Journal of Logical and Algebraic Methods in Programming 85 (2016) 655–680

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

# Implementing type systems for the IDE with Xsemantics ☆

Lorenzo Bettini

*Dipartimento di Informatica, Università di Torino, Italy*

ARTICLE INFO

*Article history:*
Received 14 March 2015
Received in revised form 2 October 2015
Accepted 23 November 2015
Available online 27 November 2015

*Keywords:*
DSL
Type systems
Semantics
Implementation
Eclipse

ABSTRACT

Xsemantics is a DSL for writing type systems, reduction rules and, in general, relation rules for languages implemented in Xtext (Xtext is an Eclipse framework for rapidly building languages together with all the typical IDE tooling). Xsemantics aims at reducing the gap between the formalization of a language (i.e., type system and operational semantics) and the actual implementation in Xtext, since it uses a syntax that resembles the rules in a formal setting. In this paper we present the main features of Xsemantics for implementing type systems and reduction rules through examples (Featherweight Java and lambda calculus). We show how such implementations are close to the actual formalizations, and how Xsemantics can be a helpful tool when proving the type safety of a language. We also describe the new features of Xsemantics that help achieving a modular and efficient implementation of type systems. In particular, we focus on specific implementation techniques for implementing type systems that are suited for the IDE (in our context, Eclipse), in order to keep the tooling responsive and guarantee a good user experience.



## 1. Introduction

Xtext [2,7] is a popular Eclipse framework for the development of programming languages and domain-specific languages (DSLs). Starting from a grammar definition it generates a parser, an abstract syntax tree based on EMF [60], and an Eclipse-based editor, with all the typical IDE tooling. Xtext comes with good defaults for all the above artifacts and the language developer can easily customize them all. For all these reasons, Xtext makes language implementation and its integration into Eclipse really easy [27].

Checking that a program is semantically correct in Xtext consists of two mechanisms that have to be customized by the developer: *scoping* (Section 2.1), i.e., resolving and binding symbols (e.g., binding a reference to its definition in the program), and *validation* (Section 2.2), which includes all the other semantic checks that do not deal with scoping (e.g., checking that the type of the returned expression in a method body is conformant to the method's return type). In a statically typed language, these two mechanisms heavily rely on types. The type system and the interpreter/compiler for a language implemented in Xtext are usually implemented in Java. Instead, we want to employ a DSL also for these tasks, in order to implement, in a more direct way, a language that has been formalized with a type system and an operational semantics. Furthermore, this will allow us to have the typing and reduction rules in a compact form, similar to the actual formalization, easy to understand and maintain, and then have the corresponding Java code automatically generated. Finally, the gap between the formalization of a language and its actual implementation in Xtext will be reduced.

☆ Work partially supported by MIUR (2010LHT4KM, proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY.
*E-mail address:* bettini@di.unito.it.

http://dx.doi.org/10.1016/j.jlamp.2015.11.005
2352-2208/

For all the above reasons, we implemented Xsemantics,[1] 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). A system definition in Xsemantics is a set of rules that act on the elements of the Abstract Syntax Tree (AST) of a program. Xsemantics will generate Java code that can be used to implement the scoping and the validation for the language.

The very first prototype of Xsemantics was introduced in [9], where it was used, together with other frameworks, in an analysis of several approaches for implementing type systems for Xtext DSLs. Then, in [8], the first release of Xsemantics was presented. At that time, Xsemantics was still a proof of concept for studying prototype language implementations starting from existing language formalizations. Since then, Xsemantics has started to be adopted also in industry (as illustrated in more details in the following), and this required to rewrite many of its parts so that complex type systems could be effectively and efficiently implemented. Its adoption in real-world type system implementations dictated the introduction of new features in the Xsemantics DSL to make its specifications modular, reusable and completely interoperable with Java. The runtime of Xsemantics had to be extended as well in order to improve the performance of the implemented type systems.

This paper extends [9] and [8] in many respects. First of all, the main features of Xsemantics are described in more details (e.g., auxiliary functions were not shown in the previous papers), still using as an example Featherweight Java (FJ) [39], a lightweight functional version of Java, which focuses on a few basic Object-Oriented features. Another example is presented, a $\lambda$-calculus DSL, to show how to implement type inference, including generic types, with unification. Although these examples are not real-world programming languages, still they feature the main characteristics of Java-like and functional languages. The above mentioned new features are described in details with examples. The internals of Xsemantics are also described, in particular, concerning the generated Java code. Moreover, general issues related to efficient implementations of type systems are discussed, especially aiming at improving the IDE experience of the implemented language.

Note that an efficient implementation of a type system is crucial in the context of an IDE: being able to quickly type a program, while the program is being edited, provides the user with an earlier error reporting. Furthermore, by using types, we can provide better code completion in the IDE (e.g., in an OOP language, we can propose the methods invokable on an expression if we know its type). Moreover, if we implement a type system for inferring types (as in Section 5), the IDE can automatically insert the inferred types in the text editor, or suggest more generic types. In any case, the type system has to be implemented efficiently, in order to keep the IDE responsive. Error recovery is another important aspect when implementing type systems. The implementation should be able to type as many parts of a program as possible, even in the presence of type errors. Similarly to error recovery in parsing, the type system should avoid to throw many errors due to previous failed type computations. This is already important in a command line compiler, but in the IDE it is even more crucial: the portions of the edited file with error markers should be kept to the minimal. This way, the user can easily spot where the problems in the program really are. To deal with this issue, as we will see in the paper (Section 7), it is required to separate type computation from type checking.

A specification DSL, like Xsemantics, must provide IDE support so that type system specifications can be easily edited with the help of all the typical tooling mechanisms that we mentioned above. Moreover, we think that such specifications should be completely interoperable with Java so that the language implementors are not forced to use Xsemantics for all the tasks and they can write some parts in Java. In that respect, Xsemantics itself is implemented in Xtext, thus it provides a complete Eclipse IDE. In particular, Xsemantics uses Xbase [24] to provide a rich Java-like syntax for defining premises in rules. Xbase is a reusable Java-like expression language that is interoperable with Java. Thanks to Xbase, from an Xsemantics definition we can refer to any existing Java library, and we can even debug Xsemantics code (Section 6.4). This also means that, when using Xsemantics, we can still implement parts of the type system directly in Java and we are not forced to use Xsemantics for everything. In an existing language implementation, this also allows for an easy incremental or partial transition to Xsemantics.

Xsemantics has proved to be mature and powerful enough to be used in real-world languages in industry.[2] Notably, it is employed to implement a full-featured type inference system for a Javascript dialect implemented in Xtext. This Javascript dialect is a super set of JavaScript with modules and classes as proposed in [3] with a static type system on top of it, which combines the type systems provided by Java, TypeScript [36] and Dart [21]. It features primitive types, declared types such as classes, interfaces, roles and union types [38] and it supports generic types and generic methods, including wildcards, requiring the notion of existential types [15]. Moreover, an industrial experience report has been recently presented at XtextCon, showing how Xsemantics helped to highly simplify the implementation of the type system in several DSLs (which have to deal with large models), making the type system implementation clearer, cleaner and easier to maintain [35].

Since Xsemantics generates readable Java code, it is possible to use all the Java tools and frameworks for code quality analysis on the generated code, e.g., Jacoco and Findbugs. Xsemantics also provides Maven integration, so that projects using Xsemantics can be built in a Continuous Integration system with build tools like Maven and Gradle. All these features have been used in the real-world industry implementations that we have just mentioned.

Finally, although Xsemantics does not aim at providing mechanisms for formal proofs, it can still be a helpful tool when proving the type safety of a language. In particular, the generated Java code keeps track of the trace of applied rules, and this

[1] Xsemantics is available as an open source project at http://xsemantics.sf.net. Sources can be found at https://github.com/LorenzoBettini/xsemantics.

[2] The author is aware of at least two commercial products using Xsemantics, having been directly involved in both of them. Currently, these languages, being commercial products, are closed-source.

```
Program: (classes += Class)* (main = Expression)?;
Class: 'class' name=ID ('extends' superclass=[Class])? '{'
    (members += Member)* '}';
Member: Field | Method;
Type: BasicType | ClassType
BasicType: 'int' | 'boolean' | 'String'
ClassType: classref=[Class]
Field: type=Type name=ID ';' ;
Method: type=Type name=ID
  '(' (params+=Parameter (',' params+=Parameter)*)? ')'
  '{' body=MethodBody '}';
Parameter: type=Type name=ID;
TypedElement: Member | Parameter;
MethodBody: 'return' expression=Expression ';' ;
Expression: Selection | TerminalExpression ;
Selection: receiver=Expression '.' message=[Member]
    ('(' (args+=Expression (',' args+=Expression)*)? ')')? ;
TerminalExpression: This | ParamRef | New | Cast | Paren ;
This: variable='this';
ParamRef: parameter=[Parameter];
New: 'new' type=ClassType '(' (args+=Expression (',' args+=Expression)*)? ')';
Cast: '(' type=Type ')' expression=Expression;
Paren: '(' Expression ')';
```

**Listing 1:** The FJ grammar implemented in Xtext.

trace, besides being useful for testing and debugging, can be used to provide hints when proving the typical type soundness formal properties (Section 4).

The paper is structured as follows. In Section 2 we provide a small introduction to Xtext. Section 3 shows the main features of Xsemantics, using FJ as example. Section 4 shows an example of reduction rules and the use of traces. In Section 5 we show another example: type inference for a $\lambda$-calculus. Section 6 describes other advanced and new features of Xsemantics. In Section 7 we sketch some implementation techniques for implementing type systems efficiently with Xsemantics (related to error recovery aspects described above). Section 8 discusses some related work and Section 9 concludes the paper.

## 2. Small introduction to Xtext

In this section we will give a brief introduction to Xtext, using Featherweight Java (FJ) [39] as an example. It is out of the scope of the paper to describe Xtext in details (we refer the interested reader to [2,7]). Here we will try to give enough details to make the features of Xsemantics understandable.

Xtext is a *language workbench* (such as MPS [64] and Spoofax [42]): it takes as input a grammar definition and it generates a parser, an abstract syntax tree, and Eclipse-based tooling features (e.g., an editor with syntax highlighting, navigation, content assist[3] and error markers). Thus, it is much more powerful than traditional parser generators (such as Flex/Bison [46] or ANTLR [50]), which only deal with the syntax of a language.

An Xtext grammar is specified using an EBNF-like syntax. The Xtext grammar for FJ is shown in Listing 1.[4] This grammar specification is similar to ANTLR [50], but it is more compact since there is no action code to specify the structure of the AST.

In an Xtext grammar, a rule is defined using a sequence of terminals (quoted strings) and non-terminals. Alternatives are marked by a pipe `|`. In rules, assignment operators define how an AST is to be constructed by the parser. The identifier on the left-hand side refers to a property of the AST class. The right-hand side is a terminal or non-terminal. The operator `+=` specifies that the property in the AST is a list. Square brackets used on the right-hand side of an assignment refer to a type of the AST. This defines a cross reference to another element in the parsed model that will be automatically linked according to the current scope definition (described in Section 2.1). For example, `body=MethodBody` means that the rule `MethodBody` will be invoked by the parser, while `superclass=[Class]` represents in the AST a reference to an element of type `Class` (it does not mean that the rule `Class` is recursively invoked). By default, cross references are parsed as identifiers.

[3] *Content assist* is the feature of the editor that automatically, or on demand, provides suggestions on how to complete the statement/expression the programmer just typed. According to the context of the program, the editor provides a list of accessible keywords, variables, etc. In Eclipse, content assist is activated by pressing `Ctrl+Space`.

[4] In this paper we show a simplified version of the implementation of FJ, in order to concentrate on Xsemantics; the complete implementation can be found at https://github.com/LorenzoBettini/xsemantics/tree/master/examples/it.xsemantics.example.fj.

```
interface Selection extends Expression {
    Expression getReceiver();
    Member getMessage();
    EList<Expression> getArgs();
}
```

**Listing 2:** The EMF Java interface generated for the rule `Selection`.

Since in FJ the class constructor has a fixed shape, we consider the constructors as implicit: when invoking `new` we must pass an argument for each field in the class, including inherited fields, in the same order of the hierarchy (arguments for inherited fields must come first). The class `Object` is implicitly defined (it is part of the FJ library). A `Type` in FJ can be either a `BasicType` or a `ClassType`; the latter is a *reference* to an FJ `Class` definition.

During parsing, the AST is automatically generated by Xtext as an EMF model (Eclipse Modeling Framework [60]). Thus, we can manipulate the AST using all mechanisms provided by EMF itself. As anticipated above, there is a direct correspondence between the rules of the grammar and the generated EMF model Java classes. For instance, in Listing 2 we show the EMF generated interface for rule `Selection` (note also the inferred inheritance relation: `Selection` is an `Expression`). Besides, Xtext generates an Eclipse editor with syntax highlighting, background parsing with error markers, outline view and content assist. Xtext automatically keeps the EMF model representing the AST and the editor's contents in synchronization.

Most of the code generated by Xtext can already be used as it is, but other parts, like type checking, have to be adapted by customizing some classes used in the framework (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 and how they relate to the features of Xsemantics. Scoping and validation together implement the mechanism for checking the correctness of a program. Keeping these two mechanisms separate fosters a modular implementation of a language, and it is typical of other approaches, such as, e.g., [56,59,25,44], where the concept of scoping is referred to as "binding".

### 2.1. Scoping

Binding the symbols (cross references) is part of checking that a program is correct. In compilers this is usually dealt with by maintaining a *symbol table*. Instead, in Xtext, when defining the grammar, we already specify that a token is actually a cross reference to a specific declared element, by using `[]`. For instance, consider the rule `Selection` in Listing 1: the feature `message` is a reference to a `Member` (`message=[Member]`), and this is reflected in the generated EMF Java code (Listing 2). In fact, `getMessage` returns a `Member`, not a string to be looked up in a symbol table. For this reason the AST is actually a graph (we will still call it AST throughout the paper).

Xtext supports the customization of binding with the abstract concept of *scope*: a scope contains all the elements (e.g., declarations) that are available in the current context of a reference. The programmer provides a customized `ScopeProvider` and Xtext, when it needs to bind/resolve a symbol, will use the scope returned by such `ScopeProvider`. Using the returned scope, Xtext will then resolve a cross reference or issue an error in case the 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. Xtext will use the returned scope also for implementing content assist.

In Java-like languages the scoping will have to deal with types and inheritance relations. For example, in FJ, the scope for `Member`s in the context of a `Selection` consists of all the members (including the inherited ones) of the class of the `receiver`. Thus, we need the type of the `receiver` expression. Such a type will be computed using the Java code generated by Xsemantics, starting from the system definition we will show in Section 3.

### 2.2. Validation

All the other checks that do not deal with name resolutions, have to be implemented through a *validator*. In a statically typed language the validation typically corresponds to checking that the program is correct with respect to types.

The programmer provides a custom `AbstractDeclarativeValidator` and define some methods with the annotation `@Check`. Xtext will automatically call these methods for validating the model according to the runtime type of the AST node being checked. The validation takes place in background, while the user is writing in the editor, so that immediate feedback is available. If an error is found during the validation, Xtext will generate the appropriate error markers. In the next sections we will show how to have a Java validator automatically generated by Xsemantics.

## 3. Xsemantics

In this section we describe the main features of Xsemantics using FJ as the underlying example language.

### 3.1. Judgments and rules

Xsemantics targets developers who are familiar with formal type systems and operational semantics since it uses a syntax that resembles rules in a formal setting [16,37,53]. A *system definition* in Xsemantics is a set of *judgments* (formally,

```
judgments {
    type |- Expression expression : output Type
        error "cannot type " + expression
    subtype |- Type left <: Type right
        error left + " is not a subtype of " + right
    subclass ||- Class left <: Class right
    subtypesequence |-
        List<Expression> expressions << List<TypedElement> elements
}
```

**Listing 3:** Judgment definitions in Xsemantics.

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

**Listing 4:** Example of a rule of the judgment `type`.

assertions about the properties of programs) and a set of *rules* (formally, implications between judgments, i.e., they assert the validity of certain judgments, possibly on the basis of other judgments [16]). Rules have a conclusion and a set of premises. Rules can act on any Java object. Typically rules will act on the EMF objects that are the elements of the 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* and the *parameters* of the judgment; parameters are separated by *relation symbols*. Symbols can be chosen from a predefined symbol set. Currently, we only support a predefined set of symbols to avoid possible ambiguities with expression operators in the premises of rules (described in Section 3.2).

Judgment symbols are

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

Relation symbols are

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

We tried to mimic the symbols that are typically used in formal systems. We also plan to add a mechanism for allowing the developer to specify custom additional 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).

The judgment definitions for FJ are shown in Listing 3. For instance, the judgment `type` takes an FJ `Expression` as input parameter and provides an FJ `Type` as output parameter. The judgment `subtype` does not have output parameters (thus its output result is implicitly boolean, stating whether the judgment succeeded). Note that `subtype` and `subclass` differ for the judgment symbol. Judgment definitions can include `error` specifications that are useful for generating custom error information (see Section 3.6). In Listing 3 we chose symbols such as `|-`, `:` and `<:` because these are the same symbols that are used in FJ formalization [39]. In general, the programmers can choose the symbols that they see fit.

Once the judgments are declared, we can start defining the rules of the 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* that can be chosen from the above predefined symbol set. To enable better IDE tooling and a more "programming"-like style, Xsemantics rules are written in the opposite direction of standard deduction rules: the conclusion comes before the premises (like, e.g., in [63,62]).

Each rule must *belong* to a judgment. The things that make a rule belong to a specific judgment are the judgment symbol and the relation symbols (which separate the parameters) of the rule conclusion. 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. In Listing 4 we show a sketched example of a rule of the judgment `type` shown in Listing 3.

The rule *environment* (in formal systems it is usually denoted by $\Gamma$ and, in the example it is named `G`) can be used to pass additional arguments to rules (e.g., contextual information, bindings for specific keywords, like `this` in FJ). An empty environment can be passed using the keyword `empty`. As shown later, the environment can be accessed with the predefined function `env`.

```
rule MyRule
    G |- Selection exp : exp.message.type   // expression instead of output parameter
from {
    // premises
}
```

**Listing 5:** Alternative implementation of the rule in Listing 4.

As a comparison, the typing rules from the original FJ type system [39,53] have the following shape, where `e` is an expression and `T` is the result type:

$$\frac{\ldots premises \ldots}{\Gamma \vdash \texttt{e} : \texttt{T}}$$

As noted above, in Xsemantics the rule conclusion comes before the premises.

At run-time, i.e., in the generated Java code, the rule environment is implemented by an instance of the Xsemantics runtime library class `RuleEnvironment`. Besides being a wrapper for a standard Java map, this class also provides additional methods for merging and nesting rule environments. Premises of a rule can also directly use the `RuleEnvironment` Java methods.

The premises of a rule, which are specified in the `from` block, can be any Xbase expression (Xbase is described in Section 3.2), or a *rule invocation*. A rule can be seen as a function declaration and a rule invocation can be seen as a function invocation, thus one must specify the environment to pass to the rule, and the input and output arguments. When passing an environment during a rule invocation, one can specify additional *environment mappings*, using the syntax `key <- value` (e.g., `'this' <- C`). Specifying additional mappings implies the creation of a copy of the environment, thus the current rule environment will not be modified. If a mapping already exists in the original rule environment, the mapping will be overwritten in the copy. Thus, the rule environment passed to a rule acts in a stack like manner.

The premises of an Xsemantics rule are considered to be in *logical and* relation and are lazily verified in the order they are specified in the block. If one needs premises (or blocks of premises) in *logical or* relation, the operator `or` must be used to separate blocks of premises. An *axiom* is a special rule that only has the conclusion, without premises. As shown in Listing 4, in the premises one can assign values to the output parameters and when another rule is invoked, upon return, the output arguments will have the values assigned in the invoked rule. An expression can be specified instead of the output parameter in the rule conclusion, e.g., the rule in Listing 5 is equivalent to the rule of Listing 4.

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. If one of the premises fails, then the whole rule will fail, and in turn the whole 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`.

### 3.2. Xbase in a nutshell

Xsemantics uses Xbase [24] to provide a rich Java-like syntax for defining premises in rules. Xbase is an extensible and reusable expression language that is interoperable with Java and its type system. The syntax of Xbase is similar to Java, though Xbase removes much "syntactic noise" from Java. Xbase should be easily understood by Java programmers. Here we sketch the main features of Xbase.

Variable declarations in Xbase start with `val` or `var`, for final and non-final variables, respectively, and do not require to specify the type 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 *extension methods* are a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. 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. Using extension methods results in a more readable code, since method calls are chained, e.g., `o.foo().bar()` rather than nested, e.g., `bar(foo(o))`.

Xbase provides *lambda expressions*,[5] which have the shape

```
[ param1, param2, ...| body ]
```

The types of parameters can be omitted if they can be inferred from the context. Xbase lambda expressions together with other syntactic sugar mechanisms allow the programmer to easily write statements and expressions that are more readable than in Java, and are very close to formal specifications. For example, a statement of the shape

$\forall x \in \texttt{L} \,.\, x \neq 0$

[5] Xbase predates Java 8 lambda expressions. By default it 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.

```
rule Subclassing
    G ||- Class left <: Class right
from {
    left.equals(right) or
    right.name.equals("Object") or
    G ||- left.superclass <: right
}
rule BasicSubtyping
    G |- BasicType left <: BasicType right
from {
    left.equals(right)
}
rule ClassSubtyping
    G |- ClassType left <: ClassType right
from {
    G ||- left.classref <: right.classref
}
```

**Listing 6:** The rules for the judgments `subclass` and `subtype`.

```
rule TCast
    G |- Cast cast : cast.type
from {
    G |- cast.expression : var Type expType
    { G |- cast.type <: expType } or { G |- expType <: cast.type }
}
```

**Listing 7:** Typing for FJ cast.

can be written in Xbase as follows[6]

```
L.forall[ x | x != 0 ].
```

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

### 3.3. Typing for FJ implemented in Xsemantics

In the next sections we sketch some rules of the type system for FJ implemented in Xsemantics.[7]

Let us first consider the rules for subclassing shown in Listing 6. The rule named `Subclassing` belongs to the judgment `subclass` and corresponds to the three subclass rules of FJ [39]. This rule takes two `Class` elements and states that the relation <: holds if one of the premises holds: the two classes are the same (subclassing is reflexive), `right` is `Object` (each class is a subclass of `Object`), recursively the superclass of `left` is subclass of `right` (subclassing is transitive). If `left.superclass` is `null`, the recursive call fails.

The subtyping between two basic types is implemented in the rule `BasicSubtyping` (belonging to the `subtype` judgment). We only need to check for equality.[8] The case for two `ClassType`s, rule `ClassSubtyping`, simply delegates to the judgment `subclass`, passing the referred class elements (recall that `ClassType` is a reference to a `Class`).

Passing to the `subtype` judgment, at runtime, any other combination of arguments different from the combinations considered by the rules `BasicSubtyping` and `ClassSubtyping`, will make the judgment fail. For example, `subtype` will fail if we pass a `BasicType` and a `ClassType`, since no rule's parameter types will match them.

In Listing 7 we show the typing rule for FJ cast expressions, to be read as: "given a `Cast` AST element, its `Type` is the type we cast to" (i.e., the attribute `type`, see the grammar rule for `Cast` in Listing 1). The premises of the rule are to be read as "if the type of the casted expression is `expType` then either the type we cast to is a subtype of `expType` or the other way round", that is, the two types must be related in the class hierarchy. Note that Xsemantics rules are type-checked by the system. The cast rule in Listing 7 is correct since `Cast` is an `Expression` and `cast.type` is a `Type`, thus the rule is conformant to the judgment declaration `type` in Listing 3.

For other FJ expressions we can simply write axioms (Listing 8). For instance, in FJ a variable can only be a reference to a method parameter, i.e., a `ParamRef`. Thus, the type of a variable is simply the type of the referred parameter (this binding has already been resolved during scoping). For typing `this` we rely on the rule environment: we access the environment with the predefined function `env`, by specifying the key and the expected Java type of the corresponding value. If no key is found in the environment or if the value cannot be assigned to the specified Java type then the premise will fail. Thus, this rule also implicitly requires that `this` is bound to a `ClassType`. This axiom assumes that the passed environment contains such a mapping for `this`, and we will see later when such mapping is passed (Listing 14, Section 3.5).

[6] Here `forall` is an extension method.

[7] For the complete implementation we refer to https://github.com/LorenzoBettini/xsemantics/tree/master/examples/it.xsemantics.example.fj.

[8] To keep the example simple we are not considering any other form of subtyping between basic types.

```
axiom TParamRef
    G |- ParamRef p : p.parameter.type
axiom TThis
    G |- This t : env(G, 'this', ClassType)
```

**Listing 8:** The axioms for parameter references and `this`.

```
rule SubtypeSequence
    G |- List<Expression> expressions << List<TypedElement> elems
from {
    expressions.size == elems.size
    var i = 0
    for (exp : expressions) {
        G |- exp : var Type expType
        G |- expType <: elems.get(i++).type
    }
}
```

**Listing 9:** The rule `SubtypeSequence`.

```
rule TNew
    G |- New newExp : newExp.type
from {
    val f = fields(newExp.type.classref)
    G |- newExp.args << f
}
rule TSelection
    G |- Selection e : e.message.type
from {
    G |- e.receiver : var ClassType receiverType
    if (e.message instanceof Method)
        G |- e.args << (e.message as Method).params
}
```

**Listing 10:** The rules `TNew` and `TSelection`.

```
auxiliary {
    superclasses(Class cl) : List<Class>
    fields(Class cl) : List<Field>
    methods(Class cl) : List<Method>
    overrides(Method current, Method previous) // predicate
        error current.name + " does not override the superclass method"
}
```

**Listing 11:** Auxiliary descriptions for FJ.

In Listing 3 we also have a judgment `subtypesequence` which takes a list of `Expressions` and a list of `TypedElements`. This judgment checks whether the expressions can be assigned to the respective typed elements (`Field` and `Parameter` are `TypedElements`) and it is implemented by the rule in Listing 9. This judgment can be used both for typing a `New` and for typing a method invocation, as shown in Listing 10.

The rule for `New` uses `fields`, which is an *auxiliary function* that reflects the homonymous auxiliary function of FJ [39]. Auxiliary functions are described in Section 3.4. Concerning `Selection`, which represents both a field selection and a method invocation, `message` is a reference to the selected `Member`, which can be either a `Field` or a `Method`. This reference is resolved by the scoping (Section 2.1), and if the selected member does not exist in the class of the receiver, the error has already been reported. The type of the selection is the type of the referred member (the feature `type` is the type of the field or the return type of the method). We also check that we can type the receiver expression. In case of a method invocation, we also check that the arguments are conformant to the parameters of the invoked method.

### 3.4. Auxiliary functions

Besides judgments and rules, Xsemantics provides *auxiliary functions*. In type systems, using auxiliary functions, rules can be written in a more compact form (see, e.g., [39,53]), delegating some tasks to such functions. *Predicates* are a special form of auxiliary functions.

Before defining auxiliary functions implementations, we need to define the signatures of such functions. These are called *auxiliary descriptions*, which play for auxiliary functions the same role as judgments for rules. For example, in Listing 11 we show some auxiliary function descriptions for FJ.

```
auxiliary superclasses(Class cl) {
    return getAll(cl,
        FjPackage.eINSTANCE.class_Superclass,
        FjPackage.eINSTANCE.class_Superclass,
        typeof(Class)
    )
}

auxiliary fields(Class clazz) {
    var fields = new ArrayList<Field>()
    // inherited fields must come first
    for (superclass : superclasses(clazz))
        fields = superclass.members.filter(typeof(Field)) + fields
    return fields + clazz.members.filter(typeof(Field))
}
```

**Listing 12:** Some auxiliary functions for FJ.

```
checkrule CheckMain for Program p from {
    empty |- p.main : var Type mainType
}
```

**Listing 13:** The checkrule for well-typedness of main expression.

The body of an auxiliary function uses the same syntax as in standard rule premises. An auxiliary function can then be called in premises as a standard Java method. At run-time, the selection of an auxiliary function takes place according to the same polymorphic dispatch mechanism of rules.

In Listing 12 we show two auxiliary functions used for implementing the type system of FJ.[9] We implemented `superclasses` by simply using an Xsemantics Java library function, `getAll`, which computes the "closure" of a graph. In particular, `getAll` collects nodes in a graph according to EMF features avoiding possible loops:

```
getAll(EObject eObject, EStructuralFeature featureToCollect,
       EStructuralFeature featureToFollow, Class expectedType)
```

An invocation of `getAll` will return a list of `expectedType`s, built by collecting all the elements from `featureToCollect` of the specified `eObject`, and recursively collecting such elements by following the feature `featureToFollow`, but avoiding possible loops in the EMF graph representing the AST. Features are specified using the standard EMF API.

The implementation of the auxiliary function `fields` shows that the Xsemantics expression language is rich and expressive thanks to Xbase. In particular, we can access any existing Java library (`filter` in Listing 12, which returns the elements compliant with the specified Java class, comes from the Xtext Java library and it is used as an extension method). Note that the return type for auxiliary functions must not be specified, since it is declared in the corresponding auxiliary description. Of course, Xsemantics will check that the returned expression is conformant to the return type of the corresponding auxiliary description.

### 3.5. Checkrules

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 an Xtext Java validator (Section 2.2). A `checkrule` has a name, a single parameter (which is the `EObject` to be checked by the validator) and the premises (but no rule environment). The syntax of the premises of a `checkrule` is the same as in the standard rules. Xsemantics will generate a Java validator with a `@Check` method for each `checkrule` (see also Section 3.6).

For instance, in Listing 13 we show the checkrule that checks that the main expression of an FJ program is type correct. We check that we can give it a type in an empty environment.

In Listing 14 we show the checkrule for checking that the type of the method body is a subtype of the method's return type. To implement this rule we first added a new judgment to our type system implementation, `assignable`, and the corresponding rule. This computes the type of the expression and checks that such a type is a subtype of the given type. Using this judgment will allow us to provide the user with better error information, as shown in the rest of this section.

In the `checkrule CheckMethodBody` we need to pass an environment where `'this'` is mapped to the class where the method is defined. The class where the method is defined is retrieved by using an additional Java utility function (`fjUtils` is a field in the type system, as explained in Section 6.1), and the mapping in the environment is added using the syntax `key <- value` described in Section 3.1. With this mapping, any occurrence of `this` can be typed using the axiom `TThis` shown in Listing 8.

[9] The Xbase keyword `typeof` specifies a Java type literal, for example the Xbase expression `typeof(String)` corresponds to the Java expression `String.class`.

```
judgments {
    ...
    assignable |- Expression expression |> Type type
        error expression + " is not assignable for " + type
        source expression
    ...
}

rule ExpressionAssignableToType
    G |- Expression expression |> Type type
from {
    G |- expression : var Type expressionType
    G |- expressionType <: type
}

checkrule CheckMethodBody for Method method
from {
    val c = method.getContainerOfType(typeof(Class))
    val classType = fjUtils.createClassType(c)
    'this' <- classType |- method.body.expression |> method.type
}
```

**Listing 14:** Well-typedness of a method body.

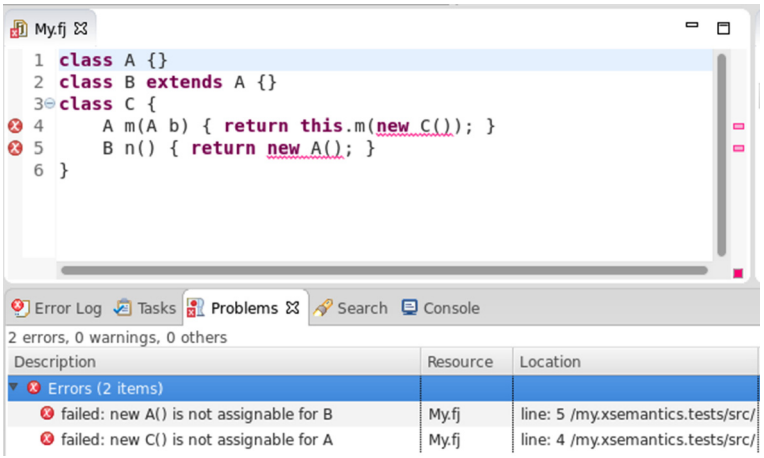


**Fig. 1.** Automatic error markers generation.

The generated Java validator will automatically generate error markers in case the corresponding rule invocations fail. Error markers will be generated according to the error information found in the trace of a failure (which is handled automatically by Xsemantics). A custom error specification can be attached to a judgment or to a single rule. For example, in the judgment declaration of Listing 14 we specified a custom error message, and the source element to place the error marker.

In Fig. 1 we show the error markers created by the generated Java validator (the error on the method invocation's argument is implemented in another `checkrule`, not shown here, that reuses the `assignable` judgment). Note that our custom error message is used. Furthermore, the error marker is generated in correspondence to the offending element (the `source` specification in the `assignable` judgment).

To provide better error diagnosis, the developer can also specify a premise in `or` conjunction with an explicit `fail` and a detailed error message. Other approaches, like [33,32], explicitly use constraints and retain a looseness in solving them to avoid bias in error diagnosis. We plan to investigate in that direction as well.

### *3.6. Generated code*

Xsemantics provides a runtime library with some Java classes. This library contains utility classes that can be used by the developer, classes for elements in the generated code (e.g., result, failures, etc.) and base classes for the generated code.

Starting from a system definition, Xsemantics generates a Java class with all the rules and auxiliary functions. The generated Java class extends the library class `XsemanticsRuntimeSystem`. This base class contains methods that can be used by the generated code. For example, the functions `env` and `getAll` are implemented in this base class.

Xsemantics generates entry point methods for the judgments and auxiliary descriptions. The programmer should call these entry methods (the generated Java methods for rules, checkrules and auxiliary functions are not meant to be called by the programmer). For example, for the judgments `type` and `subtype` in Listing 3 it will generate the following Java methods:

```
public class FjTypeSystem extends XsemanticsRuntimeSystem {
  private PolymorphicDispatcher<Result<Type>> typeDispatcher =
    buildPolymorphicDispatcher("typeImpl", "|-", ":");

  public Result<Type> type(RuleEnvironment _environment_,
          RuleApplicationTrace _trace_, Expression expression) {
    try {
      return typeInternal(_environment_, _trace_, expression);
    } catch (Exception _e_type) {
      return resultForFailure(_e_type);
    }
  }

  protected Result<Type> typeInternal(RuleEnvironment _environment_,
          RuleApplicationTrace _trace_, Expression expression)
          throws RuleFailedException {
    checkParamsNotNull(expression);
    return typeDispatcher.invoke(_environment_, _trace_, expression);
  }
```

**Listing 15:** Some snippets of the generated code (part 1).

```
public Result<Type> type(RuleEnvironment e,
                         RuleApplicationTrace trace,
                         Expression object)
public Result<Boolean> subtype(RuleEnvironment e,
                         RuleApplicationTrace trace,
                         Type left, Type right)
```

The returned `Result` contains either the result of a successful computation or a `RuleFailedException` with all the details of the failure. These classes are part of the Xsemantics runtime library. If the passed `RuleApplicationTrace` argument is not null, it will be automatically filled by Xsemantics with information about the applied rules. Traces will explained in more details at the end of this section and in Section 4.

In Listing 15 we consider the case for `type`. The implementation of such a method, which is the one called by the clients of the type system, calls the "internal" version of this method `typeInternal`. Note that since the entry point methods are not meant for throwing exceptions, the generated try-catch block catches possible exceptions and wraps them inside the returned `Result`. The internal method, after performing additional sanity checks, starts the polymorphic dispatch mechanism in order to select the generated Java methods according to the runtime type of the passed arguments. `PolymorphicDispatcher` is part of the Xtext library.

Note that, besides the above strategy for selecting Java methods, Xsemantics itself does not implement, nor 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 application's flow of control.

For the case of `type`, the polymorphic dispatcher will search for methods with the name `typeImpl`. There will be such a method for each rule in the original Xsemantics system (the same holds for auxiliary functions and checkrules). In Listing 16 we show the generated methods for the rule `TNew` previously shown in Listing 10. The `Impl` method will take care of recording information in the rule application trace and generating possible failures concerning the rule. This method then calls the generated method that contains the generated code for the premises of the rule (in this case `applyRuleTNew`). The body of this method is generated by the Xbase compiler, which we customized for Xsemantics' specific expression syntax, such as, e.g., rule invocations and `or` blocks. The reader can compare the body of this method with the original rule in Listing 10. The Xsemantics compiler also generates Java comments for rule invocations, containing the text of the original premise. In the body of `applyRuleTNew`, the method call `fieldsInternal` corresponds to the auxiliary description `fields` (shown in Listing 11) and the method call `subtypesequenceInternal` corresponds to the judgment `subtypesequence` (shown in Listing 3). Thus, the generated code for rule premises directly call the internal methods.

The Java code generated by Xsemantics can be used when implementing the Xtext scope provider as sketched in Section 2.1. Xsemantics will also generate a Java validator, with a `@Check` method for each `checkrule` (as described in Section 3.5) that can be used as the Xtext validator for the implemented language.

The failure trace and the application trace are available also for the programmer, for instance, for testing or debugging purposes. An example of use of the application trace is shown in Section 4. The Xsemantics runtime library provides some utility methods for manipulating the traces. For example, the traces shown in Section 4 are formatted using such utility methods. Note that the information to insert in the trace are computed lazily by Xsemantics, i.e., only if the passed `trace` argument is not null. This small improvement, contributed by Xsemantics' clients, drastically improved performance in a real-world type system implementation.[10]

[10] https://github.com/LorenzoBettini/xsemantics/pull/27.

```
protected Result<Type> typeImpl(RuleEnvironment G,
        RuleApplicationTrace _trace_, New newExp) throws RuleFailedException {
  try {
    RuleApplicationTrace _subtrace_ = newTrace(_trace_);
    Result<Type> _result_ = applyRuleTNew(G, _subtrace_, newExp);
    // update the rule application trace (not shown)
    return _result_;
  } catch (Exception e_applyRuleTNew) {
    // throw exception with error information (not shown)
    return null;
  }
}

protected Result<Type> applyRuleTNew(RuleEnvironment G,
      RuleApplicationTrace _trace_, New newExp) throws RuleFailedException {
  ClassType _type = newExp.getType();
  Class _classref = _type.getClassref();
  List<Field> fields = this.fieldsInternal(_trace_, _classref);
  /* G |- newExp.args << fields */
  EList<Expression> _args = newExp.getArgs();
  this.subtypesequenceInternal(G, _trace_, _args, fields);
  return new Result<Type>(_type);
}
```

**Listing 16:** Some snippets of the generated code (part 2).

```
rule RSelection
    G |- Selection e ~> Expression e1
from {
    val receiver = e.receiver as New
    val message = e.message
    switch (message) { // type-based switch from Xbase
        Field: { // message is of type Field in this context
            val fieldIndex = fields(receiver.type.classref).
                    indexOf[ f | f.name.equals(message.name) ]
            e1 = receiver.args.get(fieldIndex)
        }
        Method: { // message is of type Method in this context
            e1 = fjUtils.replaceThisAndParams(message.body.expression,
                receiver, message.params, e.args)
        }
    }
}
```

**Listing 17:** Reduction rule for `Selection`.

## 4. Reduction rules and traces

In this section we show how Xsemantics can also be used to implement the reduction rules for a language. Such rules can be used for generating target code in the compiler or to implement an interpreter. Most of all, using FJ as an example, we show that these rules can implement directly the formalized operational semantics of the language. Finally, we will sketch how the traces of applied rules can provide hints for proving the type soundness of the language.

In the original FJ formalization [53], the operational semantics is defined by the reduction relation $e \rightsquigarrow e_1$, read "the expression $e$ reduces to $e_1$ in one step", adopting a deterministic call-by-value semantics. The semantics consists of congruence rules which formalize how operators (method invocation, object creation, and field selection) are reduced only when all their sub-expressions are reduced to values (call-by-value), and actual reduction rules which apply when all subexpressions of an expression are values. In FJ the only values are `new` expressions where all arguments are themselves values.

We only show the most interesting case of reduction, i.e., `Selection`, when all subexpressions are values, in Listing 17.

The premises in this rule use some Xbase features like the type-based `switch` and lambdas, which should be easily understood. For field selection we find the index $i$ in the field list (including the inherited ones) corresponding to the selected field, and the resulting expression is the $i$-th value passed in the `new` expression. For method invocation the result is the method body expression after replacing `this` with the receiver and the parameter references with the arguments (using the Java utility method `replaceThisAndParams`, not shown here).

This rule is pretty close to the corresponding rules in FJ, which we report here for convenience (the overline notation denotes sequences):

$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})).\mathtt{f}_i \rightsquigarrow \mathtt{v}_i} \qquad \frac{\mathit{mbody}(\mathtt{C}, \mathtt{m}) = (\bar{\mathtt{x}}, \mathtt{e})}{(\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})).\mathtt{m}(\bar{\mathtt{u}}) \rightsquigarrow [\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})/\mathtt{this}, \bar{\mathtt{u}}/\bar{\mathtt{x}}]\mathtt{e}}$$

```
class A {
   Object m() { return this.n(new B()); }
   A n(A o) { return new A(); }
}
class B extends A {}
new A().m()
```

**Listing 18:** An example of FJ program.

```
TSelection [this <- A] |- this.n(new A()) : A
  TThis [this <- A] |- this : A
  SubtypeSequence [this <- A] |- [new B()] << [A o]
    ClassSubtyping [this <- A] |- B <: A
      Subclassing [this <- A] ||- B <: A
        Subclassing [this <- A] ||- B <: B
  ClassSubtyping [] |- A <: Object
    Subclassing [] ||- A <: Object
```

**Listing 19:** Application trace for `CheckMethodBody`.

```
judgments {
   subjred |= Expression e ~> output Expression : output Type <: output Type
}

rule SubjRed
   G |= Expression e ~> Expression e1 : Type C1 <: Type C
from {
   G |- e : C    G |- e ~> e1    G |- e1 : C1    G |- C1 <: C
}
```

**Listing 20:** A possible judgment and rule for testing Subject Reduction.

As hinted before, the trace of applied rules can be used for testing that the rules are applied as expected. We believe that the automatic trace mechanism is a valuable add-on, since, in case of a type system implemented directly in Java, the traces should be implemented manually. For instance, consider the FJ program in Listing 18. Using the generated Java code to check that the method `m` is well-typed, we can inspect the application trace contained in the result, which presents the rules applied as shown in Listing 19. You can compare the output with the corresponding rules presented in the previous sections (starting from Listing 14).

Xsemantics does not aim at providing mechanisms for automatic proofs; for instance, it does not produce, like other frameworks do (see, e.g., [59,63]), versions of the type system for proof assistants like Coq [5], HOL [30] or Isabelle [51]. However, Xsemantics can still help when writing the meta-theory of the language. For instance, consider the standard *Subject Reduction Theorem* which states that if an expression `e` has type `C` and it reduces in one step to `e1` then `e1` has type `C1` for some `C1 <: C`. We can write a judgment and a rule to express this property as shown in Listing 20. If we invoke the generated method `subjred` on the main expression `new A().m()` of the FJ program in Listing 18 we get the trace shown in Listing 21. We might write similar judgments and rules also for the *Substitution Lemma* and the *Progress Theorem*. This mechanism can be used to test and debug the type system and the semantics implemented in Xsemantics. While this does NOT correspond to formally proving the type soundness of the language, still we believe that it is a very useful tool when designing, formalizing and implementing a language and proving its formal properties. For example, the standard mechanism for proving the *Subject Reduction Theorem* is by "structural induction on a derivation of a reduction of an expression `e` to `e1`, with a case analysis on the reduction rule used" [39]. For each case the main used assumption is the fact that the initial expression `e` is well-typed, and this allows us to derive properties on the sub-expressions as well. Thus, in such proof, the trace of the deduction is a useful hint. It is out of the scope of the paper to get into details of formal proofs (we refer the interested reader to [39,53]). We only want to note that these proofs by structural induction are based on the structure of the derivation, i.e., on the applied rules (both reduction and type rules), that is why we think that the traces of the Xsemantics rules can help when proving the type soundness.

In that respect, we believe that Xsemantics provides mechanisms for rapidly testing the implementation of a type system and of reduction rules. Thus, Xsemantics approach is similar to the "tests complement proofs" and "tests complement even machine-checked proofs" of Redex [43].

## 5. Type inference for λ-calculus in Xsemantics

As another example of use of Xsemantics we show a prototype implementation of λ-calculus, whose Xtext grammar is shown in Listing 22. In this λ-calculus we can specify the type of the parameter of the abstraction, but we can also leave it empty; in that case, the type will be inferred. In particular, we infer types using type variables when the type of a term can be generic. The types of this λ-calculus can be basic types (in this example integer or string), arrow types, and type

```
SubjRed: [] |= new A().m() ~> new A().n(new B()) : A <: Object
 TSelection: [] |- new A().m() : Object
  TNew: [] |- new A() : A
   SubtypeSequence: [] |- [] << []
  SubtypeSequence: [] |- [] << []
 RSelection: [] |- new A().m() ~> new A().n(new B())
 TSelection: [] |- new A().n(new B()) : A
  TNew: [] |- new A() : A
   SubtypeSequence: [] |- [] << []
  SubtypeSequence: [] |- [new B()] << [A o]
   TNew: [] |- new B() : B
    SubtypeSequence: [] |- [] << []
   ClassSubtyping: [] |- B <: A
    Subclassing: [] ||- B <: A
 ClassSubtyping: [] |- A <: Object
  Subclassing: [] ||- A <: Object
```

**Listing 21:** Application trace for `SubjRed` applied to `new A().m()`.

```
Term: TerminalTerm | Application ;
TerminalTerm: '(' Term ')' | StringConstant | IntConstant |
   Arithmetics | Variable | Abstraction;
StringConstant: string=STRING ;
IntConstant: int=INT ;
Arithmetics: '-' term=Term ;
Variable: ref=[Parameter] ;
Application: fun=Term arg=TerminalTerm;
Abstraction: 'lambda' param=Parameter '.' term=Term ;
Parameter: name=ID (':' type=Type)? ;
Type: TerminalType | ArrowType ;
TerminalType: '(' Type ')' | BasicType | TypeVariable;
BasicType: {IntType} 'int' | {StringType} 'string' ;
TypeVariable: typevarName=ID ;
ArrowType: left=Type '->' right=Type ;
```

**Listing 22:** The λ-calculus grammar implemented in Xtext.

```
auxiliary {
    notoccur(Type type, Type other)
        error type + " occurs in " + other
    typesubstitution(TypeSubstitutions substitutions, Type original) : Type
    unify(TypeSubstitutions substitutions, Type left, Type right) : Type
        error "cannot unify " + left + " with " + right
}

judgments {
    type |- TypeSubstitutions substitutions |> Term term : output Type
    paramtype |~ Parameter param : output Type
}
```

**Listing 23:** Judgment and auxiliary descriptions for λ-calculus.

variables. The challenging part in writing a type system for this language is that we need to perform *unification* in order to infer the most general type (see, e.g., [57]).[11]

The judgments and auxiliary descriptions for the type system are shown in Listing 23. The auxiliary functions implementing `notoccur` (corresponding to the unification *occurcheck*) and `typesubstitution` are not interesting.[12] In particular, `typesubstitution`, given the map of substitutions for variables and a type, returns a brand new type where all the substitutions have been applied (`TypeSubstitutions` is an `HashMap<String,Type>`). Note that substitutions are applied recursively: a type variable can be mapped to another type variable, which in turn can be mapped to something else and so on. For example, given the mappings `X1 = (int->X2)` and `X2 = string`, and the arrow type `X1->X2`, the result after the substitution must be `(int->string)->string`.

[11] In the future, we plan to extend this example with Damas–Milner let polymorphism [20].

[12] The complete example can be found at https://github.com/LorenzoBettini/xsemantics/tree/master/examples/it.xsemantics.example.lambda.

```
auxiliary unify(TypeSubstitutions substitutions, Type t1, Type t2) {
    fail // if we get here we cannot unify the two types
}
auxiliary unify(TypeSubstitutions substitutions, StringType t1, StringType t2) {
    return t1
}
auxiliary unify(TypeSubstitutions substitutions,
                TypeVariable typeVar, BasicType basicType) {
    substitutions.put(typeVar.typevarName, basicType)
    return basicType
}
```

**Listing 24:** First cases of `unify`.

```
auxiliary unify(TypeSubstitutions substitutions,
          TypeVariable left, TypeVariable right) {
    val fresh = lambdaUtils.createFreshTypeVariable
    substitutions.add(left.typevarName, fresh)
    substitutions.add(right.typevarName, fresh)
    return fresh
}
auxiliary unify(TypeSubstitutions substitutions, TypeVariable v, ArrowType arrow) {
    notoccur(v, arrow) // occur check
    substitutions.add(v.typevarName, arrow)
    return arrow
}
```

**Listing 25:** Unifying two variables.

```
rule ParameterType
    G |~ Parameter param : Type type
from {
    {
        param.type != null
        type = param.type
    }
    or
    type = lambdaUtils.createFreshTypeVariable
}
```

**Listing 26:** Typing of parameters.

The auxiliary description `unify` takes the substitutions map, two types and outputs the result of the unification. It tries to unify the two input types, and if the unification succeeds it outputs the new type, which is the unified version of the two input types, and it keeps track of the substitutions. The first cases for the auxiliary description `unify` are in Listing 24.[13]

The first case is the fallback case, when we try to unify a combination of types that cannot be unified, e.g., a `StringType` and an `IntType` (recall the polymorphic dispatch for runtime selection). The cases for basic types are trivial. Unifying a variable and a basic type results in returning the basic type after recording the substitution for the variable.

Unifying two variables means creating a fresh new variable (we use a utility function that uses an incremental counter to create fresh variable names), recording the substitution for both original variables, and returning the fresh variable as result. Unifying a variable and an arrow type (the symmetric case is not shown) requires to check that the variable does not occur in the arrow type. These two cases are shown in Listing 25.

The occur check fails for terms like `lambda x.x x` that cannot be typed, since `x` should be given types `a -> b` and `a`. The unification will fail since the variable occurs in the arrow type. Finally, unifying two arrow types simply requires to call the unification recursively on the arrows' elements (not shown here).

As shown in Listing 23, we have a specific type judgment for parameters only. This will make other typing rules easier to write. In fact, in our λ-calculus DSL we can specify a type for the parameter or let the type system infer it. The only rule for this judgment simply considers these two cases (if no type is specified we create a fresh type variable), as shown in Listing 26.

We can now implement the rules for the main `type` judgment. Note that the `type` judgment also takes the substitution map as input. In fact, during type inference we will call `unify` and we will apply the type substitutions recorded in such a map during unification.

For example, the type of a variable is not simply the type of the referred parameter, since the parameter might have no declared type (that is the reason explained above for the introduction of a special judgment for the type of a parameter).

[13] We are not showing the case for `IntType` and the symmetric case for `BasicType,TypeVariable` since they are similar.

```
rule VariableType
    G |- TypeSubstitutions substitutions |> Variable variable : Type type
from {
    val paramType = env(G, variable.ref, Type)
    type = typesubstitution(substitutions, paramType)
}
```

**Listing 27:** Typing of variables.

```
rule AbstractionType
    G |- TypeSubstitutions substitutions |> Abstraction abs : ArrowType type
from {
    G |~ abs.param : var Type paramType
    G, abs.param <- paramType |- substitutions |> abs.term : var Type termType
    type = lambdaUtils.createArrowType(typesubstitution(substitutions, paramType),
        typesubstitution(substitutions, termType))
}
```

**Listing 28:** Typing of lambda abstraction.

```
rule ArithmeticsType
    G |- TypeSubstitutions substitutions |> Arithmetics arithmetics : IntType intType
from {
    intType = lambdaUtils.createIntType
    G |- substitutions |> arithmetics.term : var Type termType
    // the type of the term must unify with int type
    unify(substitutions, termType, intType)
}
```

**Listing 29:** Typing of arithmetic expression.

```
rule ApplicationType
    G |- TypeSubstitutions substitutions |> Application application : Type type
from {
    G |- substitutions |> application.fun : var Type funType
    // make sure funType can be unified with an arrow type
    val arrowType = unify(substitutions, funType, lambdaUtils.createFreshArrowType)

    G |- substitutions |> application.arg : var Type argType
    // unify arrow's left part with the type of the argument
    unify(substitutions, arrowType.left, argType)

    // the result is the arrow's right part after substitutions
    type = typesubstitution(substitutions, arrowType.right)
}
```

**Listing 30:** Type inference for an application.

Thus, we assume the inferred type of the parameter is passed in the environment, and the result is such type after applying substitutions (Listing 27).

The rule for lambda abstraction is shown in Listing 28. After typing the lambda body (with an environment containing the inferred type for the parameter), the resulting type is an arrow type after applying unification substitutions to the type of the parameter and the type of the body.

The type of an arithmetic expression is `int`, however we must check also that the type of the subexpression can be unified with `IntType`. If that is not the case, the type judgment will fail (Listing 29). This way, we can give the type `int -> int` to lambda expressions such as `lambda x.-x`. While expressions such as `lambda x.-"hi"` will issue an error of the shape "cannot unify string with int" (see the error specification in the judgment of Listing 23).

Finally, the type of an application expression (Listing 30) requires to type the left expression, and check that it is a lambda expression (i.e., unify it with an arrow type). Then we type the right part (the argument of the application), and unify it with the inferred parameter type of the lambda expression. The result will be the type of the right part of the arrow type after applying all the substitutions collected when typing the subexpressions. Some examples of lambda expressions and the corresponding inferred types are shown in Table 1.

The generated type system can be used not only for checking the validity of a program, but also for implementing other IDE features. In the λ-calculus DSL example, we use the type system to implement an editor action to automatically insert inferred types in the program, as shown in Fig. 2. Similarly, we can substitute an explicitly written type with a possible more general one (for instance, if we wrote `lambda x : int -> int. x`, we can replace it with the more general `lambda x : a -> a. x`).

Note that Xsemantics assumes that the formal type system specification is already in an algorithmic form, thus, it deals with deterministic type checkers that do not require backtracking search. If this is not case, then the implementation in

**Table 1**
Some examples of inferred types.

| | |
|---|---|
| `lambda x . x` | `a -> a` |
| `lambda x . 10` | `a -> int` |
| `lambda x . -x` | `int -> int` |
| `(lambda x . lambda y . y) 10` | `a -> a` |
| `lambda x . lambda y. x y` | `(a -> b) -> a -> b` |
| `lambda x . lambda y. y x` | `a -> (a -> b) -> b` |
| `lambda f . (lambda x. (f (f x)))` | `(a -> a) -> a -> a` |
| `lambda f . lambda g. lambda x. (f (g x))` | `(a -> b) -> (c -> a) -> c -> b` |

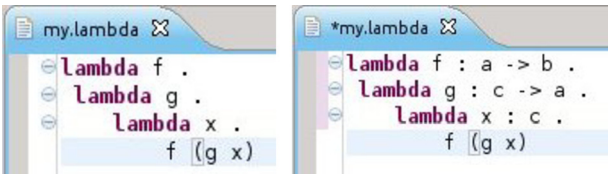


**Fig. 2.** Inferring types in the λ-calculus DSL editor.

Xsemantics requires an adaption of the original type system specification. This was not the case for FJ, whose type system rules are already in the desired form. Instead, for the λ-calculus, we had to call unification explicitly in our Xsemantics specification, while the original type system rules implicitly rely on unification. In the future, we can investigate whether we can relax these assumptions, possibly by adding to Xsemantics some special support for mechanisms like the unification itself, in order to make Xsemantics specifications as close as possible to the original formal specifications.

## 6. Other features

In this section we describe some additional features of Xsemantics that help the developer to implement type systems easily, both from the modular and the performance point of view. These are also new features with respect to the previous presentation of Xsemantics [8].

### 6.1. Fields and imports

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. This way, a type system implemented in Xsemantics can delegate specific tasks to such Java utility classes. Examples of fields are `fjUtils` in Listings 14 and 17 and `lambdaUtils` in Section 5.

Xsemantics also supports Java-like import statements, including Java static imports, so that we can refer from within Xsemantics premises to external Java classes' static methods without the fully qualified name. Note that the Xsemantics Eclipse editor supports automatic insertion of imports during content assist, just like the Eclipse Java editor. 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 3.2) and it will make Xsemantics code less verbose and more compact. Utility methods of the Xtext libraries, like, e.g., `forall` and `indexOf`, which we have already used in the paper, are implicitly imported and can be already used as extension methods.

### 6.2. System extension

An Xsemantics system can 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. The extended system 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. An example of system extension is shown in Listing 31. We extend the FJ type system shown in Section 3.3: the judgment `type` is overridden to change the error message (compare that with Listing 3) and the original axiom for `this` (Listing 8) is turned into a rule with a different error message in case of failures.

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, system extension is useful to quickly test possible modifications/improvements to an existing type system. For example, it can be used for testing that caching

```
system CustomFjTypeSystem extends FjTypeSystem

judgments {
    override type |- Expression expression : output Type
        error expression + " cannot be given a type"
}

override rule TThis
    G |- This t : Type thisType
from {
    thisType = env(G, 'this', ClassType)
    or fail error "'this' cannot be used from outside a method"
}
```

**Listing 31:** An example of system extension.

```
judgments {
    subclass ||- Class left <: Class right cached
}
auxiliary {
    superclasses(Class cl) : List<Class> cached
}
rule Subclassing // modified
    G ||- Class left <: Class right
from {
    left.equals(right) or
    right.name.equals("Object") or
    superclasses(left).contains(right)
}
```

**Listing 32:** Example of use of caching in the FJ type system.

features (Section 6.3) do not break the type system implementation. Another use case could be the use of slightly different type system implementations, for example, one for the stand-alone compiler of the language and one for the IDE.

### 6.3. Caching

As we saw in the previous sections, in a language implemented with Xtext, types are used in many places by the framework, e.g., in the scope provider and in the validator. In turn, the scope provider is also internally used by other parts of Xtext, for example, in the content assist for computing the proposals for a FJ method invocation expression.

Besides that, some type computation rules, some subtyping checks and some auxiliary functions are also used more than once in the type system implementation. For example, whether a class `A` is a subclass of another class `B` may have to be checked by many checkrules. The same holds for auxiliary functions such as `subclasses`, `fields`, etc.

For the above reasons, it makes sense to cache the results of type computations in order 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, it is crucial to invalidate the cache appropriately.

Xsemantics provides automatic caching mechanisms that can be enabled in a system specification. These mechanisms internally use at run-time the class `OnChangeEvictingCache` that is part of the Xtext utility library. This class implements a cache that stores its values in the scope of a resource. The cache will be automatically cleared as soon as the contents of the resource changes semantically. 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. Thus, Xsemantics caching is a wrapper for the Xtext caching mechanism. Besides that, our caching mechanism also handles traces according to caching. When caching is enabled, traces also keep track of results that are effectively computed and of results that are retrieved from previously cached computations. This way, traces can be used also to have statistics on how caching is effectively used in a type system implementation.

The programmer can enable caching, using the keyword `cached` on a per-judgment and per-auxiliary description basis. The rationale behind this granularity is that caching should be enabled with care, otherwise it could decrease the performance. The caching is based on the Java hashing features (in the future, it will be possible to customize this behavior as well), 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.

For example, as shown in Listing 1, an FJ `ClassType` is only a reference to a `Class` object, thus, caching the `subtype` judgment makes no sense since each `ClassType` is a different object in a program, even if they refer to the same `Class` instance. On the contrary, the judgment `subclass` can be safely cached, since it concerns objects that are shared from within an FJ program. To further benefit from caching, we can also modify the rule `Subclassing` so that it uses the

```
judgments {
    subtype |- Type left <: Type right cached {
        condition=(left instanceof BasicType && right instanceof BasicType)
    }
}
```

**Listing 33:** Example of use of caching with a condition.

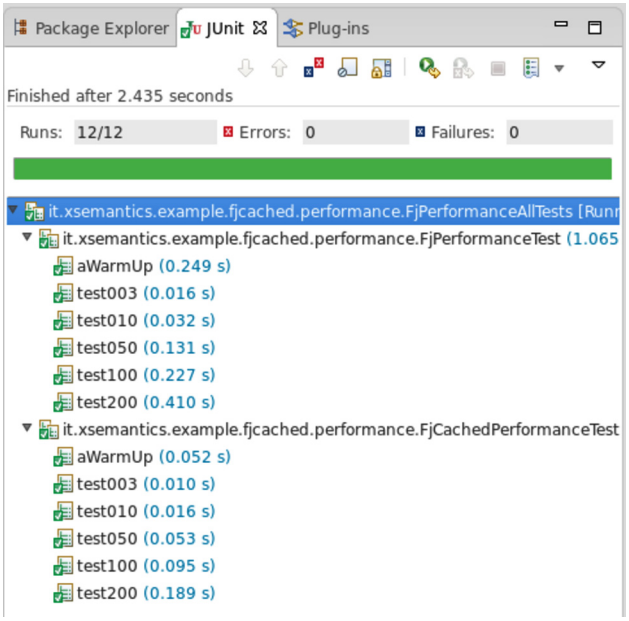


**Fig. 3.** JUnit tests for FJ type system.

auxiliary function `superclasses` (originally shown in Listing 12) and we enable caching on the `superclasses` auxiliary description. The modifications to the type system are shown in Listing 32.

The `cached` clause also supports a boolean `condition` that specifies whether caching is to be used or not, depending on the arguments passed at runtime to rules and auxiliary functions. For example, based on what described above concerning caching, references and objects, we could enable caching on the `subtype` judgment only for basic types, as shown in Listing 33.

Let us now consider an FJ main program of the shape

$$\texttt{new}\ A_n(\texttt{new}\ A_{n-1}(\ldots \texttt{new}\ A_1(\texttt{new}\ A_0(\texttt{new}\ B()))\ldots))$$

where $A_i <: A_{i-1}$, $A_0 <: B$ and $A_0$ defines a field of type $B$. The above main program requires to use subtyping extensively on the argument passed to the constructor. We ran a few JUnit tests with the standard type system presented in Section 3.3 and with the cached version presented in this section. The input is a main program of the above shape. We parse and validate the program in the JUnit tests. The results are shown in Fig. 3,[14] where the number in the name of the test method represents $n$. JUnit reports the execution time for each test case. From these tests, we see that caching halves the time and it does not impact the performance even on small programs.

In languages like Java where most elements are explicitly typed (fields, methods, variables), type computation does not require much time since it does not have to visit the AST too deeply. Let us consider another example of Xsemantics: in the *Expressions* DSL one can define variables initialized with an expression, which includes arithmetic and boolean expressions. Such expressions can refer to variables already defined in the program. Variables do not declare any type, since their types is inferred by the type system using the initialization expression.[15] If we consider programs of the shape

$$\begin{aligned} &\ldots \\ v_{n-1} &= v_{n-2} + v_{n-3} + \ldots + v_0 \\ v_n &= v_{n-1} + v_{n-2} + \ldots + v_0 \end{aligned} \tag{1}$$

[14] The test method `aWarmUp` must be ignored. It is used to make sure that both the JVM and the Xtext internal mechanisms are started and they do not interfere with the time measurements.

[15] For the complete implementation we refer to https://github.com/LorenzoBettini/xsemantics/tree/master/examples/it.xsemantics.example.expressions.

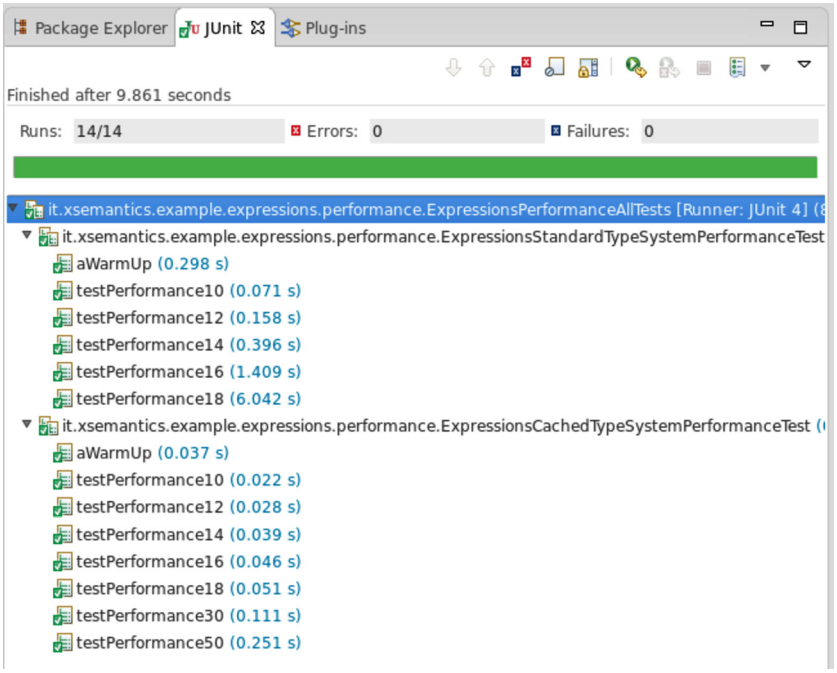


**Fig. 4.** JUnit tests for the *Expressions* DSL type system.

where $v_0$ is initialized with any numeric or boolean constant, and $v_{i-1}$ is defined before $v_i$, we can realize that type inference will take a huge amount of time, due to typing the same initialization expressions over and over again. In this case, caching does not only improve performance significantly, but it also allows the compiler and the editor to be usable. The JUnit tests shown in Fig. 4 are based on the same approach that we used for FJ. Without caching, even with 18 variables defined with the pattern shown above, the editor becomes almost unusable, since 6 s are not acceptable (with $n = 20$ the test takes too much time without caching, so we stopped at $n = 18$). On the contrary, caching makes the DSL scale without problems, even with larger corner cases.

Although the performance tests shown above, both for FJ and for the Expressions DSL, look artificial, still we think that it is important to test the type system against corner cases. In any case, caching, if enabled on the right parts of the type system, does not impact the performance on small programs, and makes a big difference on bigger ones.

### 6.4. Xsemantics IDE

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., Fig. 5). 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, with the Xsemantics IDE we can debug not only the generated Java code, but also the original Xsemantics rule definitions. Fig. 6 shows a debug session: 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). We believe that all the above features are extremely important for the effective usability of Xsemantics, especially in complex type systems.

## 7. Type systems for the IDE

Xtext makes language development easy [27], however, implementing high quality IDE mechanisms might require some fine-tuning at the right places [69,7]. Common pitfalls in language implementations are typical from the very beginning of the development. For instance, one might be tempted to rule out errors in a source program in the grammar itself. This could leave to a less efficient parsing (e.g., requiring more backtracking), not to mention that syntax errors are less informative in many cases. Another common pitfall is to mix *visibility* and *accessibility*: scoping (Section 2.1) should deal with visibility, not necessarily with validity. For example, an element can be visible in a certain program context, but that context should not access it. Think of a Java subclass trying to access a private field in a superclass. The private field is

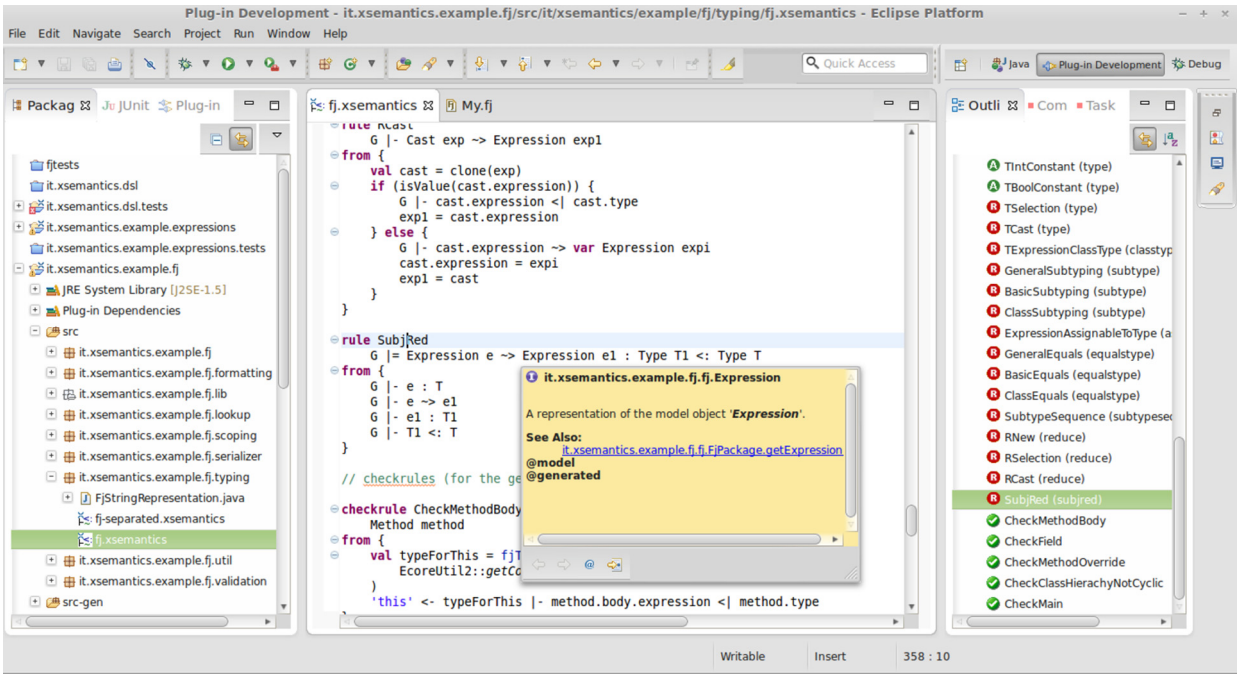


**Fig. 5.** Xsemantics IDE.

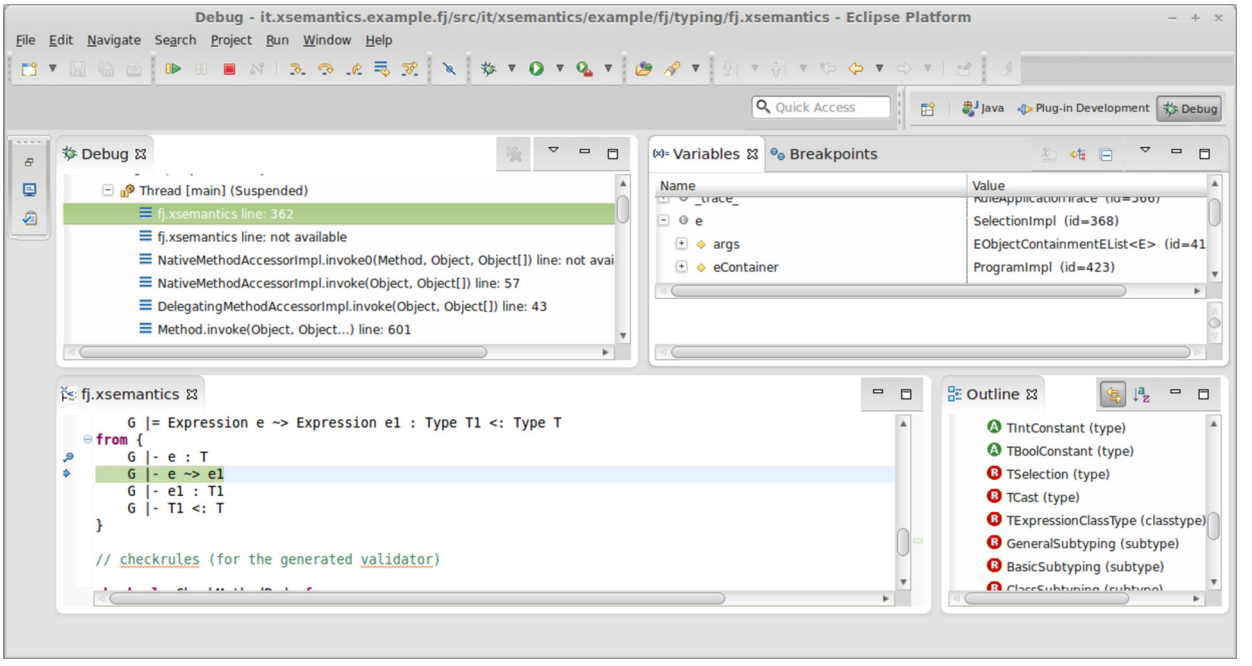


**Fig. 6.** Debugging Xsemantics code.

"visible" in the subclass, but it is not "accessible".[16] The `javac` compiler reports an informative error of the shape "the field has private access" (accessibility error), not an obscure "the symbol cannot be found" (visibility/binding error). The standard Eclipse JDT editor (Java Development Toolkit) follows a similar approach: the user will still be able to navigate to the definition of a member even if it is not accessible in that part of the program.

[16] We observe that `private`, `protected`, etc., are called *access level modifiers* [31].

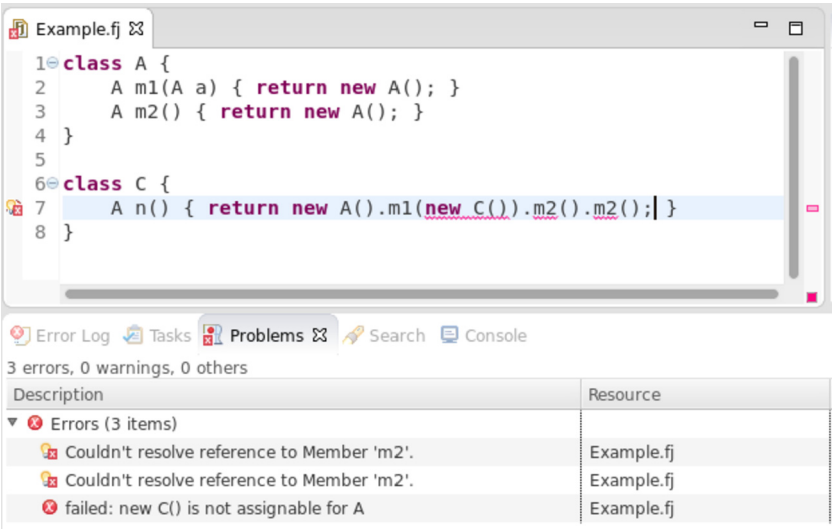


**Fig. 7.** Too many errors reported.

```
axiom TSelection
  G |- Selection selection : selection.message.type
```

**Listing 34:** Loose type inference.

In general, we think that it is better to avoid that a validation error in a program leads to a cascade of errors in the rest of the program.[17] We should keep this in mind when implementing the type system of a language, in order to provide informative errors to the programmer, that also integrate with the IDE tooling.

Typically, formal type systems specifications deal with type computation (or type inference) and validation at the same time. Thus, a type computation rule for an expression also checks that the expression is correct. We followed the same strategy in the Xsemantics rules presented in Section 3. However, type computation might not depend on the correctness of the expression. In that case, we can adopt a "loose type inference" strategy when we can compute the type of an expression even if the expression is not valid. For example, consider the rule for an FJ selection expression, shown in Listing 10. Even if the arguments passed to the method invocation are not correct (e.g., wrong number or non-conformant types), we are still able to compute its type, which is the declared type of the method. Since such type information is used during the scoping, following a loose inference strategy will avoid presenting the user with too many errors.

For example, consider the FJ program in Fig. 7. With the current implementation of typing, the rule for computing the type of the method invocation expression `new A().m1(new C())` fails, thus the scoping will fail as well, and it will not be able to resolve the subsequent method invocation `m2`. This leads to a cascade of errors, which might be confusing for the user, even in a small program.

If, on the contrary, we separate type computation from validation, we will present the user with more informative error and avoid to clutter the error reporting with too many messages. In Xsemantics this means that the type computation rule will be simpler (typically an axiom), as shown in Listing 34, and that we will write a separate `checkrule` that validates the correctness of an expression in isolation. The result can be seen in Fig. 8: the editor reports only the important error information, which is easier to understand. Similarly, the content assist will still be able to provide helpful proposals and we can also navigate to the definition of the method `m2`.

While this strategy requires to write more code, it provides a better user experience, and it should be kept in mind for providing high quality implementations.

## 8. Related work

In this section we discuss some related work, focusing on language frameworks and on DSLs for specifying type systems and reduction rules.

The work that is closest to Xsemantics is XTS [66] (Xtext Type System), 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

[17] Xtext generates an ANTLR parser [50], which uses an LL(*) algorithm. Although such parsers cannot deal with left recursive grammars directly, they have nice mechanisms for error recovery, which are essential in an IDE to provide a better feedback to the programmer.

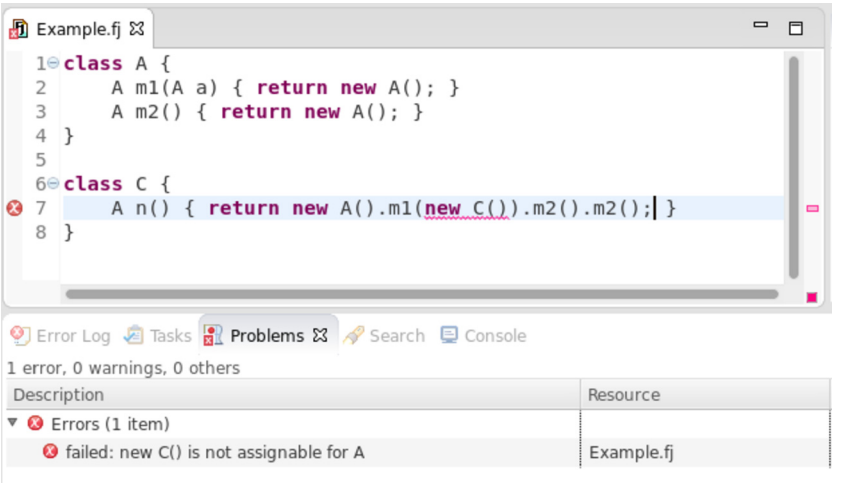


**Fig. 8.** Only important error reported.

purpose languages. Indeed, it is not straightforward to write the type system for FJ and λ-calculus in XTS. Moreover, XTS targets type systems only.

A typical way of declaring constraints for models, including EMF models, is to use OCL (Object Constraint Language) [67,49]. OCL is an expression language, while Xsemantics is based on rules. Furthermore, while OCL is suitable for specifying constraints, it might be hard to use to perform type inference and reduction rules.

There are other tools for implementing DSLs and IDE tooling (we refer to [65,52,26] for a wider comparison). Tools like IMP (the IDE Meta-Tooling Platform) [18] and DLTK (Dynamic Languages Toolkit) [1] only deal with IDE features. TCS (Textual Concrete Syntax) [40] is similar to Xtext, but with the latter it is easier to describe the abstract and concrete syntax at once, and it is completely open to customization of every part of the generated IDE (besides, TCS seems to be no longer under active development). EMFText [34], instead of deriving a metamodel from the grammar, does the opposite, i.e., the language to be implemented must be defined in an abstract way using an EMF metamodel. In that respect, Xtext enables a quicker and easier setup, since the AST metamodel is automatically inferred from the grammar. However, Xtext also allows the developer to switch to a manually maintained EMF metamodel.

Spoofax [42], another language workbench which targets Eclipse, does not require Java code for the analysis of a program, since it relies on Stratego [14] for rule-based specifications. In [63], Spoofax is extended with a collection of declarative meta-languages in order to create a language designer's workbench that supports all the aspects of language implementation including verification infrastructure and interpreters. These meta-languages include NaBL [44] for name binding and scope rules, TS for the type system and DynSem [62] for the operational semantics. Xsemantics shares with the mentioned systems the goal of reducing the gap between the formalization and the implementation.

The binding specification mechanism of [44] has been generalized and completely formalized in [48], where a theory for name binding and resolution is presented, to deal with scoping in programming languages. We are currently working on adding the possibility of specifying scoping rules in an Xsemantics specification as well. Such rules can easily access judgments and auxiliary functions defined in the Xsemantics system. This way, also the Xtext scope provider, which depends on the type system, can be easily generated automatically by Xsemantics.

Neverlang [61] allows the developer to plug and unplug programming language features and it also supports composition of specific Java constructs [17]. Similarly, JastAdd [25] supports modular specifications of extensible compiler tools and languages. Spoofax [42] provides support for language extensions and embeddings. Xtext only provides single inheritance mechanisms for grammars, so different grammars can be composed only linearly. The same holds for Xsemantics system extension (Section 6.2). These extensibility and compositionality features are not as powerful as the above mentioned frameworks, but we think they should be enough for implementing *pluggable type systems* [11].

EriLex [68] 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. MPS (Meta Programming System) [29] is another tool for developing a DSL. MPS targets *projectional editing* for building DSL editors with tables and diagrams.

We are also investigating the implementation of an automatic typesetting mechanism to produce a LaTeX or PDF document from an Xsemantics system specification, like other frameworks do, such as, e.g., [59,58,23,28].

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 [34]. Other approaches, such as, e.g., [10,29,12,23,28,68,61], 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* [56] and *Centaur* [10]. In both cases, the use of a DSL for the type system was also recognized (the latter was using several formalisms [41,47,22]).

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., [4,13,45] which are based on language specification preprocessors, [19,55] which target host language extensions and internal DSLs, [12,23,28] which do not target IDE tooling.

Xsemantics can be considered the successor of Xtypes [6]. In this respect, Xsemantics provides a much richer syntax for rules, thanks to Xbase. Premises in Xsemantics are written in a Java-like language with less "syntactic noise" and more advanced features (like lambda expressions), and can access any existing Java type. 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.

## 9. Conclusions

In this paper we described Xsemantics, a DSL for writing type systems, reduction rules (and in general relation rules) for languages implemented in Xtext (the type system of Xsemantics is implemented in Xsemantics itself). This paper extends the previous presentations [9,8] describing the main features of Xsemantics in more details, describing a new example, providing information about the general architecture of Xsemantics and presenting new features of Xsemantics, which were dictated by its adoption in real-world type system implementations.

In the Introduction we described a few aspects and goals that we think are important for implementing type systems, especially for languages that provide IDE support. In the rest of this section we evaluate how Xsemantics dealt with the above issues.

Our main goal was to reduce the gap between the formalization of a language and its implementation. With respect to manual implementations of type systems and reduction rules in Java, Xsemantics specifications are more compact and easier to maintain. In spite of resembling formal systems, Xsemantics syntax is still usable by developers who are not familiar with meta-theory. In our experience, we noted that even people with no previous type system experience, were able to quickly implement type system rules in Xsemantics within an hour.

This was possible also thanks to the complete integration of Xsemantics with Java, allowing the developers to implement specific tasks in Java and to delegate to these implementations from an Xsemantics specification (as shown in Section 6.1 and in some listings we showed in the paper).

Thanks to Xtext and Xbase, Xsemantics offers a rich Eclipse tooling. 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. Most importantly, with the Xsemantics IDE we can debug not only the generated Java code, but also the original Xsemantics rule definitions (Section 6.4). These IDE mechanisms make editing Xsemantics specifications easy, without giving up all the features that Eclipse JDT provides.

Moreover, we showed how Xsemantics takes into consideration pragmatic issues concerning efficiency and responsiveness of the integration of the type system into an IDE. In particular, Xsemantics caching mechanisms allow the developers to improve the performance of the type system, and in some cases, as shown in Section 6.3, to deal with corner cases for type inference.

We also discussed issues concerning error recovery that a type system implementation has to take into consideration to improve the user experience, especially in the IDE. In that respect, we provided hints and showed how to split type computation and type checking in the FJ example.

Summarizing, we think that Xsemantics has proved to be mature enough[18] to be successfully used also in the implementation of type systems of real-world languages in industry projects (see Section 1 and [35]). Such industrial projects also inspired and drove most adjustments and new features that are part of the current implementation of Xsemantics.

## Acknowledgements

I would like to thank the anonymous reviewers for their suggestions for improving the paper. As Xsemantics users, Jens von Pilgrim, Mark-Oliver Reiser, Andreas Heiduk and Steffen Skatulla provided useful feedback, many suggestions and proposals for new features. Sebastian Zarnekow helped me with many Xtext/Xbase internals, not to mention he contributed to Xsemantics performance improvements. A special thanks to Betti Venneri, for introducing me to the charming world of type systems.

## References

[1] DLTK, http://www.eclipse.org/dltk.

[18] 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).

[2] Xtext, http://www.eclipse.org/Xtext.
[3] Draft ECMAScript language specification. Working draft ECMA-262, 6th edition, Rev. 24, Apr. 27, 2014, ISO/IEC, Apr. 2014.
[4] D. Batory, B. Lofaso, Y. Smaragdakis, JTS: tools for implementing domain-specific languages, in: ICSR, IEEE, 1998, pp. 143–153.
[5] Y. Bertot, P.P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Texts Theoret. Comput. Sci., Springer, 2004.
[6] L. Bettini, A DSL for writing type systems for Xtext languages, in: PPPJ, ACM, 2011, pp. 31–40.
[7] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing, 2013.
[8] L. Bettini, Implementing Java-like languages in Xtext with Xsemantics, in: OOPS (SAC), ACM, 2013, pp. 1559–1564.
[9] L. Bettini, D. Stoll, M. Völter, S. Colameo, Approaches and tools for implementing type systems in Xtext, in: SLE, in: Lect. Notes Comput. Sci., vol. 7745, Springer, 2012, pp. 392–412.
[10] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, CENTAUR: the system, in: Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Not. 24 (1988) 14–24.
[11] G. Bracha, Pluggable type systems, in: Workshop on Revival of Dynamic Languages, 2004.
[12] M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM TOPLAS 24 (4) (2002) 334–368.
[13] M. Bravenboer, R. de Groot, E. Visser, MetaBorg in action: examples of domain-specific language embedding and assimilation using Stratego/XT, in: GTTSE, in: Lect. Notes Comput. Sci., vol. 4143, Springer, 2006, pp. 297–311.
[14] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Sci. Comput. Program. 72 (1–2) (2008) 52–70.
[15] N. Cameron, E. Ernst, S. Drossopoulou, Towards an existential types model for Java wildcards, in: Formal Techniques for Java-like Programs (FTfJP) 2007, July 2007.
[16] L. Cardelli, Type systems, ACM Comput. Surv. 28 (1) (1996) 263–264.
[17] W. Cazzola, E. Vacchi, Neverlang 2: componentised language development for the JVM, in: Software Composition, in: Lect. Notes Comput. Sci., vol. 8088, Springer, 2013, pp. 17–32.
[18] P. Charles, R. Fuhrer, S. Sutton Jr., E. Duesterwald, J. Vinju, Accelerating the creation of customized, language-specific IDEs in Eclipse, in: OOPSLA, ACM, 2009, pp. 191–206.
[19] T. Clark, P. Sammut, J. Willans, Superlanguages, Developing Languages and Applications with XMF, 1st edition, Ceteva, 2008.
[20] L. Damas, R. Milner, Principal type schemes for functional programs, in: POPL, ACM, 1982, pp. 207–212.
[21] Dart Team, Dart Programming Language Specification, 1.2 edition, Mar. 2014.
[22] T. Despeyroux, Typol: a formalism to implement natural semantics, Technical Report 94, INRIA, Mar. 1988.
[23] A. Dijkstra, S.D. Swierstra, Ruler: programming type rules, in: FLOPS, in: Lect. Notes Comput. Sci., vol. 3945, Springer, 2006, pp. 30–46.
[24] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, R. von Massow, Xbase: implementing domain-specific languages for Java, in: GPCE, ACM, 2012, pp. 112–121.
[25] T. Ekman, G. Hedin, The JastAdd system – modular extensible compiler construction, Sci. Comput. Program. 69 (1–3) (2007) 14–26.
[26] 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, J. van der Woning, Evaluating and comparing language workbenches: existing results and benchmarks for the future, Comput. Lang. Syst. Struct. 44A (2015), http://dx.doi.org/10.1016/j.cl.2015.08.007.
[27] M. Eysholdt, H. Behrens, Xtext: implement your language faster than the quick and dirty way, in: SPLASH/OOPSLA Companion, ACM, 2010, pp. 307–309.
[28] M. Felleisen, R.B. Findler, M. Flatt, Semantics Engineering with PLT Redex, The MIT Press, Cambridge, MA, 2009.
[29] M. Fowler, A language workbench in action – MPS, http://martinfowler.com/articles/mpsAgree.html, 2008.
[30] 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.
[31] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 7 edition, Addison-Wesley, 2013.
[32] J. Hage, B. Heeren, Strategies for solving constraints in type and effect systems, in: VODCA, in: Electron. Notes Theor. Comput. Sci., vol. 236, Elsevier, 2009, pp. 163–183.
[33] B. Heeren, J. Hage, S.D. Swierstra, Scripting the type inference process, in: Eighth International Conference on Functional Programming, ACM Press, 2003, pp. 3–13.
[34] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende, Derivation and refinement of textual syntax for models, in: ECMDA-FA, in: Lect. Notes Comput. Sci., vol. 5562, Springer, 2009, pp. 114–129.
[35] A. Heiduk, S. Skatulla, From Spaghetti to Xsemantics – practical experiences migrating typesystems for 12 languages, XtextCon, http://www.xtextcon.org/slides/2015-05-20_XtextCon-Heiduk_Skatulla_Xsemantics.pdf, 2015.
[36] A. Hejlsberg, S. Lucco, TypeScript Language Specification, 1.0 edition, Microsoft, Apr. 2014.
[37] J.R. Hindley, Basic Simple Type Theory, Cambridge University Press, 1987.
[38] A. Igarashi, H. Nagira, Union types for object-oriented programming, J. Object Technol. 6 (2) (2007).
[39] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM TOPLAS 23 (3) (2001) 396–450.
[40] F. Jouault, J. Bézivin, I. Kurtev, TCS: a DSL for the specification of textual concrete syntaxes in model engineering, in: GPCE, ACM, 2006, pp. 249–254.
[41] G. Kahn, B. Lang, B. Melese, E. Morcos, Metal: a formalism to specify formalisms, Sci. Comput. Program. 3 (2) (1983) 151–188.
[42] L.C.L. Kats, E. Visser, The Spoofax language workbench. Rules for declarative specification of languages and IDEs, in: OOPSLA, ACM, 2010, pp. 444–463.
[43] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J.A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, R.B. Findler, Run your research: on the effectiveness of lightweight mechanization, in: POPL, ACM, 2012, pp. 285–296.
[44] G. Konat, L. Kats, G. Wachsmuth, E. Visser, Declarative name binding and scope rules, in: SLE, in: Lect. Notes Comput. Sci., vol. 7745, Springer, 2012, pp. 311–331.
[45] H. Krahn, B. Rumpe, S. Völkel, Monticore: a framework for compositional development of domain specific languages, Int. J. Softw. Tools Technol. Transf. 12 (5) (2010) 353–372.
[46] J. Levine, Flex & Bison, O'Reilly Media, 2009.
[47] E. Morcos-Chounet, A. Conchon, PPML: a general formalism to specify prettyprinting, in: IFIP Congress, 1986, pp. 583–590.
[48] P. Neron, A.P. Tolmach, E. Visser, G. Wachsmuth, A theory of name resolution, in: ESOP, in: Lect. Notes Comput. Sci., vol. 9032, Springer, 2015, pp. 205–231.
[49] Object Management Group, Object Constraint Language, version 2.2, OMG document number: formal/2010-02-01 edition, http://www.omg.org/spec/OCL/2.2, 2010.
[50] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Programmers, 2007.
[51] L.C. Paulson, Isabelle: A Generic Theorem Prover, Lect. Notes Comput. Sci., vol. 828, Springer, 1994.
[52] M. Pfeiffer, J. Pichler, A comparison of tool support for textual domain-specific languages, in: Proc. DSM, 2008, pp. 1–7.
[53] B.C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, MA, 2002.
[54] D.R. Prasanna, Dependency Injection: Design Patterns Using Spring and Guice, 1st edition, Manning, 2009.
[55] L. Renggli, M. Denker, O. Nierstrasz, Language boxes: bending the host language with modular language changes, in: SLE, in: Lect. Notes Comput. Sci., vol. 5969, Springer, 2009, pp. 274–293.

[56] T. Reps, T. Teitelbaum, The synthesizer generator, in: Software Engineering Symposium on Practical Software Development Environments, ACM, 1984, pp. 42–48.
[57] J.A. Robinson, Computational logic: the unification computation, Mach. Intell. 6 (1971).
[58] G. Rosu, T.-F. Serbanuta, An overview of the K semantic framework, J. Log. Algebr. Program. 79 (6) (2010) 397–434.
[59] P. Sewell, F.Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strnisa, Ott: effective tool support for the working semanticist, J. Funct. Program. 20 (1) (2010) 71–122.
[60] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd edition, Addison-Wesley, 2008.
[61] E. Vacchi, W. Cazzola, Neverlang: a framework for feature-oriented language development, Comput. Lang. Syst. Struct. 43 (3) (2015) 1–40.
[62] V.A. Vergu, P. Neron, E. Visser, DynSem: a DSL for dynamic semantics specification, in: RTA, in: LIPIcs, vol. 36, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378.
[63] E. Visser, G. Wachsmuth, A.P. Tolmach, P. Neron, V.A. Vergu, A. Passalaqua, G. Konat, A language designer's workbench: a one-stop-shop for implementation and verification of language designs, in: Onward!, ACM, 2014, pp. 95–111.
[64] M. Voelter, Language and IDE modularization and composition with MPS, in: GTTSE, in: Lect. Notes Comput. Sci., vol. 7680, Springer, 2011, pp. 383–430.
[65] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L.C.L. Kats, E. Visser, G. Wachsmuth, DSL Engineering – Designing, Implementing and Using Domain-Specific Languages, 2013.
[66] M. Völter, Xtext/TS – a typesystem framework for Xtext, http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/, 2011.
[67] J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1999.
[68] H. Xu, EriLex: an embedded domain specific language generator, in: TOOLS, in: Lect. Notes Comput. Sci., vol. 6141, Springer, 2010, pp. 192–212.
[69] S. Zarnekow, Xtext best practices, http://www.eclipsecon.org/europe2012/sessions/xtext-best-practices, 2012.