**Research Article** **Open Access**

Lorenzo Bettini*

# Type errors for the IDE with Xtext and Xsemantics

https://doi.org/10.1515/comp-2019-0003
Received June 29, 2018; accepted February 11, 2019

**Abstract:** Providing IDE support for a programming language or a DSL (Domain Specific Language) helps the users of the language to be more productive and to have an immediate feedback on possible errors in a program. Static types can drive IDE mechanisms such as the content assist to propose sensible completions in a given program context. Types can also be used to enrich other typical IDE parts such as the Outline and the Hovering pop-ups. In this paper, we focus on statically typed imperative languages, adopting some form of type inference. We present a few general patterns for implementing efficient type systems, focusing on type error recovery. This way, the type system is able to type as many parts of the program as possible, keeping a good IDE experience. Type error messages will be placed on the important parts of the program, avoiding cascading errors that can confuse the user. We show two case studies: we apply the presented patterns to implement the type system of two statically typed DSLs, a simple expression language and a reduced Java-like language, with OOP features. We use Xtext as the language workbench for implementing the compiler and the IDE support and Xsemantics, a DSL for implementing type systems using a syntax that mimics formal systems. The patterns shown in the paper can be reused also for implementing languages with other language frameworks.

**Keywords:** language implementation, type system, IDE

## 1 Introduction

Integrated Development Environments (IDEs) help programmers with mechanisms like syntax aware editor (syntax highlighting), immediate error reporting, code completion (also known as content assist) and easy navigation to declarations. Providing IDE support for a language or a DSL (Domain Specific Language) enhances the user experience and contributes to the adoption of that language. However, developing a compiler and an IDE for a language from scratch is usually time consuming. *Language workbenches*, a term coined by Martin Fowler in 2005 [1], are software development tools designed to easily implement languages together with their IDE support. Xtext [2] is one of the most popular language workbench. Starting from a grammar definition Xtext generates a parser, an abstract syntax tree (AST), and typical IDE mechanisms. While the main target of Xtext is Eclipse, a language developed with Xtext can be easily ported to other IDEs and editors. Xtext comes with good defaults for all the above artifacts, directly inferred from the grammar definition, and the language developer can easily customize them all.

Statically typed languages provide good IDE support. Indeed, given an expression and its static type, the editor can provide useful completion proposals that make sense in that program context. The same holds for other typical IDE features, e.g., navigation to declaration, hovering and outline. In spite of the ease of implementing a language and its IDE support with a language workbench like Xtext, the implementation of the type system requires some effort and careful tweaking:

- The type system should type as many parts of a program as possible, so that the IDE can provide meaningful code completions even in the presence of an incomplete and invalid program.
- Type error messages should be placed on the important parts of the program, helping the user to quickly understand and fix the errors, avoiding cascading errors that would only confuse the user.
- Type computation should be performed quickly, in order to keep the IDE responsive.

In this paper we concentrate on mechanisms related to the implementation of type systems addressing the above issues, focusing on type errors and IDE support. The contributions of the paper are:

- We describe some general implementation patterns for implementing type systems that aim at reporting only the important errors, with useful error messages, keeping the IDE responsive. In particular, we concentrate on imperative languages, adopting some form of type inference.

***Corresponding Author: Lorenzo Bettini:** Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, Italy;
E-mail: lorenzo.bettini@unifi.it

- We then demonstrate the benefits of such patterns by implementing two example DSLs in Xtext: a simple "Expressions DSL" (with arithmetic, string and boolean expressions) and a reduced version of Java, based on Featherweight Java (FJ) [3], a well-known formal framework for studying properties of the Java language and for introducing extensions to Java.
- For both examples, we show the benefits of applying the described patterns, in terms of the user experience.

Both languages[1] are simple languages but their type systems are complex enough to help us investigate best practices in type system implementation. We implement the type system of the first DSL manually, while for FJ we use Xsemantics [4], a DSL for implementing type systems for Xtext languages. Xsemantics provides many features for implementing complex type systems, like the one of FJ (in spite of its simplicity, the type system of FJ formalizes the main parts of the Java type system). Moreover, Xsemantics provides a syntax that is close to formal type systems and this makes it suitable to implement type systems that have been formalized, like FJ's type system [3]. The syntax of the two example DSLs is partly inspired by our previous work [2, 4], but the implementations shown in this paper are new.

The patterns presented in the paper are not strictly dependent on Xtext, Xsemantics and Eclipse: these are the frameworks that are used in the paper to present our case studies. The same patterns could be applied to other language workbenches. The basic idea of the implementation patterns will be given independently from these frameworks.

The intended audience of the paper is developers of statically typed languages and DSLs with IDE support. In particular, we will use Xtext as the language workbench and Eclipse as the target IDE. A basic knowledge of language implementation and type systems is advisable. We will provide enough details on Xtext in order to make the examples understandable even by readers who are not familiar with Xtext.

The paper is structured as follows. Section 2 motivates our work and presents our general implementation patterns for type systems. Section 3 provides a brief introduction to Xtext and its main mechanisms using the Expressions DSL as a running example. Section 4 illustrates our first case study, showing the implementation of the type system for the Expressions DSL. Section 5 shows our second case study, that is, the implementation of the type system of an OO language, Featherweight Java. Section 6 discusses further improvements to the type system implementations of the two DSLs. Section 7 discusses a few related works. Section 8 concludes the paper.

## 2 General implementation patterns

A crucial issue to keep in mind when implementing type systems is *error recovery*. This is similar to error recovery in parser implementations [5]: a parser should be able to handle possible parsing errors and carry on parsing as much of the rest of the input as possible, avoiding or minimizing additional cascading errors. The same holds for the type system: it should be able to type and check as many parts of a program as possible, even in the presence of type errors. Thus, the type system should avoid throwing cascading errors due to previously failed type computations. This is important in a command line compiler, but in the IDE it is even more crucial: we should minimize the portions of the edited file marked with errors. This way, the user can easily understand the errors and quickly fix them. In particular, the type system will be continuously used by the IDE while the user is editing the program. Thus, the program that is examined by the type system will be an incomplete program most of the time.

In this section we present some general implementation patterns for implementing type systems that aim at reporting only the important errors, with useful messages and at providing a good user experience in the IDE.

We consider the implementation of a type system as the synergy of two main components: the *type inference*, that is, the mechanism responsible of computing the types of the program elements, and the *type checker*, that is, the mechanism responsible of validating the program with respect to types. These two components will also have to cooperate with the mechanism for cross reference resolution.

**Infer a type as quickly as possible.** Given a composite expression, if its type can be computed independently from its subexpressions, the subexpressions should not be inspected recursively. This has the benefit of improving error recovery and the type inference performance. For example, let us consider an arithmetic expression of the shape `e1 * e2`[2]. We can infer that the type of the expres-

1 The source code of the implemented DSLs presented in this paper can be found at: https://github.com/LorenzoBettini/xtext-type-errors-examples.

2 Where $*$ is assumed to be the standard multiplication operator.

sion is numeric, independently from the types of subexpressions. If one of the subexpressions has a non-numeric type, the whole expression is invalid. However, the validity check should be performed by the type checker, in a separate component, as detailed in the next point. Sometimes, depending on the semantics of the language, it might not be possible to avoid inspecting subexpressions to compute the type for the whole expression. For example, let us assume that in the language the operator + is overloaded and represents both the arithmetic sum and the string concatenation (like in the first case study, Section 4). The type inference needs to first infer the types of subexpressions before computing the resulting type of `e1 + e2`.

We will see a few examples in Sections 4.2 and 5.5.

**Keep type inference and type checking separate.** It is also crucial to keep type inference and type checking separate in the type system implementation. Mixing the two mechanisms could allow the language developer to implement a prototype of the type system quickly, but in the presence of type errors, the type inference could not compute types for the rest of the program. Going back to the example above, `e1 * e2`, if type inference and type checking are implemented as a single mechanism, a type error in one of the subexpressions would prevent the type system from assigning a type to the whole expression. This would also lead to cascading errors in many other parts of the program. Instead, from the type inference point of view, we can still assign a numeric type to the expression. Even if one of the subexpressions has a non-numeric type and thus the whole expression is invalid, we can still infer that the type of the expression is numeric. Being able to assign a type to such an expression, even in the presence of a type error, will allow the type system to type other parts of the program. Besides avoiding cascading errors, a type system with error recovery can also detect type errors in other parts of the program which could be useful for the user. This pattern was already applied in openArchitectureWare [6], and inspired applications in other frameworks such as Spoofax [7], see for example [8].

We will see a few examples in Sections 4.5 and 5.5.

**Useful type error reports.** Separating type inference from type checking allows us to spot the crucial type errors in the program. This way, we are able to create useful type error messages and to report them on the important parts of the program. The user is then able to fix the errors easily. For example, if in the expression `e1 * e2` the type of `e2` is not conformant with the one expected by `*` we can mark only `e2` with an error. The error message will clearly state that the actual type of `e2` does not match with the type expected by `*`. Since the type inference is still able to give a type to the whole expression, then the whole expression will not be marked with an error. If we mixed type inference and type checking, we would issue an error on the whole expression stating that it cannot be typed. This would not be helpful. We refer to Section 7 for a review of related works on type error reporting.

We will see a few examples in Sections 4.5, 5.4 and 5.5.

**Error recovery and cross references.** In a language implementation, cross reference resolution might be strictly related to the type system. For example, in an Object-Oriented language, in order to resolve a member reference in a member selection expression (e.g., a method invocation on an object), we first need to compute the type of the receiver expression. If the type inference is separate from type checking, we will be able to type a receiver expression even in the presence of errors. This allows us to resolve the member. Otherwise, besides an error on the whole receiver expression, there would be an error also on the member reference that cannot be resolved. An example will be shown in Section 5.5.

**Visibility and validity.** When computing the possible candidates for cross reference resolution, we should avoid filtering out candidates that would not be valid because of some semantic rules of the language. This means that we should not mix "visibility" and "validity": a declared element can be visible in a program context even if its use is not valid in that context. Cross reference resolution should only deal with visibility, and validity should be checked by another component of the language implementation. For example, if we tried to refer to a local variable that is never declared, cross reference resolution should actually fail. On the contrary, if we tried to refer to a local variable that is declared after the reference, we could still resolve the reference; then, another component will check if the reference is valid. In this example, the error reported by the validation component would be more informative, marking the reference as an invalid "forward reference", rather than a more general "unresolved reference" (an example is shown in Section 4.5). Another typical example, in an Object-Oriented language, is trying to access a private field from another class. We should not mark the reference as unresolvable: it is more useful to report that in that program context that field is not accessible (an example is shown in Section 5.4).

**Cache computed types.** In the type system implementation, the type of the same expression could be used in many places, especially in our context, where the components of the type system are kept separate. For this reason, the results of type computations could be cached to improve the performance so that the IDE is kept responsive. We will see a few examples in Section 6.

**Content assist and valid proposals.** The content assist should help the user by proposing completions that are sensible in a given program context. A proposal is sensible if it does not make the program invalid. Thus, the content assist should rely on the type system for filtering out proposals that would generate a type error in the program. In particular, it could perform the filtering by relying on the type checker. We will see a few examples in Sections 4.6 and 5.4.

**Show types in the IDE.** Especially when the language supports a form of type inference, where types are not declared by the user, the IDE should show the inferred types, for example, in the editor itself, in the outline, when the user hovers on a program part, etc. This way, the user can understand how the type system computes the types, and can then easily debug a type error. A few examples are shown in Section 4.6. If the type system enjoys error recovery, the IDE can then show information about types even in a program with type errors.

The patterns described in this section should lead to an efficient type system with error recovery aiming at helping the user to easily understand possible type errors and to quickly fix them. Moreover, such an implementation of the type system should also allow the language developer to implement useful IDE mechanisms, like, the content assist, code navigation (e.g., from a reference to the resolved declaration) and other features.

Finally, keeping the mechanisms of a type system separate allows us to have a modular implementation consisting of several components, loosely coupled, which are easily testable in isolation. The implementation of the DSLs presented in this paper (available at https://github.com/LorenzoBettini/xtext-type-errors-examples) contains all the JUnit tests for all the aspects (from the type system up to the UI mechanisms, like, e.g., the content assist). Indeed, the examples presented in this paper have been implemented with a test-driven development approach. Such a modular architecture also leaves the door open to an easy replacement of each single component.

In the next sections, we will apply the patterns described above to the implementation of the type systems of our DSL case studies.

For the Expressions DSL (Section 4) we will first informally define its type system and then we will proceed with its implementation keeping the aspects of the type system separate, showing how we can easily deal with error recovery thanks to our modular implementation. This allows us to show useful error messages, avoiding cascading errors, and to enrich the Eclipse tooling with type information, even in the presence of type errors.

For FJ (Section 5), instead, we first implement the type system with Xsemantics following the formalization of FJ of [3]. Unfortunately, formal type systems typically deal with type inference and type checking at the same time. Thus, this first implementation would not benefit from the advantages of the patterns described in this section. We then apply such patterns to improve the type system implementation in order to have a better error recovery mechanism and to provide better error messages.

The patterns presented in this section are meant to be general enough to be applied also when using other language workbenches, and are not strictly dependent on Xtext, Xsemantics and Eclipse. In this paper, we apply them using these frameworks and we adapt them to the rules and lifecycles of these frameworks. Xtext makes such an adaption easy because it hides most of the internal details of Eclipse components. An example is error marker generation: in Xtext it is enough to call an API method, and the Eclipse error markers will be created automatically by Xtext in all the Eclipse parts (Section 3.3). Moreover, Xtext lifecycle automatically executes most of the compilation mechanisms (Section 3.4) applying good defaults and delegating to possible custom implementations (the typical example is cross reference resolution in Xtext, Section 3.2).

Summarizing, the code of the examples implemented in the next sections might not be used as it is in other frameworks. The implementation patterns can however still be applied. How easy this can be done highly depends on the features of the language workbench and it is out of the scope of the paper.

# 3 Small introduction to Xtext

In this section we provide a brief introduction to Xtext[3], using a simple DSL for expressions that we call "Expressions DSL". This will also be the example of our first case study. In the Expressions DSL, programs consist of *variable declarations* with an initialization expression and of *evaluations statements*. The syntax of variable declarations is `var name = exp`. The syntax of evaluation statements is `eval exp`. Expressions can perform standard arithmetic operations, compare expressions, use logical connectors, and concatenate strings. Expressions can also refer to variables. We will use + both for representing arithmetic addition and for string concatenation (in this case, subexpressions are meant to be automatically converted to strings).

3 https://www.eclipse.org/Xtext/

The types used in this DSL are only integer, string and boolean types. In order to make the example more interesting, types of variables are not declared explicitly and they are automatically inferred from the initialization expression.

We will provide enough details on Xtext in order to make the examples understandable even by readers who are not familiar with Xtext. We then refer to [2] for full details on Xtext.

Xtext is a *language workbench*: the developer writes a grammar definition and starting from this definition Xtext generates a parser, an abstract syntax tree (AST), and a fully-fledged Eclipse editor with syntax highlighting, navigation, content assist and outline. Xtext also generates an automatic Eclipse builder. Xtext relies on EMF (Eclipse Modeling Framework) for the generation of the AST. EMF [9] is a modeling framework for representing and manipulating structured data models. From a model specification (the *metamodel*) EMF provides tools for code generation mechanisms and runtime support to produce a set of Java types (interfaces and classes), factories for the model, and a set of classes with observable/observer mechanisms on the model. Xtext takes care of creating the metamodel and generate the Java code. The DSL developer only needs to know the EMF conventions about the generated Java code and the EMF general API for traversing an EMF model (i.e., in this context, the AST). We partly introduce the main EMF features in the rest of this section to make the code snippets understandable.

## 3.1 Grammar

An Xtext grammar is specified using an EBNF-like syntax. The Xtext grammar for the Expressions DSL is shown in Figure 1. Before getting into the details of an Xtext grammar, we show in Figure 2 the editor generated by Xtext starting only from the grammar in Figure 1. Besides the syntax highlighting, a default content assist is also generated based on the grammar: after the "+" it proposes also the variables defined in the program. We will later customize this content assist (Section 4.6).

In an Xtext grammar, a rule is defined using a sequence of terminals (that is, quoted strings) and non-terminals (that is, names of other rules). For example, in the rule `Variable`, `"var"` and `'='` are terminals while `ID` and `Expression` are non-terminals:

```
Variable:
  "var" name=ID '=' expression=Expression;
```

```
grammar
  org.example.expressions.Expressions
  with org.eclipse.xtext.common.Terminals

generate
  expressions
  "http://www.example.org/expressions/Expressions"

ExpressionsModel:
  elements+=AbstractElement*;

AbstractElement:
  Variable | EvalExpression;

Variable:
  "var" name=ID '=' expression=Expression;

EvalExpression:
  "eval" expression=Expression;

Expression: Or;

Or returns Expression:
  And (
    {Or.left=current} "||" right=And
  )*;

And returns Expression:
  Equality (
    {And.left=current} "&&" right=Equality
  )*;

Equality returns Expression:
  Comparison (
    {Equality.left=current} op=("==" | "!=")
    right=Comparison
  )*;

Comparison returns Expression:
  PlusOrMinus (
    {Comparison.left=current} op=(">=" | "<=" | ">" | "<")
    right=PlusOrMinus
  )*;

PlusOrMinus returns Expression:
  MulOrDiv (
    ({Plus.left=current} '+' | {Minus.left=current} '-')
    right=MulOrDiv
  )*;

MulOrDiv returns Expression:
  Primary (
    ({MulOrDiv.left=current} op=('*' | '/'))
    right=Primary
  )*;

Primary returns Expression:
  '(' Expression ')' |
  {Not} "!" expression=Primary |
  Atomic;

Atomic returns Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true' | 'false') |
  {VariableRef} variable=[Variable];
```

**Figure 1:** The Xtext grammar of the Expressions DSL.

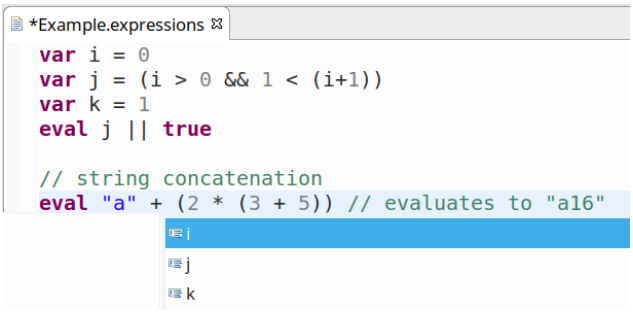


**Figure 2:** The editor generated for the Expressions DSL.

Alternatives are marked by a pipe `|`. For example, an `AbstractElement` can be either a `Variable` or an `EvalExpression`:

```
AbstractElement:
  Variable | EvalExpression;
```

In the grammar rules, assignment operators define how the AST is to be constructed during parsing. The identifier on the left-hand side refers to a property of the AST class (which is also generated by Xtext). Since the AST is generated in the shape of an EMF model [9], we will use the EMF terminology for "property", that is, *feature*. The EMF model representing the parsed AST, is contained in memory in an EMF *resource*. EMF resources are in turn contained in a *resource set*. The right-hand side is a terminal or non-terminal. When a feature in the AST is a list, the operator `+=` is used in assignments, see, for example `elements` in `ExpressionsModel`:

```
ExpressionsModel:
  elements+=AbstractElement*;
```

The standard operators can be used for the cardinality: `?` for zero or one, `*` for zero or more and `+` for one or more.

The `with` specification at the beginning of the grammar represents the *language inheritance* support provided by Xtext: similar to Object-Oriented inheritance, the rules of the parent grammar will be available in the inheriting grammar. For example the rule for a standard identifier, `ID`, used in the rule `Variable`, is defined in the parent grammar `Terminals`, one of the base grammar provided by Xtext.

Square brackets on the right-hand side of an assignment define cross references to another element in the parsed model; the name in the brackets is the type of the referred element, which, by default, corresponds to the name of a rule. For example, `variable=[Variable]`, in the `Atomic` rule, represents in the AST a reference to an element of type `Variable`:

```
Atomic returns Expression:
  ...
  {VariableRef} variable=[Variable];
```

```
public interface AbstractElement {
  Expression getExpression();
  void setExpression(Expression value);
}

public interface Variable extends AbstractElement {
  String getName();
  void setName(String value);
}

public interface EvalExpression extends AbstractElement {
}

public interface VariableRef extends Expression {
  Variable getVariable ();
  void setVariable(Variable value);
}
```

**Figure 3:** The EMF Java interfaces generated for the rules `AbstractElement`, `Variable`, `EvalExpression` and `VariableRef`.

By default, cross references are parsed as identifiers. Cross reference resolution is handled automatically by Xtext and can be customized by the developer (as described in Section 3.2).

An Xtext grammar cannot be left-recursive since Xtext relies on LL-parsing algorithms, which are known to be very good at error recovering but cannot handle left recursion [5]. For this reason, rules that would be left recursive by nature, have to be written avoiding left recursion, according to the patterns prescribed by Xtext, known as *left factoring*; we refer the interested reader to [2] for all the details. We just mention here that rules with a lower precedence (like `Or`) must be defined in terms of rules with a higher precedence (like `And`).

During parsing, the AST is automatically generated by Xtext as an EMF model. Thus, we can manipulate the AST using all mechanisms provided by EMF itself. There is a direct correspondence between the rules of the grammar and the generated EMF model Java classes. For instance, in Figure 3 we show the EMF generated interfaces for the rules `AbstractElement`, `Variable`, `EvalExpression` and `VariableRef`. Note also the inferred inheritance relation: `Variable` and `EvalExpression` inherit from `AbstractElement` and the common structure is pulled up in the supertype. Finally, cross references are evident also in the structure of the generated Java interfaces: the type of the `variable` feature in `VariableRef` is `Variable`. Internally, Xtext automatically keeps the EMF model representing the AST and the editor's contents in synchronization.

The code generated by Xtext comes with good and sensible defaults, thus, it can be used as it is for several

aspects of the implemented language. However, mechanisms like the type system of the language cannot be expressed in the grammar itself, and have to be implemented by the developer by customizing some classes used in the framework. The custom code is "injected" in the classes of the Xtext framework using Google-Guice [10], a *dependency injection* framework. The Java annotation `@Inject`, which we will use in the code snippets from now on, comes from Google-Guice.

The next subsections describe the two complementary mechanisms of Xtext, scoping and validation, respectively, which the developer typically has to customize. We will see how they relate to the mechanisms of a type system. Scoping and validation together implement the mechanism for checking the correctness of a program. These two components are kept separate to facilitate a modular implementation of a language. This separation can be found in other approaches as well, such as, e.g., [11–14], where the concept of scoping is referred to as *binding*. We will also describe the stages that Xtext executes and when the above mechanisms come into play.

## 3.2 Scoping

Xtext automatically deals with resolving cross references, that is, binding references to the original definitions. When defining an Xtext grammar, as shown in the previous section, rules specify cross references in the AST by using `[]`.

By default, Xtext binds references based on *containment* relations [9] in the EMF model of the AST. This strategy works as expected in some simple cases: a parameter reference occurring in a function body is bound to the parameter declaration in the signature of the containing function. In other situations this will not suffice. For example, in FJ (Section 5) field and method references must take into account the inheritance relation among classes, which is not modeled as a containment relation.

In order to customize the binding of cross references, in Xtext it is enough to customize the concept of *scope*, that is, the collection of all the elements that are "visible" in the current context of a reference. This is achieved by providing a custom `ScopeProvider`. When Xtext needs to resolve a symbol, it will use the scope returned by such a `ScopeProvider`. Using the returned scope, Xtext automatically resolves cross references or issue an error in case the reference cannot be resolved. Note that Xtext automatically handles cross references among different files and import mechanisms, which, however, are out of the scope of the current paper.

```
public class ExpressionsExampleValidator
             extends AbstractExpressionsValidator {
  @Check
  public void checkVariableNameLowercase(Variable v) {
    if (!Character.isLowerCase(v.getName().charAt(0)))
      warning("Variable name should start with a lowercase",
          v, null);
  }
}
```

**Figure 4:** An example of a `@Check` method.

Once cross references are resolved, the editor generated by Xtext automatically provides navigation mechanisms ("Navigate to symbol"). Moreover, the scope provider is also automatically used by the content assist. Thus, the developer customizes only the single concept of scope and the IDE mechanisms automatically work accordingly.

For FJ (Section 5), we implement the scope provider for field and method references by using the type of the receiver expression. Thus, the scope provider is implemented using the type inference.

## 3.3 Validation

Apart from cross reference resolution, which is part of the validation of a program, all the other semantic checks that are not expressable in the grammar, are delegated by Xtext to a *validator*.

The programmer provides a custom `AbstractDeclarativeValidator` and Xtext automatically calls the methods in this class annotated with `@Check` for validating the model according to the runtime type of the AST node. The validation automatically takes place in the background, while the user of the DSL is writing in the editor; an immediate feedback is provided to the user.

The DSL developer can call the methods `error` and `warning` of the validator base class, providing messages about an issue and the offending element in the AST. Xtext then automatically generates the appropriate error markers in the Eclipse workbench (e.g., in the editor, in the "Problems" view and in the "Project Explorer" view). In Figure 4 we show a possible validation for the Expressions DSL: if a variable's name does not start with a lowercase we issue a warning.

In a statically typed DSL, validation is strictly connected to the type system implementation. This implies that first types have to be computed, especially when the DSL allows types to be inferred, as in the Expressions DSL. If a type cannot be computed for a given program term, an

error should be issued. If types can be computed, then *type conformance* must be checked. Type conformance deals with checking whether an expression has a type that conforms to the type expected in a given program context.

The visit of the AST is handled by Xtext, which traverses the tree and call the appropriate `@Check` methods, according to the actual type of the AST node. Finding the right program context to validate a term depends on the semantics of the DSL. Choosing the right context for validation also allows the DSL developer to provide useful information on type errors and to report errors on the relevant parts of the program.

Note that Xtext provides some predefined validation checks that can be enabled for the DSL, like checking that names are unique in the program. For the DSLs presented in this paper, this validation mechanism can be used out of the box and we do not have to manually check for duplicate names.

## 3.4 Xtext stages

In this section we briefly describe the stages executed internally by Xtext, in order to clarify when the scope provider and the validator are used.

1. The textual program is parsed and the AST is created as an EMF model, stored in a resource (in turn, contained in a resource set). During this stage, cross references are not resolved. Instead, EMF *proxies* are used to represent cross references. EMF proxies are placeholders where Xtext stores information for later cross reference resolution, e.g., the "name" of the referred element.
2. Before the validation, the resource is traversed to resolve cross references (resolution of proxies). If the proxy resolution logic traverses cross references of the model, that are yet unresolved, it transparently triggers resolution there too. As mentioned before, Xtext automatically handles cross references among different files. During this stage possible cross references to external resources' contents are resolved as well, traversing other resources in the resource set. As described in Section 3.2, when Xtext resolves a cross reference, it queries the scope provider for possible candidates.
3. Finally, the validator is executed. Since unresolvable proxies are cached by Xtext, if there are yet unresolved cross references, the scope provider will not be used again. If the validation traverses other resources, proxy resolution is triggered on demand on the other resources. After validation has completed, possible unresolvable cross references are reported by Xtext automatically as errors.

Xtext keeps track of proxies that are currently being resolved and possible cycles during this resolution, due to bugs in the DSL implementation, are caught by Xtext, avoiding infinite loops.

The type system can be used both during scoping and validation, depending on how the language developer implements these two mechanisms. This implies, due to the above stages, that the type system itself can transparently trigger further proxy resolutions.

Finally, recall that the language developer does not customize the actual cross reference resolution: she customizes the scoping. Thus, the developer provides the candidates that are then used by Xtext to actually perform cross reference resolution. That is why it is hard to introduce bugs that lead to the above mentioned cycles.

## 3.5 Xtend

Xtext fosters the use of Xtend for implementing all the custom mechanisms of a DSL implementation. Xtend is a DSL for the JVM (implemented in Xtext itself). Xtend code is compiled directly into Java code. Xtend has a Java-like syntax removing most of the "verbosity" of Java. For example, terminating semicolons are optional and so are parenthesis in method invocations without arguments. Even `return` is optional in methods: the last expression is the returned expression (this applies also to branch instructions). Xtend is completely interoperable with the Java type system, thus any Java library can be reused from within Xtend. Xtend has a powerful type inference mechanism: types in declarations can be omitted when they can be inferred from the context (as in variable declarations with the syntax `val` or `var`, for final and non final variables, respectively). Xtend also includes enhanced switch expressions (i.e., with type guards and automatic casting) to avoid writing lengthy `instanceof` statements. Moreover, it provides *extension methods* that 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.

Xtend *lambda expressions* have the shape

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

where the types of parameters can be omitted if they can be inferred from the context.

In Xtend the equality operator == maps to the `equals` Java methods for objects. The triple operator must be used === for comparing object references.

Xtend itself is an Object-Oriented programming language, mimicking Java structures such as methods, fields, classes and interfaces, with the same semantics. Also in this case, Xtend removes "syntactic noise" adopting defaults: classes and methods are public by default and fields are private by default. Any declaration marked with the keyword `extension` allows the programmer to use its methods as extension methods.

We will use Xtend in this paper. Java programmers should have no problems in understanding the Xtend code presented in this paper.

# 4 Typing expressions

In this section we will sketch the type system implementation for the Expressions DSL that we introduced in the previous section. We apply the patterns described in Section 2. In spite of the simplicity of the Expressions DSL, we chose it as the first running example because typing expressions might be tricky to implement efficiently, especially when errors have to be reported in an IDE. Moreover, reporting relevant errors on the relevant parts of the program requires some additional effort.

## 4.1 Expressions DSL type system

First, we informally define the type system of this DSL. This informal presentation defines both the type computation rules (i.e., the type of each kind of expression) and the type checking rules (i.e., the expected types of the subexpressions):

1. constant expressions have the corresponding types as expected;
2. arithmetic expressions have type integer and require the subexpressions to have type integer, with the exception of `Plus`, as detailed in the next point;
3. a `Plus` requires the two subexpressions to have type integer. The only exception is the case when one of the subexpressions has type string: the whole expression is considered to have type string (since it is to be interpreted as the string concatenation operation); if they have both type integer, then the whole expression has type integer. Note that the semantics of `Plus` implies that any expression can be concatenated with a string, and integer and boolean expressions, in a string concatenation expression, are meant to be implicitly converted to strings;
4. boolean expressions require the subexpressions to have type boolean and the result has type boolean;
5. an `Equality` expression requires the two subexpressions to have the same type and the result has type boolean;
6. the same holds for a `Comparison`, but boolean expressions cannot be used in a `Comparison` expression;
7. the type of a variable is inferred from the initialization expression;
8. the type of a variable reference is the type of the referred variable, if the reference is valid (i.e., it refers to an actually declared variable and it is not a forward reference), or null otherwise.

The Expressions DSL comes with an interpreter and a code generator (that simply generates a textual file with the interpreted expressions); as usual, the interpreter must respect the static semantics, i.e., the type system, and must perform conversions accordingly. This is out of the scope of the current paper, and we refer the interested reader to the complete implementation of the example.

The shape of the AST, which reflects operator precedence, associativity and possible grouping by explicit parenthesis, is taken into consideration when computing types of subexpressions. However, by the rules informally defined above, the final computed type of an expression will not depend on the shape of the AST. For example, both `"a" + 1 + 2` and `"a" + (1 + 2)` will have type string. In particular, in the latter case, `(1 + 2)` has type integer but then the presence of the string `"a"` will assign the whole expression type string. On the contrary, the result of the evaluation will be different: `"a" + 1 + 2` evaluates to `"a12"`, while `"a" + (1 + 2)` evaluates to `"a3"`. Both evaluations respect the static semantics. Java string concatenation behaves the same way.

In the next section we implement this type system in the Xtend class `ExpressionsTypeSystem`. In particular, we separate the mechanisms for type computation (i.e., type inference), type conformance and type checking, as advised in Section 2.

## 4.2 Type inference

The code related to type inference is shown in Figure 5. From now on, we will use Xtend features extensively (Section 3.5). For example, in Figure 5, the expression `e.left.inferredType` uses the syntactic sugar for getters, the method `inferredType` is used as an extension

```
class ExpressionsTypeSystem {
  public static val STRING_TYPE = new StringType
  public static val INT_TYPE = new IntType
  public static val BOOL_TYPE = new BoolType

  @Inject extension ExpressionsModelUtil

  def isStringType(ExpressionsType type) {
    type === STRING_TYPE
  }
  def isIntType(ExpressionsType type) { // ... similar

  def ExpressionsType inferredType(Expression e) {
    switch (e) {
      // trivial cases
      StringConstant: STRING_TYPE
      IntConstant | MulOrDiv | Minus: INT_TYPE
      BoolConstant | Not | Comparison
          | Equality | And | Or: BOOL_TYPE
      Plus: { // recursive case
        val leftType = e.left.inferredType
        val rightType = e.right.inferredType
        if (leftType.isStringType || rightType.isStringType)
          STRING_TYPE
        else
          INT_TYPE
      }
      VariableRef: {
        if (e.isVariableDefinedBefore) // avoid infinite recursion
          e.variable.expression.inferredType
      }
    } // else the result is null
  }
// ... continues
```

**Figure 5:** Type inference for the Expressions DSL.

method, and the parenthesis for method call without arguments are omitted. In Java that would correspond to the expression `inferredType(e.getLeft())`.

Since in this DSL types are not explicit and new types cannot be defined, the types are modeled by singleton instances of simple classes (not shown here). The classes representing types all extend the base class `ExpressionsType`.

For most cases type computation is trivial, since we do not need to inspect subexpressions. For computing the type of a variable reference, we use the utility method `ExpressionsModelUtil.isVariableDefinedBefore` (used here as an extension method) that detects possible forward references. Since this is not related to typing concepts we are not showing the code here. We need to check that a variable reference does not refer to a forward declaration because this could trigger an infinite recursion. The other interesting case is the one of type inference of an additive expression: the inferred type depends on the fact that one of the two subexpression has type string. This case actually requires inspecting subexpressions.

```
// ... continues from the previous listing

/**
 * Is type1 conformant to type2: is type1 subtype of type2?
 */
def boolean isConformantTo(ExpressionsType type1,
    ExpressionsType type2) {
  return type2.isStringType || type1 === type2
}

// ... continues
```

**Figure 6:** Type conformance for the Expressions DSL.

We would like to stress that the type inference is simple and compact also because it does not perform any form of type checking. It is also straightforward to verify that the code in Figure 5 implements the type computation part of the type system informally described in Section 4.1. In particular, the only cases when a type cannot be inferred (and it is null) are forward references and references to undefined variables.

## 4.3 Type conformance

Even if the Expressions DSL is a simple language, it has the notion of type conformance: since the DSL provides implicit string conversion (with the string concatenation operation), then any expression can be used in a context where a string is expected. The implementation of type conformance is shown in Figure 6. Since we implemented types as singleton instances, we compare them directly with `===` (see Section 3.5).

## 4.4 Type expectations

The last mechanism we need in our type system is the one defining the expected type of an expression in a given context. In this DSL only non-atomic expressions have expectations on the subexpressions.

The computation of expected types is shown in Figure 7, where the `eContainer` method is part of the standard EMF API [9]. Note that this implementation only lists the cases when there are expectations, and in all other cases the returned expectation is null. Again, the case for the sum is the interesting one, since if none of the subexpressions are strings, then both subexpressions are expected to have integer type.

The implementation of Figure 7 covers all the cases of the type system informally described in Section 4.1.

```
// ... continues from the previous listing

/**
 * The expected type or null if there's no expectation
 */
def ExpressionsType expectedType(Expression exp) {
  val container = exp.eContainer
  switch (container) {
    MulOrDiv | Minus: INT_TYPE
    Not | And | Or: BOOL_TYPE
    Plus: {
      val leftType = container.left.inferredType;
      val rightType = container.right.inferredType;
      if (!leftType.isStringType && !rightType.isStringType) {
        INT_TYPE
      } else {
        STRING_TYPE
      }
    }
  }
}

// ... continues
```

**Figure 7:** Type expectations for the Expressions DSL.

## 4.5 Type checking

Now that we implemented all the mechanisms for type inference, conformance and expectations, we are ready to implement type checking in the validator of the Expressions DSL, `ExpressionsValidator`.

Since we separated type inference and type expectations, it is straightforward to verify type conformance with a single `@Check` method, as shown in Figure 8 (see Section 3.3 for the concept of `@Check` method).

First of all, we compute the inferred type and the expected type of the expression (recall from Section 4.3, Figure 6, that the expected type is computed based on the containing expression). Then,

- if both types are not null, we verify that the actual type is conformant with the expected type; if not, we issue an error (more details on this will be provided later);
- if the expected type is null, then there is nothing to check, since there are no expectation;
- if the type cannot be inferred, then the expression is a reference to a non declared variable or a forward reference. Also in this case, there is no need to issue further errors: an informative error, unrelated to types, is already issued by the other `@Check` methods of our validator (not shown here, since it is not related to the type system).

It is crucial to issue an informative error and to place it in the relevant part of the program. In Xtext, as sketched in Section 3.3, it is enough to call the method `error` with the correct information and the error markers will be automatically generated. In Figure 8, we create a string message of the shape "expected <type>, but was <actual type>", which should be useful enough; then, we specify the AST node containing the error (in this case, the containing expression of `exp`, that is, `exp.eContainer`), and the part of the AST node where the error marker should be placed, expressed as the EMF feature (in this case, it is the feature of the container where the current expression is stored, that is, `exp.eContainingFeature`[4]). The last argument, the issue code, represented by a constant string, can be used in other parts of Xtext, for example, for providing quickfixes, which are out of the scope of the paper.

```
class ExpressionsValidator
    extends AbstractExpressionsValidator {

  protected static val ISSUE_CODE_PREFIX =
    "org.example.expressions."
  public static val TYPE_MISMATCH =
    ISSUE_CODE_PREFIX + "TypeMismatch"

  @Inject extension ExpressionsTypeSystem

  @Check
  def void checkForwardReference(VariableRef varRef) {
    // not shown here
  }

  @Check
  def void checkTypeConformance(Expression exp) {
    val actualType = exp.inferredType
    val expectedType = exp.expectedType

    if (expectedType !== null &&
        actualType !== null &&
        !actualType.isConformantTo(expectedType)) {
      error("expected " +
          expectedType + ", but was " +
          actualType,
          exp.eContainer,
          exp.eContainingFeature,
          TYPE_MISMATCH)
    }
  }
// ... continues
```

**Figure 8:** Checking type conformance.

The result of this `@Check` method implementation can be seen in Figure 9, where the editor contains some invalid expressions. Let us consider `eval (3<1) && "a"`; when the `@Check` method is invoked with argument `(3<1)` the actual type and the expected type (from the containing expression `And`) are both boolean. When the `@Check` method is invoked with argument `"a"` the actual type string is not

4 Just like `eContainer`, `eContainingFeature` is part of the standard EMF API, which all the Java code generated by EMF inherits.

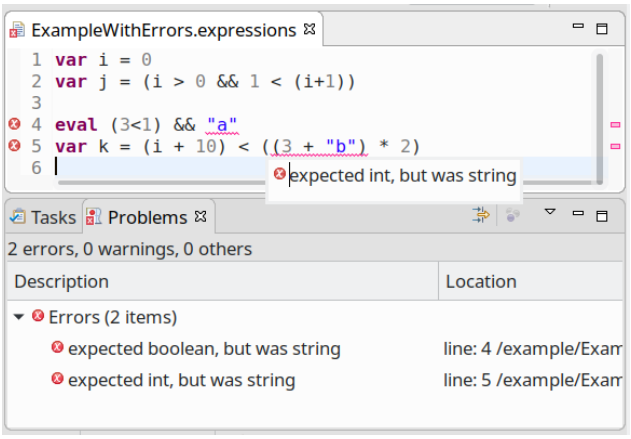


**Figure 9:** The errors about type conformance.

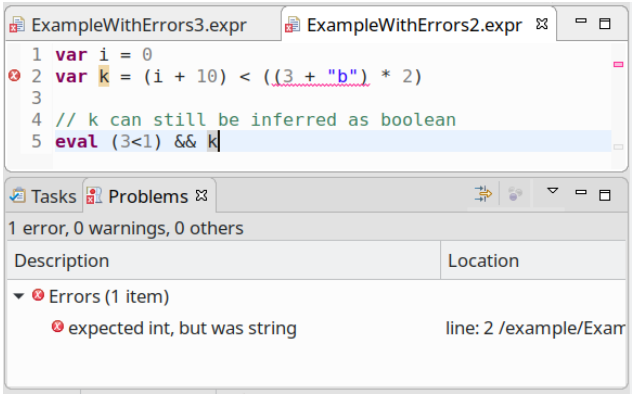


**Figure 10:** Type inference is not affected by other type errors.

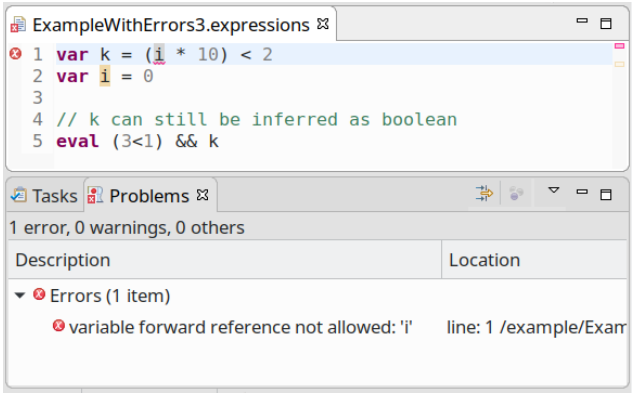


**Figure 11:** Type inference is not affected by forward reference errors.

conformant with the expected type boolean. The error is issued on the containing node `And` in the part that corresponds to the `right` feature containing `"a"` (refer to the `And` rule in Figure 1). Note that once we provide the correct arguments to the `error` method, all the markers are generated automatically by Xtext in the "Problems" view, in the editor rulers and in hovering pop-ups.

We believe that this strategy effectively produces informative errors in the relevant part of the program. The user of the DSL should have no problem in understanding what is going wrong in the program. Moreover, since we kept type inference separate from type checking, a type error in a part of the program does not prevent our type system from typing the other parts. For example, as shown in Figure 10, although the initialization expression of the variable `k` contains a type conformance error, our type system can still infer the type boolean for `k` so that other expressions using `k` are not marked with errors. Of course, the whole program is not valid, but generating cascading errors would only add "noise", making it harder to fix the errors.

Note also that type inference is not affected by other errors like forward references. Also the validator shown in Figure 8 does not perform conformance checking when the type of an expression cannot be computed, like in the case of a forward reference. That problem is issued by the other `@Check` method (not shown in the paper). Reporting another error would not add useful information. In Figure 11 an example of such a situation is shown, where only the useful error is reported. It is worth mentioning that also the scope provider implementation of the Expressions DSL allows Xtext to resolve a forward reference: the figure shows that the forward reference `i` in the variable declaration `k` is still correctly bound to the declaration of `i`.

## 4.6 Using types in the IDE

Xtext allows the DSL developer to customize many Eclipse parts concerning the DSL, for example, the "Outline" (which is automatically synchronized with the editor) and the "Hovering" for single parts of the current editor textual elements. It is quite standard for statically typed languages to use types in such IDE parts. For example, Eclipse JDT (Java Development Tools) provides nice representations of the current Java program in several views of Eclipse.

In the Expressions DSL we can use types in the "Outline" view: we can show all the variable declarations with their inferred types. That is another reason why it is crucial to be able to compute as many types as possible, disregard-

ing of type errors. We can also show type information when the user hovers on an expression. The results can be seen in Figure 12, where types are correctly shown even in the presence of errors in the program (the popup appears since we hover on `&&` in the editor). The implementation of such Eclipse mechanisms is straightforward in Xtext and in this example it is just a matter of using the `ExpressionsTypeSystem`, thus we do not show it here.

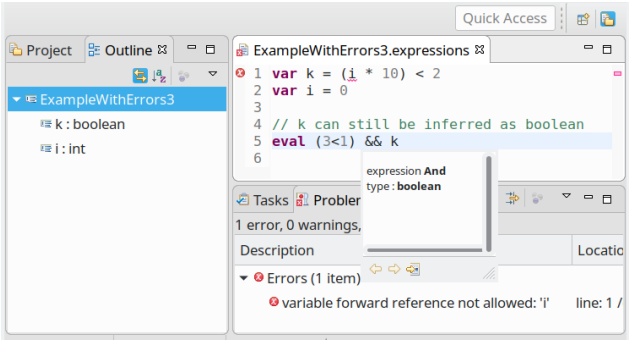


**Figure 12:** Types used in the Outline and for Hovering.

As mentioned in Section 2, the content assist should avoid suggesting proposals that would make the program invalid. After all, one of the advantages of statically typed languages is that the IDE can propose sensible completions by using the types. For the Expressions DSL, for example, the content assist should not propose completions for variable references that refer to a forward reference. Moreover, it should not propose variables whose type is not conformant to the type expected by the current expression. The desired result is shown in Figure 13, where only the variables of type integer (and that are not forward references) are proposed in the context of the multiplication expression. The interesting parts of the implementation of these filtered proposals in the content assist are shown in Figure 14 (Xtext will reflectively invoke methods in this class based on the name of the method and on the specific program context where content assist is requested).

Once again, having a type system that is able to compute types in spite of errors in other parts of the program is crucial for implementing sensible proposals in the content assist: most of the time, when the user asks for proposals, the program is not complete and probably not valid.

Finally, we can use the inferred type for implementing *code mining* for the Expressions DSL editor. A code mining[5] represents content (for example, labels and icons) that are shown within the text editor along with the source text, but are not part of the program itself (thus, that additional text is ignored by the compiler). Some examples in Java are the number of references to a given declaration or the name of the parameters in a method invocation.

Xtext supports code mining and we can use this mechanism to show the inferred types of variables also in the editor itself, while the user is writing the program. The additional text of the code mining is not part of the program, it does not have to be part of the DSL syntax and it is not even modifiable by the programmer.

An example of the code mining in action is shown in the screenshots of Figure 15, where we see how the code mining changes while the user is editing the program. Recall that the "`: <type>`" is not part of the program and it is not even part of the syntax of the DSL. Note that the type is shown even in the presence of type errors, thanks to the

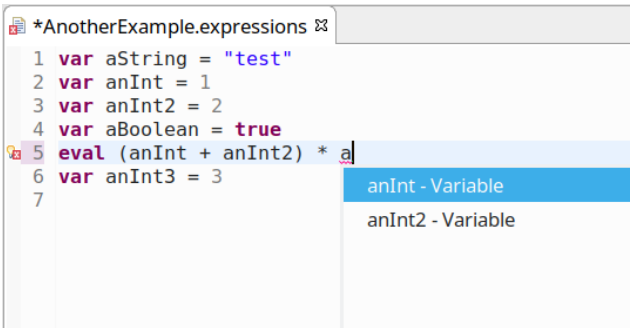


**Figure 13:** Content Assist suggests only valid proposals.

```
class ExpressionsProposalProvider
        extends AbstractExpressionsProposalProvider {
  @Inject extension ExpressionsModelUtil
  @Inject extension ExpressionsTypeSystem

  override completeAtomic_Variable(EObject elem, ...) {
    if (elem instanceof Expression) {
      val elemType = elem.inferredType
      elem.variablesDefinedBefore
        . filter [
          variable |
          variable.expression
            .inferredType.isConformantTo(elemType)
        ]
        .forEach[
          variable |
          // ... create proposal
        ]
    }
  }
}
```

**Figure 14:** The Content Assist for variable reference completion, relying on types.

5 A mechanism that has been recently added to Eclipse, in the release named "Photon".

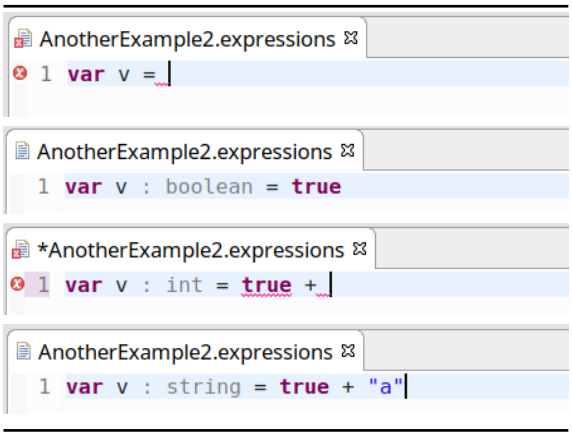


**Figure 15:** Code Mining in action for inferred types of expressions.

```
class ExpressionsCodeMiningProvider
    extends AbstractXtextCodeMiningProvider {

  @Inject extension ExpressionsTypeSystem

  override void createCodeMinings(XtextResource resource, ...) {
    // get all variable declarations
    val variables =
      EcoreUtil2.eAllOfType(resource.contents.get(0),
        Variable)

    for (variable : variables) {
      val type = variable.expression.inferredType
      if (type !== null) {
        // ... create the code mining
      }
    }
  }
}
```

**Figure 16:** The code mining for variable declarations, relying on types.

error recovery capabilities of our type system. The interesting parts of the implementation of code mining, using the type system, are shown in Figure 16.

# 5 Typing an OO language

In this section, following the patterns described in Section 2, we implement the type system of Featherweight Java (FJ) [3, 15], a lightweight functional version of Java, which focuses on the main basic features of an Object-Oriented language. FJ has been proposed to formally study the properties of the Java type system and to design Java extensions. It is not meant to be used to implement fully-featured Java programs. However, it will allow us to concentrate on typing issues, focusing on the main mechanisms of OOP, namely, inheritance and method invocation.

```
class Object { }
class A extends Object {}
class B extends Object {}

class Pair extends Object {
  private Object fst;
  private Object snd;

  public Object getFst() {
    return this.fst;
  }
  public Object getSnd() {
    return this.snd;
  }
  public Pair setFst(Object newFst) {
    return new Pair(newFst, this.snd);
  }
  public Pair setSnd(Object newSnd) {
    return new Pair(this.fst, newSnd);
  }
}

new Pair(new A(), new B()).setFst(new B())
```

**Figure 17:** An example in FJ.

We implement the type system of FJ using Xsemantics. Since FJ provides the formalization of the type system, using Xsemantics has the benefit that the implementation, that is, the specification in Xsemantics, is quite similar to the formal type system of FJ. Note that Xsemantics could be used also for implementing the type system of the Expressions DSL (Section 4). However, for the first case study we preferred to provide a hand written implementation of the type system to focus on the design choices and implementation patterns, without introducing another framework. In general, the use Xsemantics for implementing the type system could be preferred to a hand written implementation for languages where the type system has been already formalized. However, the mechanisms provided by Xsemantics, like caching and tracing (described in Section 6), could be helpful also when the formalization of the type system is not already available.

## 5.1 FJ in Xtext

An example of an FJ program, borrowed from [3] is shown in Figure 17. Note that in FJ a method's body consists of a single return statement (since FJ is a functional version of Java where block of statements are not supported). The receiver in a member selection expression must be always specified; `super` is not considered in FJ. In FJ the class constructor has a fixed shape, thus, for simplicity, we consider

```
FJProgram: (classes += FJClass)* (main = FJExpression)? ;

FJClass:
  'class' name=ID ('extends' superclass=[FJClass])? '{'
    (members += FJMember)*
  '}' ;

FJMember: FJField | FJMethod;

enum FJAccessLevel:
  PRIVATE='private' | PROTECTED='protected' |
  PUBLIC='public';

FJField : access=FJAccessLevel? type=[FJClass] name=ID ';' ;

FJMethod:
  access=FJAccessLevel?
  type=[FJClass] name=ID
  '(' (params+=FJParameter
    (',' params+=FJParameter)*)? ')' '{'
    'return' expression=FJExpression ';'
  '}' ;

FJParameter: type=[FJClass] name=ID ;

FJTypedElement: FJMember | FJParameter ;

FJExpression:
  FJTerminalExpression =>(
    {FJMemberSelection.receiver=current} '.'
    member=[FJMember]
    (methodinvocation?='(' (args+=FJExpression
      (',' args+=FJExpression)*)? ')')?
  )* ;

FJTerminalExpression returns FJExpression:
  {FJThis} 'this' |
  {FJParamRef} parameter=[FJParameter] |
  {FJNew} 'new' type=[FJClass]
    '(' (args+=FJExpression (',' args+=FJExpression)*)? ')' |
  {FJCast} '(' type=[FJClass] ')' expression=FJExpression |
  '(' FJExpression ')' ;
```

**Figure 18:** The Xtext grammar of FJ. The preamble of the grammar is not shown.

the constructors as implicit. For this reason, a `new` expression must pass an argument for each field in the class, including inherited fields (arguments for inherited fields must come first). The class `Object` is not implicitly defined in this version of FJ. A type in FJ is a reference to an FJ class definition, thus there are no basic types. After class definitions, an FJ program can contain an expression that represents the *main* expression.

The Xtext grammar for FJ is shown in Figure 18. Note that the grammar rule for member selection handles both field selection and method invocation. A method invocation differs from a field selection for the opening parenthesis. The presence of parentheses is stored in the boolean feature `methodinvocation`.

```
judgments {
  inferType |- FJExpression expression : output FJClass
    error "cannot type " + expression
    source expression
  subtype |- FJClass left <: FJClass right
    error left + " is not a subtype of " + right
  subtypesequence |-
    List<FJExpression> expressions
      << List<? extends FJTypedElement> elements
}
```

**Figure 19:** The judgments for the Xsemantics type system of FJ.

In order to make the example more interesting for the goals of the paper, we added to FJ the standard access-level modifiers for fields and methods. Since there is no "package" concept in FJ the default level is always `private`.

## 5.2 Typing FJ

The implementation of the type system of FJ in Xsemantics shown here is tailored to the goals of the paper itself, thus, it is quite different from the one presented in [4].

We will provide a brief description of the syntax of Xsemantics in order to make the examples shown in the paper understandable. We refer the interested reader to [4] for more details on Xsemantics.

Xsemantics uses a syntax that resembles rules in a formal setting [15–17]. A *system definition* in Xsemantics is a set of *judgments*, i.e., assertions about some properties of programs, and a set of *rules*, which assert the validity of certain judgments, possibly on the basis of other judgments. Rules have a conclusion and a set of premises. Rules act on the EMF objects representing the AST of the program. The Xsemantics compiler will then generate Java code that can be used in the DSL implemented in Xtext for scoping, validation and other mechanisms.

### 5.2.1 Judgments

The judgments in Xsemantics for FJ are shown in Figure 19. In general, the developers can choose the symbols that they see fit. We chose symbols such as $|-$, $:$ and $<:$ because these are the same symbols that are used in FJ formalization [3] and are typical of formal systems [15–17]. In fact, in a typical formal type system, you find judgments of the shape $\Gamma \vdash e : T$, to be read as "in the environment $\Gamma$, $e$ has type $T$" and $\Gamma \vdash T_1 <: T_2$, to be read as "in the environment $\Gamma$, $T_1$ is a subtype of $T_2$". We will get back to the concept of "environment" later in this section.

The judgment `inferType` takes an `FJExpression` as input parameter and provides an `FJClass` as output parameter. The judgment `subtype` does not have output parameters (thus its output result is implicitly boolean, stating whether the judgment succeeded).

Judgment definitions can include `error` specifications that are useful for generating custom error information. An error specification, besides the error message, specifies the "source" of the error, i.e., the AST node, and possibly the "feature" that contains the error. This information is used to generate errors using the standard Xtext validator mechanisms (Section 3.3).

The judgment `subtypesequence` basically checks subtyping of sequences of types (it is useful in the rules as we will see in the following).

Note that for FJ we did not implement the concept of "expected type" as we did in Section 4.4. Indeed the formalization of FJ [3] does not have such a concept and the conformance is checked explicitly in the type rules when needed. We followed the same approach in this implementation. Moreover, due to the reduced set of features of FJ, type conformance is checked only when passing arguments either to a constructor or to a method, and when checking that the returned expression in a method is conformant to the declared return type. Thus, having the separate concept of expected type would not improve our implementation. However, we factored out the common behavior of checking arguments against parameters in the judgment `subtypesequence`.

### 5.2.2 Rules

Rules are meant to implement declared judgments. A rule has a name, a *rule conclusion* and the *premises*. The conclusion consists of the name of the *environment* of the rule, a *judgment symbol* and the *parameters* of the rules. Parameters are separated by *relation symbols* that must respect the ones of the implemented judgment. The rule *environment* comes from formal systems, where it is usually denoted by $\Gamma$. The environment can be used to pass additional arguments to rules, e.g., contextual information and bindings for specific keywords, like `this` in FJ. The environment can be accessed with the predefined function `env`. When invoking a rule, one can specify additional *environment mappings*, using the syntax `key <- value` (e.g., as we will see later, `'this' <- C`). The empty environment can be specified using the keyword `empty`.

Xsemantics rules have a "programming"-like shape: differently from standard deduction rules, the conclusion comes before the premises (similar to other frameworks like, e.g., [18, 19]). When a rule has no premise, the special form `axiom` can be used, which defines only the conclusion.

```
rule ClassSubtyping
  G |- FJClass left <: FJClass right
from {
  left == right
  or
  superclasses(left).contains(right)
}

rule SubtypeSequence
  G |- List<FJExpression> expressions
       << List<FJTypedElement> typedElements
from {
  expressions.size == typedElements.size
  or
  fail
    error "expected " + typedElements.size +
      " arguments, but got " + expressions.size

  val typedElementsIterator = typedElements.iterator
  for (exp : expressions) {
    G |- exp : var FJClass expType
    G |- expType <: typedElementsIterator.next.type
  }
}
```

**Figure 20:** The rules for subtyping.

The premises of a rule, specified in the `from` block, can be any Xbase expression. Xbase [20] is an embeddable Java-like language, which is part of Xtext. It is the same language used in Xtend methods (refer to Section 3.5). This also means that Xsemantics can access any Java type, just like Xtend. The premises of an Xsemantics rule are considered to be in *logical and* relation unless the explicit operator `or` is employed to separate blocks of premises.

In Figure 20 we show the rules for implementing subtyping (the rules belong to the judgment `subtype` and `subtypesequence`, respectively). The subtype is the one of Java: `left` is a subtype of `right` if either the two classes are the same or `left` is a (possibly indirect) subclass of `right`. The latter condition is checked by using an *auxiliary function* that computes all the superclasses of a given class. This is implemented in Xsemantics as well, but we will not show that here (see the source code of the examples). The other rule takes a list of expressions and a list of `FJTypedElement`s (recall from Figure 18 that an `FJTypedElement` can be either an `FJMember` or an `FJParameter`) and checks that the types of the expressions are subtypes of the types of the given `FJTypedElement`s.

Premises in a rule can invoke other rules, possibly belonging to other judgments. Note that in the premises one can assign values to the output parameters and when other rules are invoked, upon return, the output argu-

```
axiom TThis
  G |- FJThis t : env(G, 'this', FJClass)
  error "'this' cannot be used in the current context"
  source t

rule TCast
  G |- FJCast cast : cast.type
from {
  G |- cast.expression : var FJClass expType

  G |- cast.type <: expType
  or
  G |- expType <: cast.type
}

rule TNew
  G |- FJNew newExp : newExp.type
from {
  var fields = fields(newExp.type)
  G |- newExp.args << fields
}
```

**Figure 21:** Some typing rules of the judgment `inferType`.

```
checkrule CheckMethodBody for
  FJMethod method
from {
  // pass an environment for "this"
  'this' <- method.eContainer |-
    method.expression : var FJClass bodyType

  empty |- bodyType <: method.type
  or
  fail
    error "Type mismatch: cannot convert from " +
      bodyType.name + " to " + method.type.name
    source method.expression
}

checkrule CheckMain for
  FJProgram program
from {
  program.main === null // nothing to check
  or
  empty |- program.main : var FJClass mainType
}
```

**Figure 22:** Two `checkrule`s for FJ.

ments will have the values assigned in the invoked rule. This is what happens in the loop of the second rule: the type of each expression is computed using the judgment `inferType` and then the subtyping is checked using the judgment `subtype`.

If one of the premises fails, then the whole rule will fail, and the whole stack of rule invocations will fail as well. In particular, if the premise is a boolean expression, it will fail if it evaluates to `false`. If the premise is a rule invocation, it will fail if the invoked rule fails.

As shown in the rule `SubtypeSequence`, Xsemantics allows the developer to use an explicit failure statement in an `or` branch in order to provide errors that could be more useful than the default ones. In the above rule, without the explicit `fail` with a detailed error, in case of lists of different size, the generic failure message "`failed: expressions.size == typedElements.size`" would be reported, which contains too many internal details to be effectively useful.

In Figure 21 we show other typing rules, belonging to the judgment `inferType`. The implementation of these rules is based on the rules formalized in [3]. The first one is the axiom for typing `this`. For typing `this` 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 rule will fail. This axiom assumes that the passed environment contains a mapping for `this`. We will see later when such a mapping is passed. Thus, if `this` is used outside a method's body, its type will not be computed and a corresponding error will be issued. Note that Xsemantics allows the developer to specify an error also at the single rule/axiom level, as in the axiom `TThis`. The rule for typing cast expressions can be read as: "the type of a cast expression is the type we cast to, provided that, 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". This corresponds to the fact that in Java the two types involved in a cast must be related in the class hierarchy. The rule for `FJNew` uses another auxiliary function (not shown here) that computes the list of all the fields in the class hierarchy. We also use the `subtypesequence` judgment for checking that the arguments of the `FJNew` expression are conformant to the types of the fields. The other typing rules follow the same pattern, especially the one for method invocation, which uses `subtypesequence` similarly to the rule for `FJNew`, and we omit them here.

### 5.2.3 Checkrules

Xsemantics provides some special rules, `checkrule`s, which do not belong to any judgment. A `checkrule` has a single parameter, which is the AST node to be checked by the validator, and the premises (but no rule environment). Xsemantics generates a Java validator with a `@Check` method (Section 3.3) for each `checkrule`.

In Figure 22 we show two `checkrule`s: the one for checking that the body of a method conforms to the declared return type and the one for checking that the main expression can be typed in an empty environment. In the

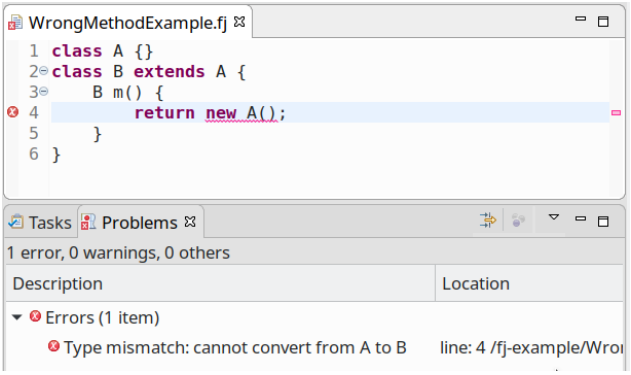


**Figure 23:** Useful error message placed on the correct part.

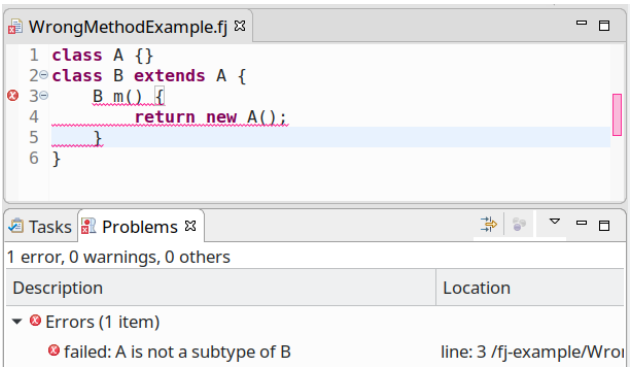


**Figure 24:** The error is not useful and it spans too much contents in the program.

first rule we explicitly pass a mapping for `this` in the environment for computing the type of the method body, so that occurrences of `this` can be successfully typed with the axiom shown in Figure 21. In particular, `this` is mapped to the container of the method, i.e., the containing `FJClass`. Then, we verify that such a type is a subtype of the declared return type. Note that we explicitly generate an error in case of failure with a useful description and with the `source` of the error, i.e., the node in the AST that will be marked with the error. The result in Eclipse is shown in Figure 23. Had we not made the error explicit, the default generated error would not be useful enough and it would be placed on the whole method declaration, as shown in Figure 24. Other checkrules, e.g., checking the class hierarchy is not cyclic, that method overrides is correct, etc., are not shown in the paper.

## 5.3 Scoping for member selection

Implementing cross reference resolution in an OO language is strictly related to typing when it concerns a member selection expression, since we first have to compute the type of the receiver expression and then inspect the hierarchy of that type. Note that the receiver expression might be in turn another member selection expression. Thus, in the implementation of the scope provider for FJ we use the Java code generated by Xsemantics from the type system specification we sketched in the previous section. Note that Xsemantics generates a Java method for each Xsemantics judgment; such methods accept also an explicit environment (further details about Xsemantics code generation can be found in [4]).

```
class FJScopeProvider extends AbstractFJScopeProvider {

  @Inject FJTypeSystem typeSystem

  override getScope(EObject context, EReference reference) {
    switch (context) {
      FJMemberSelection: {
        val receiverType =
          typeSystem.inferType
            (environmentForThis(context), context.receiver)
        if (receiverType !== null)
          Scopes.scopeFor(
            typeSystem.fields(receiverType) +
            typeSystem.methods(receiverType)
          )
        else
          IScope.NULLSCOPE
      }
      default: super.getScope(context, reference)
    }
  }

  def private environmentForThis(EObject context) {
    val env = new RuleEnvironment
    val containingClass =
      getContainerOfType(context, FJClass)
    if (containingClass !== null)
      env.add("this", containingClass)
    return env
  }
}
```

**Figure 25:** Scoping implementation for FJ member selection.

The implementation of the scope provider is shown in Figure 25. Xtext calls the method `getScope` passing the context expression and the reference that must be resolved. In our case, we are only interested in the case when the context is an `FJMemberSelection` and the reference to be resolved is a reference to an `FJMember`. We have to compute the type of the receiver expression, providing an environment with a mapping for `this`; this is constructed by retrieving the container of type `FJClass` (using the standard EMF utility API, `getContainerOfType`). Remember that in case `this` was used in the "main" expression of an FJ expression, such a container would be null and occurrences of `this` could not be typed. If the receiver expres-

```
override getScope(EObject context, EReference reference) {
  // ... as before
  if (receiverType !== null) {
    if (context.methodinvocation)
      Scopes.scopeFor(
        typeSystem.methods(receiverType)
      )
    else
      Scopes.scopeFor(
        typeSystem.fields(receiverType)
      )
  }
  // ... as before
}
```

**Figure 26:** Alternative scoping implementation for FJ member selection, which is too strict.

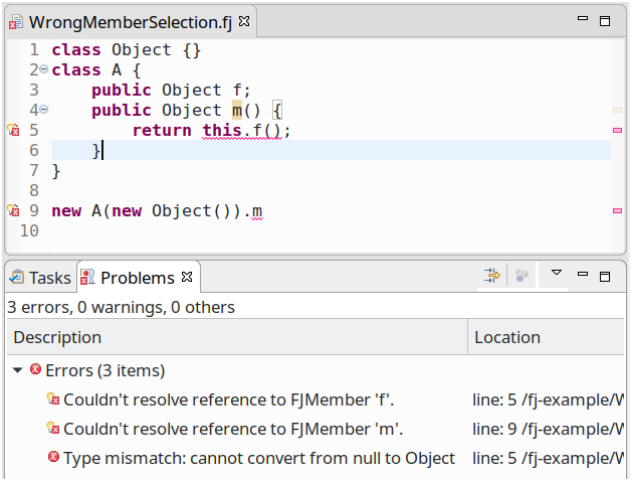


**Figure 27:** Too generic errors about member selection and cascading error on the method body due to unresolved member.

sion can be given a type, the scope will consist of all the fields and methods, including the inherited ones, of such a type (computed through the auxiliary functions `fields` and `methods`, respectively).

Further details about the implementation of this scope provider are described in the next section, where we address crucial issues concerning useful error reporting.

## 5.4 Scoping vs. validity

In the scope provider implementation of Figure 25 we never check whether the current member selection must refer to a field or to a method. As we said in Section 5.1, an `FJMemberSelection` deals with both field selection and method invocation, using the boolean feature `methodinvocation` to distinguish them.

It might be tempting to filter the returned scope according to the kind of member selection, so that in a method invocation expression fields cannot be referred at all and the other way round for field selection. For example, an alternative implementation is shown in Figure 26, where the returned scope takes into consideration whether the member selection must refer to a field or to a method.

This implementation is certainly sound, but it has a few drawbacks. First of all, if the user selects a field and uses it as a method (in a method invocation expression) the resulting error will be a generic "Couldn't resolve reference to…" (similarly for the case when a method is used as a field). Instead, if a member is used in a member selection expression in the wrong way, but it is still a visible member of the receiver's class, we should generate a more useful error message. Moreover, since the member cannot be resolved, other parts of the type system will fail as well, losing the error recovery property. An example is shown in Figure 27, where, besides the generic "unresolved reference" errors, another error is issued on the body of the method: since the field `f` is not resolved, the body of the method cannot be given a type and the check for conformance w.r.t. the method return type fails as well.

As described in Section 2, we should not mix visibility and validity. We then should be less strict on the validity of the scope, as we did in the original scope provider implementation of Figure 25. This way, even if a member is used in the wrong way, the type system can still type the rest of the program. We then implement a separate check rule, where we check whether the (already resolved) member is valid in the member selection expression. This allows us to make the error clear: the member can be referred, since it is visible in the current context, but it is being used in the wrong way. The check rule is shown in Figure 28. Note that Xsemantics error specification also allows the developer to specify the part of the node to mark with the error (mimicking the arguments of the standard Xtext validator `error` method that we used in Figure 8). The `feature` is specified using the constants in the AST classes, automatically generated by Xtext starting from the grammar definition.

The result can be seen in Figure 29. Note how the errors are much clearer than the ones in Figure 27, not to mention that cascading errors are also avoided (indeed, the members are correctly resolved and linked—see the occurrences of `m` correctly marked in the editor).

Mixing visibility and validity in an OO language typically leads to mixing visibility and *accessibility*. An example of an accessibility error is a subclass trying to access a private field in a superclass. The private field is "visible" in that context, but it is not "accessible". In fact, `private`, `protected`, etc., are called *access level modifiers*

```
checkrule CheckMemberSelection for
  FJMemberSelection sel
from {
  val member = sel.member

  if (member instanceof FJField && sel.methodinvocation) {
    fail
      error "Method invocation on a field"
      source sel
      feature FJ_MEMBER_SELECTION__MEMBER
  } else if ... // similar for the other way round
}
```

**Figure 28:** Checking member selection.

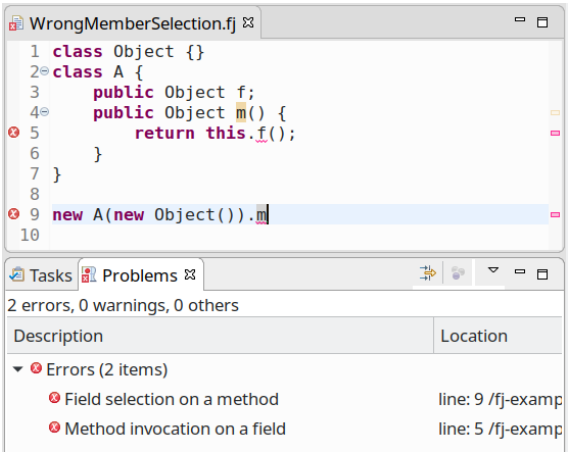


**Figure 29:** Errors concerning wrong member selections.

in Java [21], not visibility modifiers. The Java compiler issues an informative error of the shape "the field has private access" (accessibility error), instead of a generic "the symbol cannot be found" (visibility error). The Eclipse JDT editor follows the same strategy. Moreover, the Java editor still allows the user to navigate to the definition of a member even if it is not accessible in that part of the program.

For the above reason, we did not implement any filtering based on the access level in our scope provider implementation (Figure 25) and we have a specific `checkrule` implementing this check, shown in Figure 30. The check rule relies on the auxiliary function `isAccessible`. The reason why we have a dedicated auxiliary function will be clear in the following.

Keeping visibility and validity separate allows us to provide useful errors, without preventing cross reference resolution, as shown in Figure 31.

As done in the previous DSL, we customize the content assist in order to filter out completions that would make the program invalid. For FJ, the proposals for a member selection expression must not include members that are

```
auxiliary isAccessible(FJMember member, EObject context) {
  val receiverClass = getContainerOfType(context, FJClass)
  val memberClass = getContainerOfType(member, FJClass)

  return member.access == FJAccessLevel.PUBLIC ||
    receiverClass === memberClass ||
    {
      empty |- receiverClass <: memberClass
      member.access != FJAccessLevel.PRIVATE
    }
}

checkrule CheckAccessibility for
  FJMemberSelection sel
from {
  val member = sel.member

  isAccessible(member, sel.receiver)
  or fail
    error "The " + member.access + " member "
      + member.name + " is not accessible here"
    source sel
    feature FJ_MEMBER_SELECTION__MEMBER
}
```

**Figure 30:** Checking member accessibility.

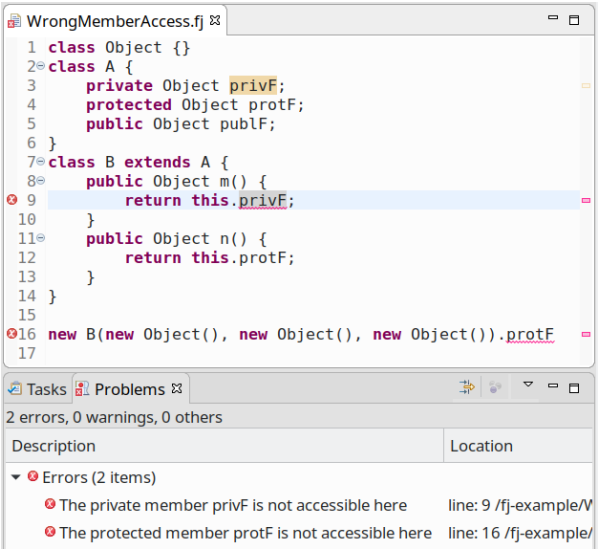


**Figure 31:** Errors due to invalid member access.

not accessible in that program context. We do that by using the auxiliary function `isAccessible` defined in the FJ type system (Figure 30). The customization of the content assist is shown in Figure 32, in a simplified form, not showing internal details of Xtext.

The customized content assist in action is shown in Figure 33: the private members of the superclass are not shown as completions since in that method they are not accessible, while public and protected members are proposed.

```
class FJProposalProvider
      extends AbstractFJProposalProvider {

  @Inject FJTypeSystem typeSystem

  override completeFJExpression_Member(EObject model, ...) {
    lookupCrossReference(...) [
      exp | typeSystem.isAccessible(exp as FJMember, model)
    ]
  }
}
```

**Figure 32:** The content assist for FJ filters members that would not be accessible in the current context.

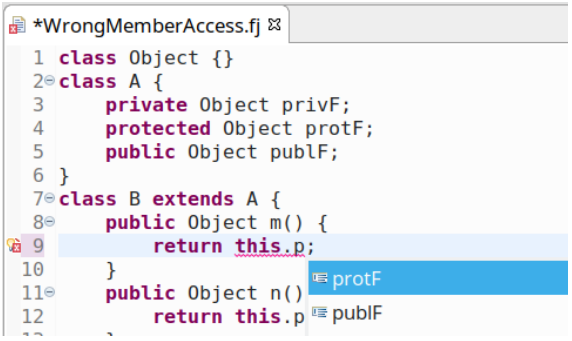


**Figure 33:** Filtered proposals.

It is worth mentioning that the default implementation of the Xtext proposal provider simply uses the scope provider. Thus, if we implemented filtering at the scope provider level we would not need to customize the proposal provider at all. On the contrary, since we do not filter the elements in our scope provider implementation we must customize the content assist by filtering out invalid proposals. This is the price to pay if we want to achieve a better DSL user experience, as we saw in this section.

## 5.5 Type computation vs. type checking

As anticipated in Section 2, formal type systems typically deal with type computation and type checking at the same time. Since the typing rules for FJ that we showed in Section 5.2 aimed at mimicking the formal type system presented in [3], type computation and type checking are mixed together in our implementation. For example, considering the typing rule `TNew` in Figure 21, the type system will not compute the type of a `FJNew` expression if the arguments passed to the constructor are not valid. Member references on the `FJNew` expression will then be unresolved. An example is shown in Figure 34. In the figure, the errors are not helpful, especially the ones related to the unresolved members. Moreover, since the type system does not recover from errors, possible actual errors in the subsequent expressions are not detected at all. For example, the problem with the invocation of the method `m` is that we do not pass any argument, while one is expected. This error is not detected, since `m` cannot be resolved at all.

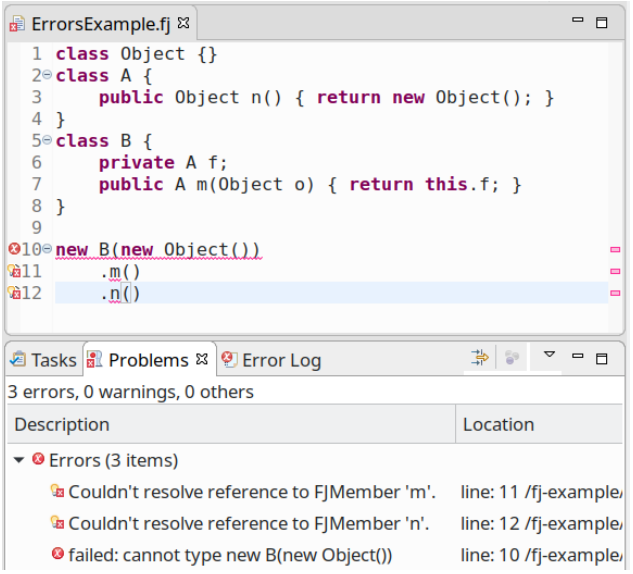


**Figure 34:** Too many cascading errors.

```
axiom TNew
  G |- FJNew newExp : newExp.type

axiom TCast
  G |- FJCast cast : cast.type

checkrule CheckNew for FJNew newExp from {
  // ... type checking moved here
}
checkrule CheckCast for FJCast cast from {
  // ... type checking moved here
}
```

**Figure 35:** Improved type system implementation separating type computation from type checking.

In this subsection, we improve the implementation of the type system by separating type computation from type checking. We turn type computation rules into axioms, since the type can be computed without checking subexpressions. Then, for each FJ expression we write an additional `checkrule`: the premises of the rules of the original implementation are moved into the new checkrules. Some modifications are sketched in Figure 35.

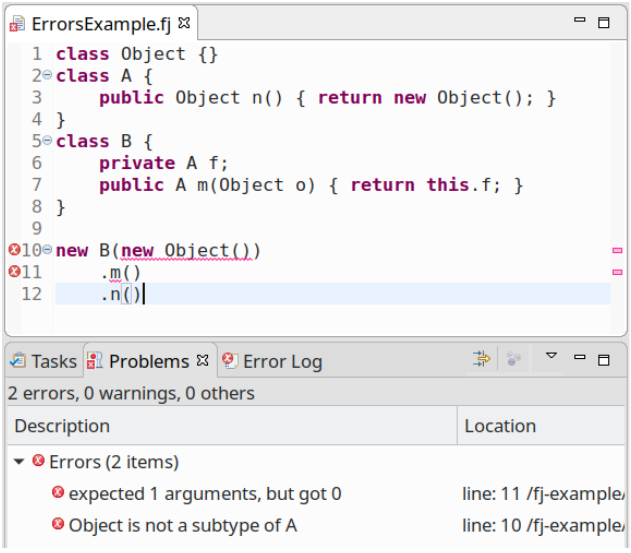


**Figure 36:** Only useful errors are reported.

With this modified type system, only the useful errors are reported, and the scope provider can resolve member references even in the presence of errors, as shown in Figure 36. Indeed, `m` and `n` can be resolved in spite of the errors in the receiver expressions. Moreover, the error concerning invoking `m` without arguments is now detected.

# 6 Other improvements

As we said in Section 2, caching type computations can increase the performance of the type system implementation. The type system is used by many Eclipse parts, for continuously checking the program while the user is editing it, and for populating several views. Caching would then keep the IDE responsive. In particular, typing rules might be used many times for computing the type of a given expression. Some subtyping checks and some auxiliary functions are also used more than once from within the type system implementation. In the FJ example the subtyping relation between the same two classes can be checked by many checkrules during the validation.

In order to avoid the typical problems when dealing with caching, that is, keeping track of changes that should invalidate the cached values, Xtext provides some caching mechanisms that automatically discard the cached values as soon as the contents of the program changes.

The sources of the "Expressions DSL" make use of such caching mechanisms and a few benchmarks about the impact of caching are available in the example's unit tests. Further benchmarks on the importance of caching have already been shown in [4, 22].

An Xsemantics specification can enable caching at several levels (e.g., judgments, auxiliary functions, etc.) so that Xsemantics generates Java code that automatically uses the Xtext caching. The implementation of FJ (see the source code of the examples) uses the caching mechanisms provided by Xsemantics.

Note that it is up to the developer to enable caching only on specific judgements. Indeed, caching should be enabled with care, otherwise it could decrease the performance. In fact, the caching is based on the Java hashing features, thus it makes sense only when used with actual object instances, not with references. Indeed, in the AST of a program there might be many different references to the same object. In such a scenario references as cache keys will only lead to many cache misses.

Xsemantics automatically generates errors through the standard Xtext error mechanisms described in Section 3.3 by using the error information generated during the application of the rules and checkrules. By default, Xsemantics will use the inner most error information referring to a node in the AST. This is usually the interesting error. However, this behavior can be customized by the developer, for example, in order to show all the rules that failed. This can be seen in the "Lambda DSL" example that is part of Xsemantics distribution; in that example DSL we implement the type system with unification for a simple $\lambda$-calculus (also briefly described in [4]), and when a term cannot be typed it is useful to show the whole trace of applied rules that made the unification fail.

Thus, Xsemantics internally keeps track of the trace of the applied rules when invoking a judgment implemented in Xsemantics. This trace could be useful also for debugging purposes or for testing the type system itself (in [4] an example of use of the trace for proving formal properties is also shown). Most of all, as we saw above, it contains all the information that leads to a possible failure in the type system. Besides the customization of the strategy for generating errors, the trace of failures is also directly available in the Xsemantics specification itself, for creating useful error information in the rules and checkrules. The access is provided by the implicit variable `previousFailure`. This is automatically handled by Xsemantics at run-time: in case of a rule failure, this variable contains all the problems that took place when applying the rule.

An example of this can be seen in the checkrule for a method invocation expression, show in Figure 37, which is the one generating the error on the invocation of `m` in Figure 36. When checking whether the passed arguments respect the method parameters, in case of a failure the

```
checkrule CheckSelection
  for FJMemberSelection selection
from {
  // check message if it's a method call
  val member = selection.member
  if (member instanceof FJMethod) {
    'this' <- member.eContainer
        |- selection.args << member.params
    or fail
      error previousFailure.message
      source selection
      feature FJ_MEMBER_SELECTION__MEMBER
  }
}
```

**Figure 37:** Checking method invocation with involved error handling.

whole expression would be marked with error, that is, the receiver and the invoked method. This would be distracting for the user. For this reason, in the rule of Figure 37, we create an explicit error by specifying the part of the expression to be marked with error (with the `feature` specification) but we reuse the original error message, retrieved by means of `previousFailure`.

This mechanism is useful when implementing a complex type system, where errors have to be really informative to help the user fix type errors. An example is shown in [22], where error messages in the presence of Java-like generics are created by inspecting errors in the trace of applied rules by means of `previousFailure`.

# 7 Related work

In this section we discuss some related work, concerning type errors, language workbenches and type systems frameworks.

## 7.1 Type errors

Type inference in a language relieves programmers from the burden of explicitly specifying types of declared elements. However, when a type cannot be inferred, error messages might become obscure, making it hard to fix the problem in the program. This is especially true for functional languages with almost full type inference, like Haskell and OCaml (see, for example, the papers [23–28], just to mention a few).

In particular, in functional languages, it is common to implement embedded DSLs [29], that is, DSLs developed as a particular form of API in the host general purpose language. The crucial problem with such DSLs is that possible type errors usually leak details of the DSL implementation and are less informative for the user [30]. For this reason, several mechanisms have been proposed for functional languages to customize the type errors generated by the compiler [31–37], in order to make such errors expressed in terms of the domain.

For an insightful and more complete study on the current research status of support for debugging type errors we refer the interested reader to [38].

The context of this paper is slightly different: we focus on imperative external DSLs with a limited form of type inference. Thus, the above mentioned problems with type error reporting are less prominent. In fact, the domain is part of the DSL, and the type system can already provide type error messages that include domain specific information. However, in a General Purpose Language, improving type error messages, and allowing the developer to customize them for specific contexts, is still useful. For example, C++ and Java can sometime generate obscure error messages (see, e.g., [39, 40], respectively).

Thus, we plan to investigate if and how some of the approaches presented in the above cited papers could be merged into the patterns presented in this paper. In particular, we plan to study how the implementation patterns proposed in the paper can be applied to type inference systems based on unification like the Hindley-Milner type systems [41], where, as mentioned above, it is challenging to generate informative error messages. As stated in Section 6, we have an implementation of the type system with unification for a simple $\lambda$-calculus (also briefly described in [4]) and when a term cannot be typed we show the whole trace of applied rules that made the unification fail. Thus, the error information shown to the user try to cover several program positions that contribute to the type error (along the lines of other implementations [25, 42, 43]). However, this strategy is not optimal, since error messages are not filtered, and they could overwhelm the user with internal details of the unification algorithm.

Since in this paper we target the IDE, it also becomes interesting to investigate on possible mechanisms, especially applied to Xsemantics, to automatically generate suggestions to the user in order to fix a type error. With that respect, existing works would be used as inspiration, like, e.g., [26, 44–47]. In particular, such approaches use heuristics for detecting type errors and helping debugging type errors. Other approaches, like [48], also use machine learning to provide useful type error messages. We plan to investigate in that direction, even because machine learning is employed in other parts of IDE mechanisms, like the content assist [49] and the code formatter [50].

## 7.2 Language workbenches

Concerning the implementation framework, we chose Xtext since it is the de-facto standard framework for implementing DSLs in the Eclipse ecosystem, it is supported with a wide community and it also has many applications in the industry. As shown in Section 8, Xtext targets other development environment besides Eclipse.

There are many tools for implementing DSLs and IDE tooling and we refer to [51–53] for a wider comparison. Tools like IMP (The IDE Meta-Tooling Platform) [54] only deal with IDE features. TCS (Textual Concrete Syntax) [55] is similar to Xtext, but TCS assumes that a metamodel for the AST is already implemented and requires the developer to associate syntactical elements to metamodel elements. On the contrary, Xtext only requires the grammar specification and derives automatically the metamodel from such a specification. EMFText [56] is also similar to Xtext, but, similarly to TCS, it requires the language to be implemented to be defined in an abstract way using an EMF metamodel.

Other language workbenches with type system support are described in the next subsection.

## 7.3 Type systems frameworks

MPS (Meta Programming System) [57] is another tool for developing a DSL. MPS targets *projectional editing* for building DSL editors with tables and diagrams. MPS has its own DSL for specifying the type system. The developer specifies type system equations in this DSL and a solver tries to solve all the equations relevant to a given program in order to compute and check types. Spoofax [7] is another language workbench that targets Eclipse and it relies on Stratego [58] for rule-based specifications to analyze the programs. In [18], 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: NaBL [14] for name binding and scope rules, TS for the type system and DynSem [19] for the operational semantics. More recently, the NaBL2 language has been introduced, which is based on the constraint language described in [59]. The approach presented in [59] has been further generalized to structural and generic types in [60]: Statix is introduced, a domain-specific metalanguage for the specification of static semantics, based on scope graphs and constraints. With that respect, Xsemantics shares with the mentioned systems the goal of reducing the gap between the formalization and the implementation and it aims at being complementary to Xtext concerning the type system implementation. Moreover, we are currently working on extending Xsemantics with scoping rule definitions in order to have also the Xtext scope provider automatically generated.

Xtext only provides single inheritance mechanisms for grammars, so different grammars can be composed only linearly. The same holds for Xsemantics system extension, illustrated in [4]. These extensibility and compositionality features are not as powerful as frameworks like [7, 13, 61], where language specifications are fully modular.

Since Xsemantics targets mainly Xtext language implementations, the work that is closest to Xsemantics is XTS [62], which shares with Xsemantics the main goals. However, XTS aims at expression based languages, not at general purpose languages, and it would not be straightforward to write the type system for FJ in XTS. Moreover, XTS targets type systems only, while Xsemantics can be used also for writing reduction rules (e.g., for compilers and interpreters) as sketched in [4].

Systems like Silver [63], JastAdd [13] (and its integration with EMF, JastEMF [64]) and LISA [65], allow the developer to specify type systems using *attribute grammars* [66], that is, associating attributes with AST elements. These attributes can capture arbitrary data about the element including its type. Xsemantics shares something with attribute grammars: it associates a type attribute with program elements. Since Xsemantics aims at defining the semantics of a language, it provides a more concise notation and a more specific error reporting mechanism. However, in contrast to general-purpose attribute grammar systems, Xsemantics only aims at rules for semantics specification. Moreover, since Xsemantics specifications are not part of the grammar specifications, Xsemantics type systems could be "injected" in existing implementations, or different implementations of type systems can be enabled in a DSL. Indeed, Xsemantics is not bound to an Xtext grammar: it refers to Java types representing nodes of the AST.

In fact, Xsemantics might also be used to validate any model, independently from Xtext itself, and possibly be used also with other language frameworks like EMFText [56]. This is an advantage with respect to other approaches (e.g., [57, 61, 67–71]), which instead require the programmer to use the framework also for defining the syntax of the language.

Xsemantics is the successor of Xtypes [72] and it provides a much richer syntax for rules, thanks to Xbase. Moreover, Xtypes targets type systems only, while Xsemantics deals with any kind of rules. The very first prototype of Xsemantics was introduced in [73], where it

was used, together with other frameworks, in an analysis of several approaches for implementing type systems for Xtext DSLs. Then, in [74], 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 [75]. Moreover, it has been used to implement a typed Javascript dialect N4JS [22, 76], that is, a super set of JavaScript with modules and classes with a static type system on top of it, which combines the type systems provided by Java, TypeScript and Dart. N4JS, besides primitive types, declared types such as classes, interfaces and roles, supports union types [77], generic types and generic methods, including wildcards, requiring the notion of existential types [78]. Thus its type system implementation represents a nice case study of the usability of Xsemantics for real world languages. The adoption of Xsemantics in the industry required to rewrite many of its parts so that complex type systems could be effectively and efficiently implemented [4]. This also led Xsemantics to become part of the Eclipse eco-system as an official Eclipse project https://github.com/eclipse/xsemantics.

Finally, we would like to stress that thanks to the complete integration of Xsemantics with Java, the developers are not forced to implement all the aspects of the semantics of a language with Xsemantics. Some specific tasks can be implemented directly in Java. Xsemantics and Java code can then co-exist and from Xsemantics one can delegate to Java code (as done, for example, in the implementation of N4JS).

## 8 Conclusions

In this paper we presented a few general patterns to implement a type system that is able to type as many parts of the program as possible, detecting only the most important type errors and avoiding cascading errors. We then presented two case studies related to the implementation of type systems, using such patterns, aiming at meaningful error reporting and useful IDE features. We also showed how the type system can be used for resolving cross references. We believe that the type error messages produced by such implementations have the typical properties found in the literature (see, e.g., [40, 79]).

One could start using such patterns from the very beginning of the language implementation, like we did for the Expressions DSL (Section 4). Alternatively, one could start implementing a first prototype of the type system following a formal specification, like we did for FJ (Section 5), and then concentrate on the improvements applying the patterns presented in the paper. In both cases, we believe that a good test suite, with good code coverage, is really crucial. This is what we actually did for the actual implementations (https://github.com/LorenzoBettini/xtext-type-errors-examples).

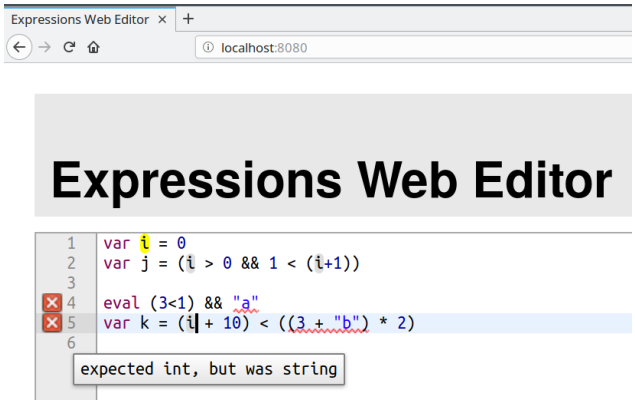


**Figure 38:** The web editor of the Expressions DSL, generated by Xtext.

Statically computed types are used also to enrich several parts of the IDE. The most important one is the content assist that relies on the static types to filter proposals, avoiding completions that would make the program invalid. The fact that the type system enjoys error recovery is crucial for implementing a useful content assist, since when the content assist is invoked, the program will surely be incomplete (the user is still writing the program). Other parts of the IDE can benefit from computed types, as we have shown throughout the paper, like Outline views, Hovering pop-ups and code mining.

Although the main target of Xtext is Eclipse, Xtext targets also other platforms. For example, a web editor for the DSL can be automatically generated by Xtext reusing the implementation of the DSL. Figure 38 shows the web editor generated by Xtext for the Expressions DSL, reusing all the mechanisms we implemented for the DSL (compare this figure with Figure 9).

The importance of IDE support for languages also recently led to the creation of the *Language Server Protocol* (LSP)[6], that is, a protocol to be used between an editor or IDE and a language server that provides the typical IDE features. The idea is to avoid repeating the effort to implement IDE mechanisms targeting different

6 https://microsoft.github.io/language-server-protocol/

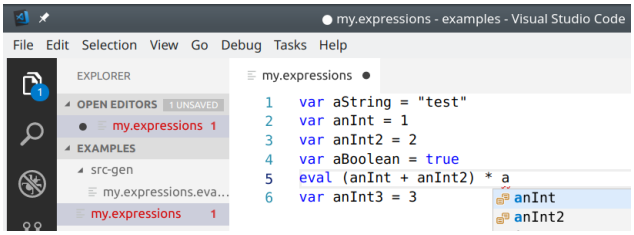


**Figure 39:** The Visual Studio Code editor of the Expressions DSL using LSP support by Xtext.

IDEs. A *Language Server* provides the language-specific features and communicates with the IDE over a protocol that enables inter-process communication. The same Language Server can be re-used in multiple IDEs, which, in turn, can support multiple languages with minimal effort. Eclipse and Xtext fully support LSP. Languages implemented with Xtext can automatically provide IDE mechanisms to Eclipse and to all the IDEs and editors that support LSP. Figure 39 shows the Expressions DSL editor in Visual Studio Code, relying on the LSP support provided by Xtext (compare this figure with Figure 13).

While we focused on language implementation targeting the IDE, Xtext can also automatically generate a command line compiler for the DSL, reusing all the implemented mechanisms (apart from the ones related to the UI). This is out of the scope of the paper. We only stress that the same useful error messages generated by the implementation shown in the paper would be reported on the console.

In the implementations shown in the paper we relied on Xtext and Xsemantics, exploiting their features for the IDE support and for mimicking formal systems, respectively. However, the patterns presented in Section 2 are general and could be applied in other language implementation frameworks.

**Acknowledgements:** I would like to thank the anonymous reviewers for their suggestions for improving the paper.

## References

[1] Fowler M., Language Workbenches: The killer-app for domain specific languages?, 2005, https://www.martinfowler.com/articles/languageWorkbench.html
[2] Bettini L., Implementing domain-specific languages with Xtext and Xtend, Packt Publishing, 2nd ed., 2016
[3] Igarashi A., Pierce B., Wadler P., Featherweight Java: a minimal core calculus for Java and GJ, ACM TOPLAS, 2001, 23(3), 396–450
[4] Bettini L., Implementing type systems for the IDE with Xsemantics, Journal of Logical and Algebraic Methods in Programming, 2016, 85(5, Part 1), 655–680
[5] Aho A. V., Lam M. S., Sethi R., Ullman J. D. (Eds.), Compilers: principles, techniques, and tools, Addison Wesley, 2nd ed., 2007
[6] Haase A., Völter M., Efftinge S., Kolb B., Introduction to open Architecture Ware 4.1.2, In: MDD Tool Implementers Forum, 2007
[7] Kats L. C. L., Visser E., The Spoofax language workbench, Rules for declarative specification of languages and IDEs, In: OOPSLA, ACM, 2010, 444–463
[8] Hemel Z., Groenewegen D. M., Kats L. C. L., Visser E., Static consistency checking of Web applications with WebDSL, Journal of Symbolic Computation, 2011, 46(2), 150–182
[9] Steinberg D., Budinsky F., Paternostro M., Merks E., EMF: Eclipse Modeling Framework, Addison-Wesley, 2nd ed., 2008
[10] Prasanna D. R., Dependency Injection: Design patterns using Spring and Guice, Manning, 1st ed., 2009
[11] Reps T., Teitelbaum T., The synthesizer generator, In: Software Engineering Symposium on Practical Software Development Environments, ACM, 1984, 42–48
[12] Sewell P., Nardelli F. Z., Owens S., Peskine G., Ridge T., Sarkar S., Strnisa R., Ott: Effective tool support for the working semanticist, Journal of Functional Programming, 2010, 20(1), 71–122
[13] Ekman T., Hedin G., The JastAdd system – modular extensible compiler construction, Science of Computer Programming, 2007, 69(1-3), 14–26
[14] Konat G., Kats L., Wachsmuth G., Visser E., Declarative name binding and scope rules, In: SLE, LNCS, Springer, 2012, 7745, 311–331
[15] Pierce B. C., Types and Programming Languages, Cambridge, MA: The MIT Press, 2002
[16] Cardelli L., Type systems, ACM Computing Surveys, 1996, 28(1), 263–264
[17] Hindley J. R., Basic Simple Type Theory, Cambridge University Press, 1987
[18] Visser E., Wachsmuth G., Tolmach A. P., Neron P., Vergu V. A., Passalaqua A., Konat G., A language designer's workbench: a one-stop-shop for implementation and verification of language designs, In: Onward!, ACM, 2014, 95–111
[19] Vergu V. A., Neron P., Visser E., DynSem: A DSL for dynamic semantics specification, In: RTA, LIPIcs, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015, 3, 365–378
[20] Efftinge S., Eysholdt M., Köhnlein J., Zarnekow S., Hasselbring W., von Massow R., Xbase: implementing domain-specific languages for Java, In: GPCE, ACM, 2012, 112–121
[21] Gosling J., Joy B., Steele G., Bracha G., Buckley A., The Java language specification, Java SE 7 Edition, Addison-Wesley, 2013
[22] Bettini L., von Pilgrim J., Reiser M.-O., Implementing a typed Java script and its IDE: a case-study with Xsemantics, International Journal on Advances in Software, 2016, 9(3-4), 283–303
[23] Johnson G. F. , Walz J. A., A maximum-flow approach to anomaly isolation in unification-based incremental type inference, In: POPL, ACM, 1986, 44–57
[24] Wand M., Finding the source of type errors, In: POPL, ACM, 1986, 38–43
[25] Heeren B., Top Quality Type Error Messages, PhD thesis, Utrecht University, Netherlands, 2005
[26] Lerner B. S., Flower M., Grossman D., Chambers C., Searching for type-error messages, In: PLDI, ACM, 2007, 425–434

[27] Chambers C., Chen S., Le D., Scaffidi C., The function, and dysfunction, of information sources in learning functional programming, Journal of Computing Sciences in Colleges, 2012, 28(1), 220–226
[28] Tirronen V., Uusi-Mäkelä S., Isomöttönen V., Understanding beginners' mistakes with Haskell, Journal of Functional Programming, 2015, 25
[29] Hudak P., Building domain-specific embedded languages, ACM Computing Surveys, 1996, 28(4es), 196
[30] Hage J., DOMain Specific Type Error Diagnosis (DOMSTED), Tech. Rep. UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014
[31] Heeren B., Hage J., Swierstra S. D., Scripting the type inference process, In: ICFP, ACM, 2003, 3–13
[32] Stuckey P. J., Sulzmann M., Wazny J., Improving type error diagnosis, In: SIGPLAN Workshop on Haskell, ACM, 2004, 80–91
[33] Wazny J., Type inference and type error diagnosis for Hindley/Milner with extensions, PhD thesis, University of Melbourne, Australia, 2006
[34] Plociniczak H., Miller H., Odersky M., Improving human-compiler interaction through customizable type feedback, Tech. Rep. 197948, EPFL, 2014
[35] Charguéraud A., Improving type error messages in OCaml, In: ML/OCaml, EPTCS, 2014, 198, 80–97
[36] Serrano A., Hage J., Type error diagnosis for embedded DSLs by two-stage specialized type rules, In: ESOP, LNCS, Springer, 2016, 9632, 672–698
[37] Serrano A., Hage J., Type error customization in GHC: Controlling expression-level type errors by type-level programming, In: IFL, ACM, 2017, 2:1–2:15
[38] Wu B., Chen S., How type errors were fixed and what students did?, In: OOPSLA, ACM, 2017, 1, 105:1–105:27
[39] Chen S., Erwig M., Early detection of type errors in C++ templates, In: PEPM, ACM, 2014, 133–144
[40] el Boustani N., Hage J., Improving type error messages for generic Java, Higher-Order and Symbolic Computation, 2011, 24(1), 3–39
[41] Damas L., Milner R., Principal type schemes for functional programs, In: POPL, ACM, 1982, 207–212
[42] Stuckey P. J., Sulzmann M., Wazny J., Interactive type debugging in Haskell, In: SIGPLAN Workshop on Haskell, ACM, 2003, 72–83
[43] Haack C., Wells J., Type error slicing in implicitly typed higher-order languages, Science of Computer Programming, 2004, 50(1), 189–224
[44] Hage J., Heeren B., Heuristics for type error discovery and recovery, In: IFL, LNCS, Springer, 2007, 4449, 199–216
[45] el Boustani N., Hage J., Corrective hints for type incorrect generic Java programs, In: PEPM, ACM, 2010, 5–14
[46] Chen S., Erwig M., Counter-factual typing for debugging type errors, In: POPL, ACM, 2014, 583–594
[47] Zhang D., Myers A. C., Vytiniotis D., Jones S. L. P., Diagnosing type errors with class, In: PLDI, ACM, 2015, 12–21
[48] Wu B., Campora III J. P., Chen S., Learning user friendly type-error messages, In: OOPSLA, ACM, 2017, 1, 106:1–106:29
[49] Bruch M., Monperrus M., Mezini M., Learning from examples to improve code completion systems, In: ESEC/SIGSOFT FSE, ACM, 2009, 213–222
[50] Parr T., Vinju J. J., Towards a universal code formatter through machine learning, In: SLE, ACM, 2016, 137–151
[51] Voelter M., Benz S., Dietrich C., Engelmann B., Helander M., Kats L. C. L., Visser E., Wachsmuth G., DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, 2013
[52] Pfeiffer M., Pichler J., A comparison of tool support for textual domain-specific languages, In: OOPSLA Workshop on Domain Specific Modeling, 2008, 1–7
[53] Erdweg S., van der Storm T., Völter M., Tratt L., Bosman R., Cook W. R., et al., Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems & Structures, 2015, 44(A)
[54] Charles P., Fuhrer R., Sutton Jr. S., Duesterwald E., Vinju J., Accelerating the creation of customized, language-Specific IDEs in Eclipse, In: OOPSLA, ACM, 2009, 191–206
[55] Jouault F., Bézivin J., Kurtev I., TCS: a DSL for the specification of textual concrete syntaxes in model engineering, In: GPCE, ACM, 2006, 249–254
[56] Heidenreich F., Johannes J., Karol S., Seifert M., Wende C., Derivation and Refinement of Textual Syntax for Models, In: ECMDA-FA, LNCS, Springer, 2009, 5562, 114–129
[57] Voelter M., Language and IDE modularization and composition with MPS, In: GTTSE, LNCS, Springer, 2011, 7680, 383–430
[58] Bravenboer M., Kalleberg K. T., Vermaas R., Visser E., Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming, 2008, 72(1-2), 52–70
[59] Van Antwerpen H., Néron P., Tolmach A., Visser E., Wachsmuth G., A constraint language for static semantic analysis based on scope graphs, In: PEPM, ACM, 2016, 49–60
[60] Van Antwerpen H., Bach Poulsen C., Rouvoet A., Visser E., Scopes as types, In: Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA), 114:1–114:30
[61] Vacchi E., Cazzola W., Neverlang: A Framework for Feature-Oriented Language Development, Computer Languages, Systems & Structures, 2015, 43(3), 1–40
[62] Völter M., Xtext/TS - a type system framework for Xtext, 2011, http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/
[63] Wyk E. V., Bodin D., Gao J., Krishnan L., Silver: an extensible attribute grammar system, Science of Computer Programming, 2010, 75(1), 39–54
[64] Bürger C., Karol S., Wende C., Aßmann U., Reference attribute grammars for metamodel semantics, In: SLE, LNCS, Springer, 2011, 6563, 22–41
[65] Mernik M., Lenic M., Avdicausevic E., Zumer V., LISA: an interactive environment for programming language development, In: Compiler Construction, LNCS, Springer, 2002, 2304, 1–4
[66] Knuth D. E., Semantics of context-free languages, Mathematical Systems Theory, 1968, 2(2), 127–145
[67] Borras P., Clement D., Despeyroux T., Incerpi J., Kahn G., Lang B., Pascual V., CENTAUR: the system, In: Software Engineering Symposium on Practical Software Development Environments, SIGPLAN, ACM, 1988, 24, 14–24
[68] Van den Brand M., Heering J., Klint P., Olivier P. A., Compiling language definitions: the ASF+SDF compiler, ACM TOPLAS, 2002, 24(4), 334–368
[69] Dijkstra A., Swierstra S. D., Ruler: programming type rules, In: FLOPS, LNCS, Springer, 2006, 3945, 30–46
[70] Felleisen M., Findler R. B., Flatt M., Semantics Engineering with PLT Redex, Cambridge, Mass.: The MIT Press, 2009
[71] Xu H., EriLex: An Embedded Domain Specific Language Generator, In: TOOLS, LNCS, Springer, 2010, 6141, 192–212

[72] Bettini L., A DSL for writing type systems for Xtext languages, In: PPPJ, ACM, 2011, 31–40
[73] Bettini L., Stoll D., Völter M., Colameo S., Approaches and tools for implementing type systems in Xtext, In: SLE, LNCS, Springer, 2012, 7745, 392–412
[74] Bettini L., Implementing Java-like languages in Xtext with Xsemantics, In: OOPS (SAC), ACM, 2013, 1559–1564
[75] Heiduk A., Skatulla S., From Spaghetti to Xsemantics - Practical experiences migrating type systems for 12 languages, XtextCon, 2015
[76] Bettini L., von Pilgrim J., Reiser M.-O., Implementing the Type System for a Typed Javascript and its IDE, In: COMPUTATION TOOLS, IARIA, 2016, 6–11
[77] Igarashi A., Nagira H., Union types for object-oriented programming, Journal of Object Technology, 2007, 6(2), 31–45
[78] Cameron N., Ernst E., Drossopoulou S., Towards an Existential Types Model for Java Wildcards, In: Formal Techniques for Java-like Programs (FTfJP), 2007
[79] Yang J., Michaelson G., Trinder P., Wells J. B., Improved Type Error Reporting, In: Workshop on Implementation of Functional Languages, Aachner Informatik-Berichte, 2000, 71–86