only more readable than in Java, but they are also very close to formal specifications.

The premises of a rule, which are specified in a **from** block, can be any Xbase expression, or a *rule invocation*. 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. When another rule is invoked, upon return, the output arguments will have the values assigned in the invoked rule. 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. Note that, besides this strategy for selecting a specific rules, Xsemantics itself does not implement, neither it defines, any other strategy.

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. *Predicates* can be seen as a special form of auxiliary functions. In an Xsemantics system, we can specify some special rules, *checkrule*, 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. In an Xsemantics system, fields can be defined, which 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 makes it easier 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. Custom error information can be specified on judgments, rules and auxiliary functions. This can be used for providing useful error information. 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. Moreover, in *or* blocks, the implicit variable `previousFailure` is available. This allows us to build informative error messages as shown in Section III-B.

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. 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. The cached values will be automatically discarded as soon as the contents of the program changes.

A. *Related Work*

Xsemantics can be considered the successor of Xtypes [8]. With this respect, Xsemantics provides a much richer syntax for rules that can access any existing Java library. In Xsemantics, a system can extend an existing one (adding and overriding rules). However, these extensibility and compositionality features are not as powerful as the ones of other frameworks such as, e.g., [9], [10], [11].

There are other tools for implementing DSLs and IDE tooling (see [12], [13] for a wider comparison). Spoofax [10], another language workbench which targets Eclipse, relies on Stratego [14] for rule-based specifications. Xtext Type System (XTS) [15] 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. EriLex [16] supports specifying syntax, type rules, and dynamic semantics but no 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 [17]. Other approaches, such as, e.g., [11], [16], [18], [19], [20], [21], [22], instead require the programmer to use the framework also for defining the syntax of the language.

The importance of targeting IDE tooling was recognized also in older frameworks, such as *Synthesizer* [23] and *Centaur* [18] (the latter was using several formalisms [24], [25], [26]). Finally, 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, [20], [21], [22], [27], [28], [29], [30], [31].

III. CASE STUDY

In this section we will describe our real-world case study: the implementation of the type system for a JavaScript dialect with a full-featured static type system implemented with Xsemantics. We will also describe some performance benchmarks related to the type system and draw some evaluations.

A. *N4JS—Typed JavaScript*

We have used Xsemantics to implement the type system of a real-world language called N4JS. N4JS is a super set of JavaScript also known as ECMAScript with modules and classes as proposed in [32]. Most importantly N4JS adds a full-featured static type system on top of JavaScript, similar to TypeScript [33] or Dart [34]. N4JS is still under development, but it is already being used internally at NumberFour AG. Moreover, it will be made available as an open source project in the near future.

```
function f(A p) {
    var pNotNull = p || {name: "default", age: 42};
    ...
}
```

Figure 2. Typical usage of union types in N4JS

```
class A {
    f(union{B,C} p) {
        if (p instanceof B) { f_B(p) }
        else { f_C(p) }
    }
}
```

Figure 3. Union types used for emulated method overloading in 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 [35]. N4JS supports generics similar to Java or TypeScript, 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 [36]). The syntax of type expressions is similar to Java's type expressions as far as possible.

Union types are an important feature in typed ECMAScript-related languages. For example, logical operators do not return a single boolean value in ECMAScript. This is often used in JavaScript programs in order to avoid null checks as demonstrated in Figure 2. The type of `pNotNull` is to be inferred as the union type of `A` and the object literal with a property "name" of type `string` and a property "age" of type `number`. Union types can also be used as a technique to emulate method overloading, which is not directly supported by ECMAScript, as shown in Figure 3.

Functions are first-class citizens in the ECMAScript language, which is reflected by the notion of *function types* in N4JS. In combination with generic methods, function expressions and type inference, this becomes a convenient and powerful feature, as shown in Figure 4. Note that the method call requires a lot of type inference capabilities: firstly, the type variable `T` has to be substituted correctly with `A`; secondly, the signature of the function expression has to be inferred from the formal parameter's type, taking the type variable substitution into account.

In Figure 5, we show the N4JS Eclipse editor in action. Note that the type system correctly inferred the type of the invoked method and instantiated the type parameter according

```
function <T> exists(Array<T> list,
        {function(T p):boolean} predicate): boolean { ... }

function existsJohn (Array<A> list): boolean {
    return exists ( list ,
        function(p) { return p.name == "John" }
    );
}
```

Figure 4. Generic method call with function expression and type inference

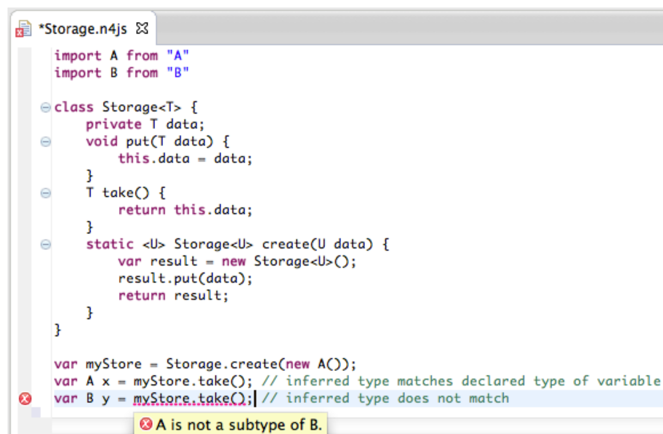

Figure 5. The N4JS editor and its type inference in action.

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

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

Figure 6. N4JS union types implemented with Xsemantics.

to the argument passed to the generic method `create`.

B. Type System

The Xsemantics based type system is not only used for validation purposes, but also for implementing the scoping (see Section II), e.g., in order to find the correct method in case of overridden methods. The whole type system of N4JS is modeled by means of Xsemantics judgments, implemented by approximately 30 axioms and 80 rules. Since type inference rules can be implemented almost 1:1 with Xsemantics, many rules are simple adaptations of rules described in the aforementioned papers. For example, the subtype relation for union types is implemented with the rules shown in Figure 6. Note that we use many Xbase features, e.g., lambda expressions and extension methods (described in Section II).

In the implementation of the N4JS type system in Xsemantics we made a heavy use of the rule environment. We are using it not only to pass contextual information 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 checked using the standard Java object equality. To make the type system more readable, we implemented some static methods in a separate Java class `RuleEnvironmentExtensions`, and we 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 7.

```
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 7. 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 8. Typing of conditional expression.

Other examples are shown in Figure 8 and 9. 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 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, 2) custom error messages using **fail**, 3) customized error messages due to failed nested judgments using `previousFailure` (described in Section II). 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 would 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 10.

C. 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

```
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 9. Typing of array literal expression.

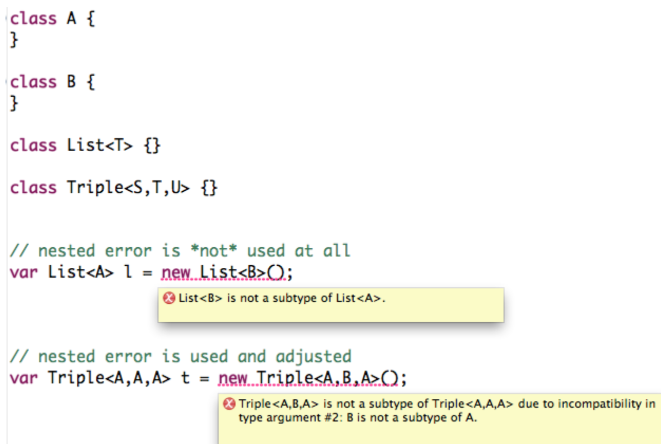

Figure 10. The N4JS IDE and error reporting.

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

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

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

Figure 11. Scenario snippets used in performance tests

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 which 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 11 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 11) 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 11) 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

TABLE I. PERFORMANCE MEASUREMENTS (RUNTIME IN MS)

Scenario size	**without caching** typed	inferred	**with caching** 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

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

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.

IV. Conclusions

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.

Thanks to Xtext, Xsemantics offers a rich Eclipse tooling, including the debugger for Xsemantics rule definitions. These features are extremely important for the effective usability of Xsemantics, especially in complex type systems like N4JS' one. With respect to manual implementations of type systems in Java, Xsemantics specifications are more compact and closer to formal systems. We also refer to [37] 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 [38].

Acknowledgment

The first author was partially supported by itemis Schweiz, MIUR (proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY. We also 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.

References

[1] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, 2013.

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

[3] ——, "Xsemantics," http://xsemantics.sf.net, 2016, accessed: 2016-01-07.

[4] ——, "Implementing Type Systems for the IDE with Xsemantics," Journal of Logical and Algebraic Methods in Programming, 2016, to Appear.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

[24] 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.

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

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

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

[28] 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.

[29] 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.

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

[31] 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.

[32] "Draft ECMAScript Language Specification," ISO/IEC, Working Draft ECMA-262, 6th Edition, Apr. 2014.

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

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

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

[36] 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.

[37] 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.

[38] A. Heiduk and S. Skatulla, "From Spaghetti to Xsemantics - Practical experiences migrating typesystems for 12 languages," XtextCon, http://xtextcon.org, 2015.