Automatic Generation of Object-Oriented Type Checkers

Francisco Ortin, Daniel Zapico, Jose Quiroga, and Miguel Garcia

***Abstract*—Type systems are aimed at preventing programming language constructions from having wrong behavior at runtime. Different formalisms are used to guarantee that a type system is well defined. However, the type systems implemented in commercial language processors (type checkers) do not commonly use tools that translate these formalisms into code. We propose a framework to facilitate the implementation of object-oriented type checkers, following widespread design patterns. A tool generates the specific implementation of a type checker, receiving a specification of the type system as its input. The generated code interacts with an API, accessible from the rest of the language processor implementation.**

***Index Terms*—Type checker, compiler construction, design patterns, semantic analysis, language implementation.**

I. Introduction

A *type* is a collection of objects having similar structure [1]. *Type theory* is the branch of mathematics and logic that concerns with classifying objects into types. With the appearance of computers and programming languages, type theory found practical application in the construction of type systems. A *type system* is defined as a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [2]. Therefore, a type system classifies program values into types, considering certain program constructions illegal on the basis of its type. For instance, a type system may prohibit passing a string value as an argument to a function that receives an integer parameter.

Typechecking is the analysis that detects semantic inconsistencies and anomalies, guarantying that entities match their declaration and preventing unsafe and ill-behaved programs from ever running. The algorithm that performs type checking is called *type checker* [3]. A language processor (e.g., compiler or interpreter) typically implements a *type checker* in its semantic (contextual) analysis phase [4].

Static (compile-time) type checking provides many benefits [2]. The most obvious one is *error detection,* which allows the early detection of programming errors, making possible to fix them immediately rather than discovering the type errors at runtime. Another common benefit is runtime performance, because many type operations are checked by the compiler, reducing the number of type checks performed at runtime [5], [6]. Types also enforce disciplined programming, defining a kind of partial contract between implementers and users. Additionally, type systems may involve better *safety*, as programs accepted by a safe type system will behave with the absence of runtime type errors [7]. Finally, types also constitute a form of *documentation*, specifying up-to-date information about the behavior of entities [2].

These are the most common benefits of the traditional use of type systems for language design and implementation. However, type systems have also been applied in multiple different scenarios, such as reverse engineering [8], testing [9], web metadata [10], and even security [11].

A. Type System Formalization

Type systems are formalized with precise notations and definitions. The detailed proof of formal properties (e.g., type safety) give confidence in the appropriateness of their definition [7]. Typical landmarks in type system formalization are the Russell's original ramified theory of types [1], Ramsey's simple theory of types [12], Church's typed lambda-calculus [13], Löf's constructive type theory [14], and Berardi's pure type systems [15]. *Type checkers* are sometimes specified by means of attribute grammars [16]. Based on these formal methods, a range of tools have been developed, achieving different goals:

- *Proof assistants* or interactive theorem provers are software tools to assist with the development of formal systems and their proofs, by means of human-machine collaboration. A formal system such as the type system and the semantics of a programming language can be defined with these tools. Theorems such as type safety can then be proved. Examples of two powerful proof assistants are Isabelle [17] and Coq [18].
- *Lightweight analysis and verification of programs*. These tools are sometimes referred to as *lightweight formal methods* because they do not prove the correctness of type systems. They commonly perform model checking in order to detect programming errors that are not ordinarily detected until runtime. Different examples of these tools are ESC/Java [19], SLAM [20] or SPIN [21]. PLT Redex is another lightweight mechanization tool designed for specifying, debugging and testing operational semantics and type systems of programming languages [22].
- *Language processor development tools*. Most of the existing compiler and interpreter construction tools are centered in the development of scanners and parsers. Others offer the development of *type checkers* and interpreters, employing different mechanisms. For instance, the Synthesizer Generator [23] supports attribute grammars with incremental attribute evaluation, whereas the ASF+SDF Meta-Environment is based on

Manuscript received December 16, 2013; revised April 1, 2014. This work was partially funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*.

The authors are with the Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007, Oviedo, Spain (e-mail: ortin@uniovi.es, quirogajose@uniovi.es, garciarmiguel@uniovi.es).

conditional term rewriting rules [24]. There are also works based on translating to code the inference rules that formalize a type system, such as TCG [25] and Tinker-Type [26].

B. *Implementation of Type Checkers*

The code generated by the tools mentioned above commonly follows the execution flow of the formalization used (e.g., a deductive system based on a set of inference rules), and it commonly involves lower runtime performance than an *ad hoc* implementation [27]. Besides, the generated code is usually difficult to debug and trace, because it comes from a general translation of mathematical specifications. In occasions, the formalizations used by the language designer (and hence the generated code) are unknown to the compiler programmer.

In this paper we present TyS, a framework to facilitate the implementation of object-oriented type checkers whose type soundness might be previously proven. The generated code follows widespread object-oriented design patterns to facilitate the integration with the available reusable code, components, tools and frameworks. The language independent design patterns used are easily understandable and maintainable by the language processor developer [28], and they model the reusable dynamic and static type checking features of existing type systems. Moreover, they offer a straightforward way to extend and adapt type checkers from the specific requirements of different programming language type systems.

II. Architecture

Fig. 1 shows the architecture of the TyS framework. A type checker specification text file identifies all the types in the type system, the subtyping relation, and all the operations (rules) supported by each type. This file is processed by TyCC, the type checker constructor. TyCC generates a specific implementation of the type checker described in the specification file, depending on the input parameters passed. These parameters include the exception class used to report type errors (e.g., `SemanticException` and `ParserException` in ANTLR [29]), the selected output language used to implement the generated type checkers, the class implementing the `TypeFactory` interface (Section IV), and the directory where the API is placed. Regardless the parameters passed to TyCC, the type checker implementation follows the same object-oriented design patterns. If there is any error in the specification file, TyCC shows the appropriate message and aborts the generation of the output type checker implementation.

The API provides the general services to use any type checker generated by TyCC. These services include a mechanism to represent type expressions, a type table and a type factory to reutilize objects representing the same type, and the runtime type exceptions used in the framework. The library has one implementation for each different output language supported. The functionality provided by the API can be used to implement both static and dynamic type checking (e.g., in compilers and interpreters).

Both the API and the generated type checker are used to implement the language processor, together with the rest of the implementation. TyS has been designed to be independent of the rest of phases and tools used in the language implementation. We have used TyS with both yacc/bison and ANTLR parser generators, and in single- and multi-pass language processors [30].

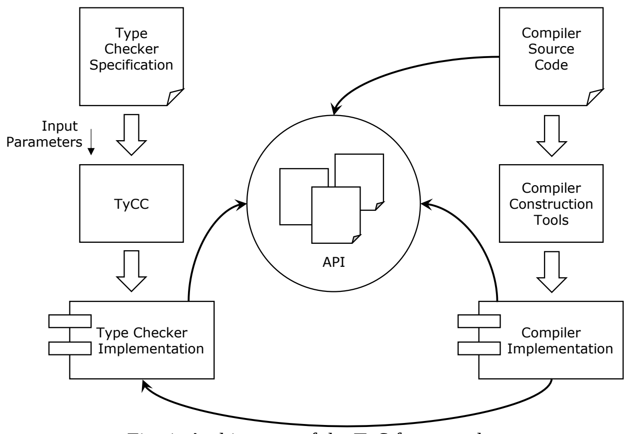
Type Checker Specification
Input Parameters
TyCC
Type Checker Implementation
API
Compiler Source Code
Compiler Construction Tools
Compiler Implementation

Fig. 1. Architecture of the TyS framework.

III. Specification of Types

We are going to describe the framework by specifying an example type checker of an imperative language, quite similar to the C programming language – the source code is available for download [30].

A. *Type Identification*

As mentioned, the types and its operations are described in a text file. We first identify the types and their subtyping relation. In our example we specify 9 different types:

```
Types = {
     Char < Int < Real
     Char < TString
     Bool < Int
     Int  < Bool
     Array
     Pointer
     Function
     Void
}
```

All the types must be declared in the `Types` block, and they can appear more than once. The subtyping relation is identified with the < symbol, indicating that the type on the left is a subtype (promotion, coercion or implicit cast) of the type on the right. In our example, `Char` is a subtype of `Int`, which in turn is a subtype of `Real`. Notice that the set of types can be partially ordered, that is, not every pair of types need be related (e.g., `Function` or `Void`).

The subtyping relation generates an `implicitCast (Type): Type` polymorphic method used to perform the implicit type coercions in the generated type checker. If the type represented by the object passed as the implicit parameter (`this`) promotes to the type represented by the argument, `implicitCast` returns the argument; `null` is returned otherwise. The behavior of the `implicitCast` operation can be overridden in particular types. Therefore, it is not mandatory to define all the subtyping rules with the < symbol. More complex type convention rules, such as covariant, invariant, and contravariant subtyping relations (e.g., for `Array` and `Function`), can be later defined as type operations (rules) –an example is presented later on in the following subsection.

B. Type Operations

After identifying the types and their basic subtyping relations, the specific operations (rules) of each type should be defined. The code generated by TyCC follows the *Composite* design pattern (Fig. 2) [31]. Any type is generalized with the `Type` interface, where all the available operations are declared. Two kinds of types can be defined: primitive types (leaf nodes in the *Composite* pattern) that represent a type by themselves; and composite types (intermediate nodes) that are constructed using other (primitive or composite) types.

The default implementations of the operations declared in `Type` are provided by the `BaseType` abstract class. These default implementations are inherited by all the derived classes. The particular behavior for a specific type can be modified by implementing the corresponding method in the derived classes, using dynamic binding (method overriding).

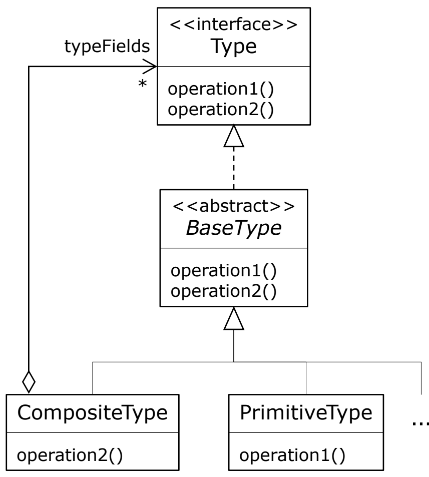

Fig. 2. Classes representing types, following the *Composite* design pattern.

A primitive type is simply declared by specifying the type identifier and, between curly braces, the list of the operations it provides. The following example is an excerpt of the `Int` type specification:

```
class Int {
  Type comparison(Type t) {
    if ( t.implicitCast(TypeTable.getInstance()
            .getType("Real")) == null )
      throw new TypeException("Int and "+t.getName()+
                              " cannot be compared");
    return TypeTable.getInstance().getType("Bool");
  }
}
```

The `comparison` operation defines the type rule for the six comparison operators (==, !=, <, <=, > and >=), defined over integers in our example language. If the type of the second operand is a subtype of `Real` (included), the comparison is correct and the `Bool` type is returned. Otherwise, a `TypeException` is thrown holding an error message (`getName` returns the type identifier). Objects representing types in TyS are obtained through a `TypeTable` class (detailed in Section IV), included in the framework API.

When an operation is defined in a type (e.g., `comparison`), it is included in the `Type` interface. The default implementation (in `BaseType`) throws an exception indicating that the operation is not applicable for that type. Therefore, the corresponding method could be called over any type, but only those types defining that operation will infer the resulting type –as we will see later on, the default implementation of the operations can also be changed.

Composite types are constructed using other (primitive or composite) types. This recursive composition is modeled in the *Composite* design pattern (Fig. 2) with an association from the composite type to the child ones (`typeFields`). The following excerpt defines the `Array` composite type in TyS:

```
class Array(Type) {
  String getName() {
    return getChildType().getName() + "[]";
  }
  Type getChildType() {
    return getTypeField(0);
  }
  Type implicitCast(Type t) {
    if (t instanceof Array && getChildType().
             .implicitCast(t.getChildType())!=null )
      throw new TypeException(getName()+
              "cannot be converted to "+t.getName());
    return t;
  }
  Type indexing(Type t) {
    if ( t.implicitCast(TypeTable.getInstance()
              .getType("Int")) == null )
      throw new TypeException(t.getName() +
                     " cannot be used as an index");
    return getChildType();
  }
}
```

The `Array` type declaration indicates that another child type is required by writing "`(Type)`" after its type identifier. More types can be specified using a separator (comma or blank space); and `Type+` represents a variable length collection of at least one type –scalar values can also be used [32]. In the generated `Array` class, a vector of `Types` (`typeFields`) is added as a private attribute, and so is the corresponding `getTypeField(int):Type` public method. In the `Array` example, `getTypeField(0)` returns the type of the elements of the array (its child type).

Two methods generated by TyCC are overridden in the `Array` type. The default behavior of `getName` (returning the "`Array`" string) is overridden; "`[]`" is added as a suffix to the type name of the element. Similarly, the behavior of `implicitCast` is also changed. In the example language, we define covariant subtyping for arrays: an array A_1 promotes to another array A_2 when the elements of A_1 promote to the elements of A_2 [2].

The `indexing` operation represents the inference rule of the `[]` operator. Since the first operand (`this`) is an `Array`, the only condition to be checked is that the second one is a subtype of `Int`. If so, the type of the array element is returned; an exception is thrown otherwise. In the example code provided [30], the `indexing` operation is also defined by the `Pointer` type. As in the C programming language, the `[]` operator can be applied to both arrays and pointers [30].

A fragment of the `Function` type specification is [30]:

```
class Function(Type+) {
  Type getReturnType() {
    return getTypeField(0);
  }
  int getNumberOfArguments() {
    return typeFields.size()-1;
  }
  Type checkArgument(int i, Type t) {
    if (i>getNumberOfArguments())
      throw new TypeException("Function " +getName()+
            " takes "+getNumberOfArguments()+" args");
    if (t.implicitCast(getTypeField(i)) == null)
      throw new TypeException("The " +i+ "th arg "+
            " is not a subtype of the parameter");
    return getTypeField(i);
  }
}
```

A function type comprises at least a return type plus a possibly empty collection of the types of its arguments (`Type+`). Therefore, we consider the first mandatory type in `typeFields` as the return type (`getReturnType`). Similarly, the `numberOfArguments` is the number of child types minus one, the return type.

The `checkArgument` operation checks whether an argument of type `t` can be passed as the i^{th} parameter. First, the function is tested to have at least `i` parameters. Second, the type of the argument (`t`) must be a subtype of the i^{th} parameter. If so, the type of the parameter is returned.

As shown in Fig. 2, `BaseType` holds the default behavior of all the operations defined for any type. The type checker developer can also use TyS to modify the default implementations given for any operation. The following specification defines two example default implementations, the `assignment` and `cast` type operations for every type:

```
class BaseType {
  Type assignment(Type t) {
    if (t.implicitCast(this) == null)
      throw new TypeException(t.getName() +
              " cannot be assigned to "+getName());
    return this;
  }
  Type cast (Type t) {
    String t1 = getName(), t2=t.getName();
    if (t1.equals("Void")||t1.equals("Function")||
        t2.equals("Void")||t2.equals("Function"))
      throw new TypeException("Cannot cast from " +
                  getName() + " to " + t.getName());
    return t;
  }
}
```

In an assignment, the type of the expression on the right must promote to the operand on the left, and the resulting type is the type of the operand on the left. The example default behavior of the `cast` operation is that any type but `Void` and `Function` can be cast to another type. This default behavior is then modified in `Array` (not shown for the sake of brevity), so that two arrays can be cast when the types of their corresponding elements can also be cast [30].

IV. Framework API

We have seen how types are specified by defining the operations they provide. We now describe how TyS can be used by the rest of components comprising a compiler implementation (Fig. 1). For this purpose, the TyS API provides services for building and managing the objects that represent the types specified by the programmer.

A. Obtaining Types

The `TypeTable` class in Fig. 3 is in charge of managing the objects representing types. A single instance of this class is created, following the *Singleton* design pattern (`getInstance():TypeTable`) [31]. The main method of this class is `getType(String):Type` that receives the type identifier as a parameter, and returns the object representing that type.

In our example [30], we have used the ANTLR tool for language recognition [29]. The following piece of code shows an excerpt of our type checker implementation:

```
type returns [Type t] { t= null;} :
  ( "int"  {t=TypeTable.getInstance().getType("Int");}
  | "real" {t=TypeTable.getInstance().getType("Real");}
  | "bool" {t=TypeTable.getInstance().getType("Bool");}
  | "char" {t=TypeTable.getInstance().getType("Char");}
  | "string" {t=TypeTable.getInstance()
                          .getType("TString");}
  )
  ( "[" "]" {t=TypeTable.getInstance()
                .getType("Array("+t.getName()+")"); }
  | "*"     {t=TypeTable.getInstance()
                .getType("Pointer("+t.getName()+")");}
  )*
  ;
```

The previous production defines the syntax of types, excluding functions, in our imperative language. The production returns the corresponding `Type` object. The first primitive type (`int`, `real`, `real`, `bool`, `char` or `string`) is retrieved from the type table, and stored in the `t` local variable. Afterwards, a repetition of zero o more `[]` or `*` symbols may appear, defining a composite type. The type identifier is built by concatenating the type constructor (`Array` or `Pointer`) with the child type (`t`) between brackets. For example, the type `int*[]` is represented with the string `"Array(Pointer(Int))"`. The returned object representing that type is an `Array` whose child type is a `Pointer` whose child type is an `Int`.

The design of the type table follows the *Flyweight* pattern [31]. As shown in Fig. 3, a `TypeTable` instance holds a collection of types. If the requested object type (`getType`) is in that collection, it is simply returned; otherwise, a new object representing the demanded type is created (calling the `createType` method of `TypeFactory`), added to the collection, and returned. The main objective of this design is the reutilization of objects representing the same type, avoiding high memory consumption [28].

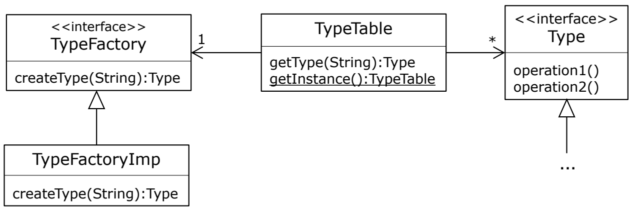

Fig. 3. Class diagram of the type table in the TyS API.

Objects representing types cannot be directly created by the programmer. The constructors of all the type classes generated by TyCC are declared with the Java default information hiding level (package). Therefore, new instances of these classes cannot be created from outside their package. Type factories (classes derived from `TypeFactory` in Fig. 3) are in charge of creating types with their `createType` method. They are in the same package as the type classes generated by TyCC. Type factories are not `public`, and they are used by the `public` class `TypeTable`. Therefore, the `getType` method of `TypeTable` is the only mechanism to create instances of types from outside their package, as commanded by the *Flyweight* pattern [31].

The `createType` method of `TypeFactory` must parse the string representing a type passed as a parameter, and return the appropriate `Type` object. For this purpose, the default implementation of this functionality (`TypeFactoryImp`) obeys the *Little Language* design pattern [33]. The language these strings follow has the next syntax:

$$
\begin{aligned}
&\textit{type} \rightarrow \textit{compositeType} \ \texttt{(}\ \textit{type} \ (\ \texttt{,}\ \textit{type})^{*}\ \texttt{)} \\
&\qquad\quad \mid \ \textit{primitiveType}
\end{aligned}
$$

The *primitiveType* and *compositeType* nonterminal symbols depend on the language been specified. In our example, *primitiveType*s are `Char`, `Int`, `Real`, `TString`, `Bool`, and `Void`; while *compositeType*s are `Array`, `Pointer`, and `Function`. For instance, if we pass the `"Function(Int,Array(Int),Char)"` string to the `createType` method of `TypeFactory`, the object graph shown in Fig. 4 is returned. The `tf` object represents the type of a function that returns an integer and receives two parameters: an array of `int`, and a `char`.

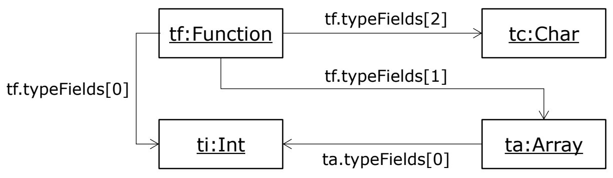

Fig. 4. Object diagram representing the `"Function(Int,Array(Int),Char)"` type.

It is worth noting that `TypeTable:getType` uses `TypeFactory:createType` and vice versa. The `getType` methods calls `createType` when the requested object is not in the collection of types stored in the `TypeTable`. Similarly, when `createType` is building a new composite type, it calls `getType` to retrieve its child types.

B. *Invoking Type Operations*

Once we have an object representing the appropriate type, we only need to call the required operations to perform type checking. Recall that all the operations in all the types are included as methods in the `Type` interface. If the corresponding operation is not applicable, a runtime exception is thrown.

The following fragment of our ANTLR type checker example, shows how the `[]` and `*` operators are checked:

```
rightOperator [Type arg] returns [Type t]
          { t = arg;
            Type typeOfIndex = null;
          } :
  ( "[" typeOfIndex = expression "]"
                { t = t.indexing(typeOfIndex); }
  | "*"         { t = t.getPointedType(); }
  )*
  ;
```

When the *rightOperator* is an expression between square brackets, the inferred type (`t`) is the result of calling the `indexing` operation against the type received as an argument (`arg`). If that type is an array, its child type is inferred; otherwise, an exception is raised. If the *rightOperator* is `*`, the `getPointedType` operation is called instead. That operation simply returns the child type of `Pointer`s, throwing an exception otherwise. This algorithm is executed in a loop, while a `[]` or `*` operator exists.

V. Implementation

The TyS framework has been developed in C++ (TyCC) and Java (the API). We have implemented different type checkers using the flex, bison/yacc and ANTLR compiler construction tools. These implementations comprise both single- and multi-pass language processors [34].

TyS has been used to implement different examples such as a simple arithmetical expression interpreter (Calc), an object-oriented programming language (Drill), a multi-pass imperative language compiler (Frog), and the example shown in this article (SubC) [30]. It has been used to implement the first prototype of the nitrO virtual machine [35], and its design principles have been applied in the development of the *StaDyn* programming language [36], [37]. It has also been used for educational purposes, in a compiler construction course [28].

The four examples mentioned in the previous paragraph (including the one presented in this article), the binaries and source code of TyS, and its documentation are available for download at http://www.reflection.uniovi.es/tys.

VI. Conclusion

The TyS framework proposes a collection of object-oriented design patterns, a type system construction tool, and an API to facilitate the implementation of imperative object-oriented type checkers. TyS has been used in the implementation of different kinds of language processors including those developed as single- and multi-pass compilers and processors that support both static and dynamic typing [30]. These implementations have used TyS in combination with different tools such as yacc/bison and ANTLR. The generated code is highly understandable, following widespread design patterns instead of translating code from a high-level formalization. Memory consumption has been considered, implementing a type table in the API that reutilizes the existing objects representing types.

Currently, TyCC only generates Java source code. We plan to generate other object-oriented languages such as C++ and C#. We also plan to include support for polymorphic types, including type variables and the implementation of a unification algorithm in the framework API [28].

Acknowledgments

This work was partially funded by the Department of

Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*.

REFERENCES

[1] A. N. Whitehead and B. Russell, *Principia Mathematica*, Cambridge University Press, 1910.

[2] B. C. Pierce. *Types and Programming Languages*, MIT Press, 2002.

[3] L. Cardelli, "Type systems," *The Computer Science and Engineering Handbook*, CRC Press, 2004.

[4] A. V. Aho, J. D. Ullman, and R. Sethi, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1985.

[5] F. Ortin, M. A. Labrador, and J. M. Redondo, "A hybrid class- and prototype-based object model to support language-neutral structural intercession," *Information and Software Technology*, vol. 56, no. 2, pp. 199-219, February 2014.

[6] J. M. Redondo, F. Ortin, and J. M. Cueva, "Optimizing reflective primitives of dynamic languages," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 6, pp. 759-783, September 2008.

[7] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and Computation*, vol. 115, no. 1, pp. 38-94, November 1994.

[8] C. Grothoff, J. Palsberg, and J. Vitek, "Encapsulating objects with confined types," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 6, October 2007.

[9] D. J. Richardson, "TAOS: Testing with analysis and Oracle support," in *Proc. International Symposium on Software Testing and Analysis*, August 1994, pp. 138-153.

[10] H. Hosoya and B. C. Pierce, "XDuce: A statically typed XML processing language," *ACM Transactions on Internet Technology*, vol. 3, no. 2, pp. 117-148, May 2003.

[11] M. Abadi, "Security Protocols and Specifications," *Foundations of Software Science and Computation Structures (FOSSACS), Lecture Notes in Computer Science*, vol. 1578, pp. 1-13, March 1999.

[12] F. P. Ramsey, "The foundations of mathematics," in *Proc. the London Mathematical Society*, vol. 25, 1925.

[13] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Verification and Synthesis*, vol. 5, no. 2, pp. 56-68, June 1940.

[14] P. Martin-L öf, *Intuitionistic Type Theory*, Bibliopolis, 1984.

[15] S. Berardi, "Towards a mathematical analysis of the Coquand-Huet Calculus of Constructions and the other systems in Barendregt's cube," Technical report, Department of Computer Science, Carnegie-Mellon University, 1988.

[16] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127-145, 1968.

[17] T. Nipkow. (December 2013). Programming and Proving in Isabelle/HOL. [Online] Available: http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2013-2/doc/prog-prove.pdf

[18] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, The MIT Press, November 2013.

[19] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe, *ESC/Java User's Manual*, Compaq Systems Research Center, 2000.

[20] T. Ball, and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," *Principles of Programming Languages (POPL)*, pp. 1-3, January 2002.

[21] G. J. Holzmann, *The Spin Model Checker*, Addison Wesley, 2003.

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

[23] T. W. Reps and T. Teitelbaum, "The synthesizer generator," *Software Engineering Symposium on Practical Software Development Environments (SDE)*, pp. 42-48, May 1984.

[24] M. Brand *et al.*, "The ASF+SDF Meta-environment: A Component-Based language development environment," in *Proc. Compiler Construction (CC), European Joint Conference on Theory and Practice of Software (ETAPS)*, April 2001, pp. 365-370.

[25] H. Gast, "A generator for type checkers," Ph.D. Thesis, Eberhard-Karls-Universit at T übingen, 2004.

[26] M. Y. Levin and B. C. Pierce, "TinkerType: A language for playing with formal systems," *Journal of Functional Programming*, vol. 13, no. 2, pp. 295-316, March 2003.

[27] M. L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

[28] F. Ortin, J. M. Cueva, and D. Zapico, "Patterns for teaching type checking in a compiler construction course," *IEEE Transactions on Education*, vol. 50, no. 3, pp. 273-283, August 2007.

[29] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed., Pragmatic Bookshelf, January 2013.

[30] F. Ortin and D. Zapico. (December 2013). A Framework to Facilitate the Development of Object-Oriented Type Checkers. *TyS*. [Online]. Available: http://www.reflection.uniovi.es/tys

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, October 1994.

[32] D. Zapico, "Design and implementation of a tool for the automatic generation of object-oriented type systems," Master Thesis, Polytechnic School of Engineering, University of Oviedo, March 2003.

[33] M. Grand, *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated With UML*, vol. I, Wiley, 1999.

[34] F. Ortin, B. Lopez, and J. B. G. Perez-Schofield, "Separating adaptable persistence attributes through computational reflection," *IEEE Software*, vol. 21, no. 6, pp. 41-49, November 2004.

[35] F. Ortin and D. Diez, "Designing an adaptable heterogeneous abstract machine by means of reflection," *Information and Software Technology*, vol. 47, no. 2, pp. 81-94, February 2005.

[36] F. Ortin, D. Zapico, J. B. B. Perez-Schofield, and M. Garcia, "Including both static and dynamic typing in the same programming language," *IET Software*, vol. 4, no. 4, pp. 268-282, August 2010.

[37] F. Ortin and M. Garcia, "Union and intersection types to support both dynamic and static typing," *Information Processing Letters*, vol. 111, no. 6, pp. 278-286, February 2011.

Francisco Ortin was born in 1973. He is an associate professor of the Computer Science Department at the University of Oviedo, Spain. He is the head of the Computational Reflection Research group. He received his B.Sc in computer science in 1994, and his M.Sc in computer engineering in 1996. In 2002, he was awarded his PhD entitled "A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine". He has been the principal investigator of different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include dynamic languages, type systems, aspect-oriented programming, computational reflection, and runtime adaptable applications.

Daniel Zapico was born in 1975. He is a Ph.D. student of the Computer Science Department at the University of Oviedo. He received his B.Sc degree in computer science in 1999, and an M.Sc in computer engineering in 2003. After working in different software companies, he has returned to the University of Oviedo as a PhD student, receiving funds from the Spanish Department of Science and Innovation. His PhD thesis is focused on the definition of type systems and its implementation in production language processors. He is the main developer of TyS.

Jose Quiroga was born in 1982. He is a research assistant of the Computer Science Department at the University of Oviedo. He was awarded his B.Sc degree in computer science in 2004 and, in 2009, the M.Sc in computer engineering. He worked in the Research and Development Department of the CTIC Foundation until 2012, when he became a Research Assistant financed by the Spanish Department of Competitiveness and Productivity. As a Ph.D. student, he is doing research on the optimization of dynamically typed programming languages, performing compile-time inference of type information.

Miguel Garcia was born in 1979. He is a postdoctoral research assistant of the Computer Science Department at the University of Oviedo. He received his B.Sc degree in computer science in 2005. In 2008 he was awarded an MSc in Web Engineering, and an M.Sc in software engineering research in 2010. In 2013 he presented his Ph.D. dissertation entitled *Improving the Runtime Performance and Robustness of Hybrid Static and Dynamic Typing Languages*. His research interests include compiler construction, programming languages design, and aspect-oriented software development.