Zhi# – OWL Aware Compilation

Alexander Paar[1] and Denny Vrandečić[2]

[1] University of Pretoria, Hillcrest, Pretoria 0002, South Africa
alexpaar@acm.org
[2] KIT Karlsruhe Institute of Technology, Karlsruhe, Germany, and
ISI Information Sciences Institute, University of Southern California, USA
denny.vrandecic@kit.edu

Abstract. The usefulness of the Web Ontology Language to describe domains of discourse and to facilitate automatic reasoning services has been widely acknowledged. However, the programmability of ontological knowledge bases is severely impaired by the different conceptual bases of statically typed object-oriented programming languages such as Java and C# and ontology languages such as the Web Ontology Language (OWL). In this work, a novel programming language is presented that integrates OWL and XSD data types with C#. The Zhi# programming language is the first solution of its kind to make XSD data types and OWL class descriptions first-class citizens of a widely-used programming language. The Zhi# programming language eases the development of Semantic Web applications and facilitates the use and reuse of knowledge in form of ontologies. The presented approach was successfully validated to reduce the number of possible runtime errors compared to the use of XML and OWL APIs.

Keywords: OWL DL, C#, Zhi#, ontologies, programmability

1 Introduction

A typical OWL DL [12] knowledge base comprises two components: a *TBox* defining the formal relations between the classes and properties of the ontology; and an *ABox* containing assertional knowledge about the individuals of the ontology. The TBox is often regarded to be the more stable part of the ontology, whereas the ABox may be subject to occasional or even constant change. In particular, modifications may lead to an ABox that violates constraints given by the TBox, such as cardinality constraints or value space restrictions of OWL datatype properties. Up to now, ontological knowledge bases are modified using APIs, which are provided by a variety of different ontology management systems [7,21]. From a software developer's perspective, there is no support for statically detecting illegal operations based on given terminologies (e.g., undefined classes, invalid datatype property values) and conveniently integrating ontological classes, properties, and individuals with the program text of a general purpose programming language.

A common difficulty of widely used OWL APIs and the usage of wrapper classes to represent entities of an ontology are the different conceptual bases of types and instances in a programming language and classes, properties, individuals, and XML Schema Definition [4] data type values in OWL DL. In particular, the Web Ontology Language reveals the following major differences to object-oriented programming languages and database management systems:

- In contrast to object-oriented programming languages, OWL provides a rich set of class constructors. For example, classes can be described via cardinality and value restrictions on properties (e.g., a small meeting is a meeting with at most three participants).
- OWL class descriptions can be automatically classified in a subsumption hierarchy. Imitating this inherent behavior of ontological knowledge bases using a hierarchy of programming language wrapper classes would result in reimplementing a complete OWL DL reasoner.
- Unlike object-oriented programming languages or database management systems, OWL makes the *open world assumption* (OWA), which codifies the informal notion that in general no single observer has complete knowledge. The open world assumption limits the deductions a reasoner can make. In particular, it is not possible to infer that a statement is false just because it is not stated explicitly. The OWA is closely related to the monotonic nature of first-order logic (i.e. adding information never falsifies previous conclusions).
- The Web Ontology Language does not make the *unique name assumption* (UNA). In contrast to logics with the unique name assumption, different ontological individuals do not necessarily refer to different entities in the described world. In fact, two individuals can be inferred to be identical (e.g., values of functional object properties). In OWL, it is also possible to explicitly declare that two given named individuals refer to the same entity or to different entities.
- Unlike object-oriented programming languages, ontological properties in OWL DL are not defined as part of class definitions but form a hierarchy of their own (i.e. property centric modeling).
- In OWL, property domain and range declarations are not constraining. Instead, the declared domain and range of an OWL property is used to infer the types of the subjects and objects of assertions, respectively. Thus, OWL properties facilitate *ad hoc relationships* [13] between entities that may not have been foreseen when a class was defined.

The Zhi#[3] programming language is a superset of conventional C# version 1.0 boasting programming language inherent support for XML Schema Definition and the Web Ontology Language. Zhi#'s *OWL aware compilation* includes static typing and type inference for XSD data types and a combination of static typing and dynamic checking for OWL DL ontologies. XSD constraining facets and ontological reasoning were integrated with host language features such as method overriding, user-defined operators, and runtime type checks. For the lack of space,

[3] Zhi (Chinese): Knowledge, information, wisdom.

only elementary examples of an integrated use of XSD data types and ontological class descriptions in Zhi# are presented. The Zhi# programming language is implemented by a compiler framework [16] that is – by means of plug-ins – extensible with external type systems[4]. Detailed descriptions of the compiler framework and the XSD and OWL plug-ins can be found in [17]. Zhi# programs are compiled into conventional C# and are interoperable with .NET assemblies. The Zhi# approach is distinguished by a combination of features that is targeted to make ontologies available in an object-oriented programming language using conventional object-oriented notation.

In contrast to naïve approaches that are based on the generation of wrapper classes for XSD and OWL types, no code generation in form of an additional class hierarchy is required in Zhi#. Instead, ontologies are integrated into the programming language, which facilitates OWL aware compilation including type checking on the ontology level. At runtime, the results of ontological reasoning influence the execution of Zhi# programs: Zhi# programs don't just execute, they reason. The underlying ontology management system can be substituted without recompilation of Zhi# programs. The Zhi# programming language provides full support for XSD data types. Thus, Zhi# can compensate for datatype agnostic OWL APIs. Zhi# programs can be used concurrently with API-based knowledge base clients to allow for a smooth migration of an existing code-base.

2 The Zhi# Programming Language

The type system of the C# programming language implements *nominal* subtyping. In nominative type systems type compatibility is determined by explicit declarations. A type is a subtype of another if and only if it is explicitly declared to be so in its definition. The XML Schema Definition type system extends nominal subtyping with *value space-based* subtyping. An atomic data type is a subtype of another if it is explicitly declared to be so in its definition or if its value space (i.e. the set of values for a given data type) is a subset of the value space of the other type. The subset relation of the types' value spaces is sufficient. The two types do not need to be in an explicitly declared derivation path. In the Web Ontology Language, nominal subtyping is augmented by *ontological reasoning*. An inferred class hierarchy can include additional subsumption relations between class descriptions. Ontological individuals can be explicitly declared to be of a given type and they can be inferred to be in the extension of further class descriptions. Some object-oriented programming languages provide a limited set of isomorphic mappings from XSD data types to programmatic types. In general, however, compilers for programming languages such as Java or C# are unaware of the subtyping mechanisms that are used for XSD and OWL.

The Zhi# programming language is a proper superset of ECMA 334 standard C# version 1.0 [6]. The only syntactical extensions, which are entailed by Zhi#'s

[4] Given the general extensibility of the Zhi# programming language with external type systems and for the sake of brevity, in this work, XSD data types and OWL class descriptions are subsumed under the term *external types*.

extensibility with respect to external type systems, are the following: External types (i.e. XSD data types and OWL class descriptions) can be included using the keyword *import*, which works analogously for external types like the C# *using* keyword for .NET programming language type definitions. It permits the use of external types in a Zhi# namespace such that, one does not have to qualify the use of a type in that namespace. An *import* directive can be used in all places where a *using* directive is permissible. As shown below, the *import* keyword is followed by a *type system evidence*, which specifies the external type system (i.e. XSD or OWL). Like *using* directives, *import* directives do not provide access to any nested namespaces.

import *type_system_evidence alias* = *external_namespace*;

In Zhi# program text that follows an arbitrary number of *import* directives, external type and property references must be fully qualified using an alias that is bound to the namespace in which the external type is defined. Type and property references have the syntactic form *#alias#local_name* (both the namespace alias and the local name must be preceded by a '#'-symbol).

External types can be used in Zhi# programs in all places where .NET types are admissible except for type declarations (i.e. external types can only be imported but not declared in Zhi# programs). For example, methods can be overridden using external types, user defined operators can have external input and output parameters, and arithmetic and logical expressions can be built up using external objects. Because Zhi#'s support for external types is a language feature and not (yet) a feature of the runtime, similar restrictions to the usage of external types apply as for generic type definitions in the Java programming language (e.g., methods cannot be overloaded based on external types from the same type system at the same position in the method signature).

In Zhi# programs, types of different type systems can cooperatively be used in one single statement. As shown in line 5 in the following code snippet, the .NET *System.Int32* variable *age* can be assigned the XSD data type value of the OWL datatype property *hasAge* of the ontological individual Alice.

```
1 import OWL chil = http://chil.server.de;
2 class C {
3   public static void Main() {
4     #chil#Person alice = new #chil#Person("#chil#Alice");
5     int age = alice.#chil#hasAge;
6  }
7 }
```

2.1 Static Typing

C# is a statically typed programming language. Type checking is performed during compile time as opposed to runtime. As a consequence, many errors can be caught early at compile time (i.e. fail-fast), which allows for efficient execution at runtime. This section describes the static type checks that can be performed on ontological expressions in Zhi# programs.

Syntax checks. The most fundamental compile-time feature that Zhi# provides for OWL is checking the existence of referenced ontology elements in the imported terminology. The C# statements below declare the ontological individuals A and B. Individual B is added as a property value for property R of individual A. For the sake of brevity, in this work, the URI fragment identifier "#" may be used to indicate ontology elements in Zhi# programs instead of using fully-qualified names. The object o shall be an instance of an arbitrary OWL API. The given code is a well-typed C# program. It may, however, fail at runtime if in the TBox of the referenced ontology classes A and B and property R do not exist.

```
1 IOWLAPI o = [...];
2 o.addIndividual("#A", "#A");
3 o.addIndividual("#B", "#B");
4 o.addObjectPropertyValue("#A", "#R", "#B");
```

In Zhi#, the same declarations can be rewritten as shown below, turning the ontological properties into first-class citizens of the programming language. As a result, the Zhi# compiler statically checks if class descriptions A and B and property R exist in the imported ontology and raises an error if they are undefined. Note how in line 4 the RDF triple [A R B] is created in the shared ontological knowledge base using object-oriented member access.

```
1 import OWL alias = ontology namespace;
2 #A a = new #A("#A");                    // variable a refers to individual A
3 #B b = new #B("#B");                    // variable b refers to individual B
4 a.#R = b;
```

Creation of individuals. In C#, the *new*-operator can be used to create objects on the heap and to invoke constructors. In Zhi#, the *new*-operator can also be used to return ontological individuals in a knowledge base as follows.

```
1 #Event e = new #Meeting("#Briefing"); // Because Meeting ⊑ Event...
2 #Meeting m = new #Event("#Briefing"); // ...this assignment is rejected
```

Zhi# provides a constructor for OWL class instances that takes the URI of the individual. As in conventional C#, the *new*-operator cannot be overloaded. In contrast to .NET objects, ontological individuals are not created on the heap but in the shared ontological knowledge base, and as such they are subject to ontological reasoning. This is also in contrast to naïve approaches where wrapper classes for ontological classes are instantiated as plain .NET objects. Zhi# programs use handles to the actual individuals in the shared ontological knowledge base. Also note that an existing individual in the ontology with the same URI is reused, following Semantic Web standards. As for assignments of .NET object creation expressions to variables or fields, the type of the individual creation expression must be subsumed by the type of the lvalue based on the class hierarchy (see line 2 in the code snippet above). Zhi# supports covariant coercions for ontological individuals and arrays of ontological individuals.

Disjoint classes. In OWL DL, classes can be stated to be disjoint from each other using the *owl:disjointWith* constructor. It guarantees that an individual that is a member of one class cannot simultaneously be a member of the other class. In the following code snippet, the Zhi# compiler reports an error in line 2 for the disjoint classes *MeetingRoom* and *LargeRoom*.

```
1 #LargeRoom l = [...];                    // LargeRoom ⊑ ¬MeetingRoom
2 #MeetingRoom m = (#MeetingRoom) l;       // ⇝ InvalidCastException
```

If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. Assuming the OWL object property *takesPlaceInAuditorium* relates *Lecture*s with *LargeRoom*s, line 2 in the following code snippet results in a compile-time error due to the disjointness of *MeetingRoom* and *LargeRoom*. Property domain declarations are treated analogously.

```
1 #Lecture l = [...];
2 l.#takesPlaceInAuditorium = new #MeetingRoom([...]);
```

Disjoint XSD data types. In Zhi#, a "frame-like" view on OWL object properties is provided by the *checked*-operator used in conjunction with assignments to OWL object properties (see Section 2.2). For assignments to OWL datatype properties in Zhi# programs, the "frame-like" composite view is the default behavior. The data type of the property value must be a subtype of the datatype property range restriction. The following assignment in line 2 fails to type-check for an OWL datatype property *hasCapacity* with domain *MeetingRoom* and range *xsd#byte* because in Zhi# programs the literal *23.5* is interpreted as a .NET floating point value (i.e. *xsd#double*), which is disjoint with the primitive base type of *xsd#byte* (i.e. *xsd#decimal*).

```
1 #MeetingRoom r = [...];
2 r.#hasCapacity = 23.5;
```

The XSD compiler plug-in allows for downcasting objects to compatible XSD data types (i.e. XSD types that are derived from the same primitive base type). The assignment in line 3 in the following Zhi# program is validated by a downcast. In general, this may lead to an *InvalidCastException* at runtime, which prevents OWL datatype properties from being assigned invalid property values.

```
1 int i = [...];
2 #MeetingRoom r = [...];
3 r.#hasCapacity = (xsd#byte) i;
```

Properties. Erik Meijer and Peter Drayton note that "at the moment that you define a [programming language] class *Person* you have to have the divine insight to define all possible relationships that a person can have with any other possible object or keep type open" [13]. Ontology engineers do not need to make early commitments about all possible relationships that instances of a class may have. In Zhi# programs, ontological individuals can be related to other individuals

and XSD data type values using an object-oriented notation. In contrast to authoritative type declarations of class members in statically typed object-oriented programming languages, domain and range declarations of OWL object properties are used to infer the types of the subject (i.e. host object) and object (i.e. property value). Hence, the types of the related individuals do not necessarily need to be subsumed by the domain and range declarations of the used object property *before* the statement. The only requirement here is that the related individuals are not declared to be instances of classes disjoint to the declared domain and range. In the following Zhi# program, the ontological individuals referred to by e and l are inferred to be not only an *Event* and a *Location* but also a *Lecture* and a *LargeRoom*, respectively.

```
1 #Event e = [...];                    // e refers to E, E:Event
2 #Location l = [...];                 // l refers to L, L:Location
3 e.#takesPlaceInAuditorium = l;       // E:Lecture, L:LargeRoom
```

Both for OWL object and non-functional OWL datatype properties the property assignment semantics in Zhi# are *additive*. The following assignment statement *adds* the individual referred to by b as a value for property R of the individual referred to by a; it does not remove existing triples in the ontology.

```
a.#R = b;
```

Correspondingly, property access expressions yield arrays of individuals and arrays of XSD data type values for OWL object properties and non-functional OWL datatype properties, respectively, since an individual may be related to more than one property value. Accordingly, the type of OWL object property and non-functional OWL datatype property access expressions in Zhi# is always an array type, where the base type is the range declaration of the property.

The type of an assignment to an OWL object property and a non-functional OWL datatype property is always an array type, too. This behavior is slightly different from the typical typing assumptions in programming languages. Because the assignment operator (=) cannot be overloaded in .NET, after an assignment of the form $x = y = z$ all three objects can be considered equal based on the applicable kind of equivalence (i.e. reference and value equality). The same is not always true for assignments to OWL properties considering the array ranks of the types of the involved objects. In the following cascaded assignment expression, the static type of the expression $b.\#R = c$ is `Array Range`(R) because individual B may be related by property R to more individuals than only C. As a result, with the following assignment in Zhi#, individual A is related by property R to *all* individuals that are related to individual B by property R.

```
a.#R = b.#R = c; // a, b, and c refer to individuals A, B, and C, respectively
```

Ontological equality. In Zhi#, the equality operator (==) can be used to determine if two ontological individuals are identical (i.e. refer to the same entity in the described world). The inequality operator (!=) returns true if two individuals are known (either explicitly or implicitly) to be *not* identical. Note that the

inequality operator is thus *not* implemented as the logical negation of == as individuals can be unknown to be identical or different.

Auxiliary properties and methods. The Zhi# compiler framework supports a full-fledged object-oriented notation for external types. In particular, compiler plug-ins can provide methods, properties, and indexers for static references and instances of external types. The OWL compiler plug-in implements a number of auxiliary properties and methods for ontological classes, properties, and individuals in Zhi# programs. For example, in order to remove property value assertions from the ontology, the OWL compiler plug-in provides the auxiliary methods *Remove* and *Clear* for OWL properties to remove one particular value and all values for an OWL property of the specified individual, respectively. In line 2 in the following code snippet, the ontological individual B is removed as a property value for property R of individual A. In line 3, all property values for property R of individual A are removed.

```
1  #A a = new #A("#A");          // Note: Other clients of the ontological know-
2  a.#R.Remove(new #B("#B"));    // ledge base could have added B as a property
3  a.#R.Clear();                 // value between the execution of line 1 and 2.
```

As a second example, for static references of OWL classes the auxiliary properties *Exists*, *Count*, and *Individuals* are defined. The *Exists* property yields true if individuals of the given type exist in the ontology, otherwise false. *Count* returns the number of individuals in the extension of the specified class description. *Individuals* yields an array of defined individuals of the given type. The *Individuals* property is generic in respect of the static type reference on which it is invoked. In the following array definition, the type of the property access expression *#Person.Individuals* is `Array` *Person* (and not `Array` *Thing*). Accordingly, it can be assigned to variable *persons* of type `Array` *Person*.

```
#Person[] persons = #Person.Individuals;
```

Note that all described functionality is provided in a "pay-as-you-go" manner: in Zhi#, there is no runtime performance or code size overhead for conventional C# code and Zhi# programs that do not use external type definitions.

2.2 Dynamic Checking

In a statically typed programming language such as C# the possible types of an object are known at compile time. Unfortunately, the non-contextual property centric data modeling features of the Web Ontology Language render static type checking only a partial test on Zhi# programs. As a result, the OWL plug-in for the Zhi# compiler framework and the Zhi# runtime library facilitate dynamic checking of ontological knowledge bases.

Ontological individuals can be in the extensions of a number of different class descriptions. In the same way, explicitly made RDF type assertions may be inconsistent with particular property values or the number of values for a particular property of an individual. More severely, ontological knowledge bases are

subject to concurrent modifications via interfaces of different levels of abstraction (e.g., RDF triples, logical concept view). In Zhi#, before each single usage of an individual 1) the individual is dynamically checked to be in the extension of the declared class and 2) the knowledge base is checked to be consistent; an exception is thrown if either is not the case.

Runtime type checks. Reasoning is used to infer the classes an individual belongs to. This corresponds to the use of the *instanceof* and *is*-operator in Java and C#, respectively. In Zhi#, the *is*-operator is used to determine whether an individual is in the extension of a particular class description. The use of the *is*-operator is completely statically type-checked both on the programming language and the ontology level. For example, the Zhi# compiler will detect if an individual will never be included by a class description that is disjoint with its asserted type. See the Zhi# program in Section 3 for an exemplary use of the *is*-operator.

Checked property assignments. In general, neither domain nor range declarations of OWL properties are constraints. This is in contrast to frame languages and object-oriented programming languages. In statically typed object-oriented programming languages such as C#, properties are declared as class members. The domain of a property corresponds to the type of the containing host object. Only instances of the domain type can have the declared property. The range of a property (i.e. class attribute) is also given by an explicit type declaration. This type declaration is constraining, too. All objects that are declared to be values of a property must be instances of the declared type at the time of the assignment. Many ontology engineers favor a rather frame-like composite view of classes and their associated properties, too. Indeed, the advantage of using property domain and range descriptions to constrain the set of conforming RDF triples is a more succinct structuring of an ontology or schema. In Zhi#, the *checked*-keyword, which can be used as an operator or a statement, supports the frame-like notion of OWL object properties. The following example demonstrates the *checked*-operator on an OWL object property assignment expression. For an OWL object property *takesPlaceInAuditorium*, which relates *Lecture*s with *LargeRoom*s, an exception is thrown at runtime if the individuals referred to by e and l are not in the extensions of the named class descriptions *Lecture* and *LargeRoom*, respectively.

```
1 #Event e = [...];                                  // Lecture ⊑ Event
2 #Location l = [...];                               // LargeRoom ⊑ Location
3 e.#takesPlaceInAuditorium = checked(l);  // ⇝ InvalidCastException
```

XSD and OWL were integrated with the Zhi# programming language similarly like generic types in Java. XSD data types and OWL DL class descriptions in Zhi# programs are subject to type substitution where references of external types are translated into 1) a constant set of proxy classes and 2) function calls on the Zhi# runtime library and its extensions for XSD and OWL. Detailed explanations how Zhi# programs are compiled into conventional C# are given in [17].

3 Validation

The Zhi# compiler was regression tested with 12 KLOC of Zhi# test code, which was inductively constructed based on the Zhi# language grammar, and 9 KLOC of typing information to regression test the semantic analysis of Zhi# programs.

The ease of use of ontological class descriptions and property declarations in Zhi# is illustrated in [17] by contradistinction with C#-based implementations of "ontological" behavior for .NET objects. The Zhi# approach facilitates the use of readily available ontology management systems compared to handcrafted reasoning code.

Examples for the advantage of *OWL aware compilation* in the Zhi# programming language over an API-based use of ontology management systems can be shown based on the following programming tasks, which are all frequent for ontology-based applications. Assume the following TBox.

$A,\ B \sqsubseteq \neg C,\ \geq 1R \sqsubseteq A,\ \top \sqsubseteq \forall R.B,\ \top \sqsubseteq \leq 1R,\ \top \sqsubseteq \forall U.xsd\#positiveInteger$

Task 1 Make an ontology available in a computer program.
Task 2 Create individual instances A, O, B, and C of classes $\top$, B, and C.
Task 3 Add individual O as a value for property R of individual A.
Task 4a) List the RDF types of individual O.
Task 4b) Check whether individual O is included by class description B.
Task 4c) List all individuals in the extension of class description B.
Task 5 Add individual C as a value for property R of individual A, which causes an inconsistent ABox since class descriptions B (range of property R) and C (which includes C) are disjoint.
Task 6 Add individual B as a value for property R of individual A and test if individuals O and B are equal (i.e. is there an inferred $sameAs(\text{O}, \text{B})$ statement in the ontology?).
Task 7 Add literals 23, -23, and string literal "NaN" as values for property U of individual O, where -23 and "NaN" are invalid values for the given TBox.

In line 1 in the Zhi# program shown below, the given TBox, which is defined in the *http://www.zhimantic.com/eval* namespace, is imported into the Zhi# compile unit. In line 2, XML Schema Definition's built-in data types are imported. In lines 4–9, ontological individuals are created. In line 9, the fully qualified name of individual C is inferred from the containing namespace of the named class description C. In line 10, the RDF triple [A R O] is declared in the ontology. Note how Zhi# facilitates the declaration of ad hoc relationships (instead of enforcing a frame-like view, where *Thing* A would not have a slot R). In line 11, a *foreach*-loop is used to iterate over all values of the auxiliary *Types* property of individual O. Note, in line 12, how ontological individuals are implicitly convertible to .NET *string*s. In line 15, the *is*-operator is used to dynamically check the RDF type of individual O. Be aware that the type check is performed on the ontology level. In line 16, a *foreach*-loop iterates over all values of the auxiliary *Individuals* property of the static class reference B. Note that the *Individuals* property is generic with respect to the static class reference on

which it is invoked. The assignment statement in line 19 causes a compile-time error since individual C cannot simultaneously be in the extension of class C (its declared type) and B (the range restriction of property R) for consistent ontologies. In line 22, individuals O and B are compared for equality. Note that the equality operator (==) evaluates on the ontology level (i.e. $o == b$ evaluates to *true* since O and B were both used as values for functional property R). In lines 23–25, XSD data type variables are defined. In lines 26–28, property values are declared for datatype property U of individual O. Lines 27–28 cause compile-time errors since the XSD data types *xsd#integer* and *xsd#string* are not subsumed by the range restriction of datatype property U (i.e. *xsd#positiveInteger*).

```
1  import OWL ont = http://www.zhimantic.com/eval;
2  import XML xsd = http://www.w3.org/2001/XMLSchema;
3  namespace N { class Program { static void Main () {
4   #owl#Thing a =                                                    ↪
5    new #owl#Thing("http://www.w3.org/2002/07/owl#A");
6   #owl#Thing o =                                                    ↪
7    new #owl#Thing("http://www.w3.org/2002/07/owl#O");
8   #ont#B b = new #ont#B("http://www.zhimantic.com/eval#B");
9   #ont#C c = new #ont#C("C");
10  a.#ont#R = o;
11  foreach (string T in o.Types) {
12   Console.WriteLine(T);
13  }
14  Console.WriteLine(o + " is" +                                     ↪
15                    (o is #ont#B ? "" : " not") + " a 'B'!");
16  foreach(#ont#B v in #ont#B.Individuals) {
17   Console.WriteLine(v);
18  }
19  a.#ont#R = c;   // Compile-time error in Zhi#!
20  a.#ont#R = b;
21  Console.WriteLine("Individuals 'o' and 'b' are" +                 ↪
22                    (o == b ? "" : " not") + " identical!");
23  #xsd#positiveInteger xpi = 23;
24  #xsd#integer xi = -23;
25  #xsd#string xs = "NaN";
26  o.#ont#U = xpi;
27  o.#ont#U = xi; // Compile-time error in Zhi#!
28  o.#ont#U = xs; // Compile-time error in Zhi#!
29 }}}
```

Java code using the Jena Semantic Web Framework for the same given programming tasks is available online[5]. It can be seen that the Zhi# code shown above treats the OWL terminology as first-class citizens of the program code, and is thus not only inherently less error-prone but can also be checked at compile time. Zhi#'s inherent support for ontologies facilitates type checking on the ontology level, which is completely unavailable if OWL APIs are used.

[5] `http://sourceforge.net/p/zhisharp/wiki/Examples of Use/`

Finally, we mapped the CHIL OWL API [8,16,17] on auxiliary properties and methods of ontology entities in Zhi# programs. The functionality of 50 of the 91 formally specified CHIL OWL API methods could be implemented as Zhi# programming language features, which facilitates the static checking of related method preconditions. Thus, more than half of the possible exceptions that may occur at runtime with API-based access could be eliminated.

The author leaves it to the reader to assess hybrid approaches that propose methodological means of integrating OWL models, which are managed by frameworks such as Protégé and Jena, with computer programs (see Puleston et al. [19] for an OWL-Java combination). Experience shows that integration shortcomings of hybrid approaches can barely be compensated by methodologies, which usually put the burden to behave compliantly to the ontology on the programmer.

4 Related Work

A major disadvantage of using an OWL API compared to, for example, Java-based domain models is the lack of type checking for ontological individuals. This lack of compile-time support has lead to the development of code generation tools such as the Ontology Bean Generator [18] for the Java Agent Development Framework [22], which generates proxy classes in order to represent elements of an ontology. Similarly, Kalyanpur et al. [9] devised an automatic mapping of particular elements of an OWL ontology to Java code. Although carefully engineered the main shortcomings of this implementation are the blown up Java class hierarchy and the lack of a concurrently accessible ontological knowledge base at runtime (i.e. the "knowledge base" is only available in one particular Java virtual machine in the form of instances of automatically generated Java classes). This separation of the ontology definition from the reasoning engine results in a lack of available ABox reasoning (e.g., type inference based on nominals). The two latter problems were circumvented by the RDFReactor approach [25] where a Java API for processing RDF data is automatically generated from an RDF schema. However, RDFReactor only provides a frame-like view of OWL ontologies whereas Zhi# allows for full-fledged ontological reasoning.

In stark contrast to these systems, the Zhi# programming language syntactically integrates OWL classes and properties with the C# programming language using conventional object-oriented notation. Also, Zhi# provides static type checking for atomic XSD data types, which may be the range of OWL datatype properties, while many ontology management systems – not to mention the above approaches – simply discard range restrictions of OWL datatype properties. A combination of static typing and dynamic checking is used for ontological class descriptions. In contrast to static type checking that is based on generated proxy classes, Zhi#'s OWL compiler plug-in adheres to disjoint class descriptions and copes well with multiple inheritance.

Koide and Takeda [11] implemented an OWL reasoner for the $\mathcal{FL}_0$ Description Logic on top of the Common Lisp Object System [5] by means of the Meta-

Object Protocol [10]. Their implementation of the used structural subsumption algorithm [2] is described, however, to yield only incomplete results. The integration of OWL with the Python programming language was suggested by Vrandečić and implemented by Babik and Hluchy [3] who used metaclass-programming to embed OWL class and property descriptions with Python. Their approach, however, offers mainly a syntactic integration in form of LISP-like macros. Also, their prototypical implementation does not support namespaces and open world semantics.

The representation and the type checking of ontological individuals in Zhi# is similar to the type *Dynamic*, which was introduced by Abadi et al. [1]. Values of type *Dynamic* are pairs of a value v and a type tag T, where v has the type denoted by T. The result of evaluating the expression *dynamic e:T* is a pair of a value v and a type tag T, where v is the result of evaluating e. The expression *dynamic e:T* has type *Dynamic* if e has type T. Zhi#'s dynamic type checking of ontological individuals corresponds to the *typecase* construct as proposed by Abadi et al. in order to inspect the type tag of a given *Dynamic*. In Zhi# source programs, the use of OWL class names corresponds to explicit *dynamic* constructs. In compiled Zhi# code, invocations of the *AssertKindOf* method of the Zhi# runtime correspond to explicit *typecase* constructs.

Thatte described a "quasi-static" type system [23], where explicit *dynamic* and *typecase* constructs are replaced by implicit coercions and runtime checks. As in Thatte's work, Zhi#'s dynamic typing for OWL detects errors as early as possible to make it easy to find the programming error that led to the type error. Abadi et al. and Thatte's dynamic types were only embedded with a simple λ-calculus. The same is true for recent *gradual typing* proposals [20]. Tobin-Hochstadt and Felleisen developed the notion of *occurrence typing* and implemented a Typed Scheme [24]. Occurrence typing assigns distinct subtypes of a parameter to distinct occurrences, depending on the control flow of the program. Such distinctions are not made by Zhi#'s OWL compiler plug-in since it is hard to imagine that appropriate subtypes can be computed considering complex OWL class descriptions.

5 Conclusion

The Zhi# programming language makes the property-centric modeling features of the Web Ontology Language available via C#'s object-oriented notation (i.e. normal member access). The power of the "." can be used to declare ad hoc relationships between ontological individuals on a per instance basis. Zhi#'s *OWL aware compilation* integrates value space-based subtyping of XML Schema Definition and ontological classification with features of the programming language such as method overriding, user-defined operators, and runtime type checks. The Zhi# programming language is implemented by an extensible compiler framework, which is tailored to facilitate the integration of external classifier and reasoner components with the type checking of Zhi# programs. The compiler was written in C# 3.0 and integrated with the MSBuild build system for Mi-

crosoft Visual Studio. An Eclipse-based frontend was developed including an editor with syntax highlighting and autocompletion. The complete Zhi# tool suite totals 110 C# KLOC and 35 Java KLOC. Zhi# is available online[6].

Zhi# offers a combination of static typing and dynamic checking for ontological class descriptions. Ontological reasoning directly influences the execution of programs: Zhi# programs don't just execute, they reason. Thus, the development of intelligent applications is facilitated. In contrast to many OWL APIs, Zhi# contains extensive support for XSD data types. Zhi# code that uses elements of an ontology is compiled into conventional C#. All functionality related to the use of ontologies is provided in a "pay-as-you-go" manner. The underlying ontology management system can be substituted in the Zhi# runtime library without recompilation of Zhi# programs.

Future work will include the transformation of Ontology Definition Metamodels [15] into Zhi# programs. With ontological class descriptions being first-class citizens the complete MOF [14] modeling space can be translated into the Zhi# programming language. We further plan to investigate the interplay of closed world semantics in an ontology with autoepistemic features (e.g., the epistemological $\mathcal{K}$-operator) with the static typing in Zhi#.

The Zhi# solution to provide programming language inherent support for ontologies is the first of its kind. Earlier attempts either lack ABox reasoning, concurrent access to a shared ontological knowledge base, or fall short in fully supporting OWL DL's modeling features. In recent years, numerous publications described the – apparently relevant – OWL-OO integration problem. However, the plethora of naïve code generation approaches and contrived hybrid methodologies all turned out to not solve the problem in its entirety. This work demonstrates that OWL DL ontologies can be natively integrated into a general-purpose programming language. The Zhi# compiler infrastructure has shown to be a viable approach to solving the OWL-OO integration problem.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. 13
2. F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, Cambridge, United Kingdom, 2003. 13
3. M. Babik and L. Hluchy. Deep integration of Python with Web Ontology Language. In C. Bizer, S. Auer, and L. Miller, editors, *2nd Workshop on Scripting for the Semantic Web (SFSW)*, volume 183 of *CEUR Workshop Proceedings*, June 2006. 13
4. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004. 2
5. L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *1st European Conference on Object-Oriented Programming (ECOOP)*, pages 151–170, London, UK, 1987. Springer-Verlag. 12

[6] `http://zhisharp.sourceforge.net`

6. A. Hejlsberg, S. Wiltamuth, and P. Golde. C# language specification version 1.0. Technical report, ECMA International, 2002. 3
7. HP Labs. Jena Semantic Web Framework, 2004. 1
8. Information Society Technology integrated project 506909. Computers in the Human Interaction Loop (CHIL), 2004. 12
9. A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic mapping of OWL ontologies into Java. In F. Maurer and G. Ruhe, editors, *16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 98–103, June 2004. 12
10. G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. 13
11. S. Koide and H. Takeda. OWL Full reasoning from an object-oriented perspective. In R. Mizoguchi, Z. Shi, and F. Giunchiglia, editors, *1st Asian Semantic Web Conference (ASWC)*, volume 4185 of *Lecture Notes in Computer Science*, pages 263–277. Springer Verlag, September 2006. 12
12. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), February 2004. 1
13. E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. 2, 6
14. Object Management Group (OMG). MetaObject Facility, August 2005. 14
15. Object Management Group (OMG). Ontology Definition Metamodel, 2005. 14
16. A. Paar. Zhi# – programming language inherent support for ontologies. In J.-M. Favre, D. Gasevic, R. Lämmel, and A. Winter, editors, *ateM '07: Proceedings of the 4th International Workshop on Software Language Engineering*, number 4/2007 in Mainzer Informatik-Berichte, pages 165–181, Mainz, Germany, October 2007. Johannes Gutenberg Universität Mainz. Nashville, TN, USA. 3, 12
17. A. Paar. *Zhi# – Programming Language Inherent Support for Ontologies*. PhD thesis, Universität Karlsruhe (TH), Am Fasanengarten 5, 76137 Karlsruhe, Germany, July 2009. available online: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019039. 3, 9, 10, 12
18. Protégé Wiki. Ontology Bean Generator, 2007. 12
19. C. Puleston, B. Parsia, J. Cunningham, and A. L. Rector. Integrating object-oriented and ontological representations: A case study in Java and OWL. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, editors, *7th International Semantic Web Conference (ISWC)*, volume 5318 of *Lecture Notes in Computer Science*, pages 130–145. Springer Verlag, 2008. 12
20. J. G. Siek and W. Taha. Gradual typing for functional languages. In M. W. Bailey, editor, *7th Workshop on Scheme and Functional Programming (Scheme)*, volume 41 of *ACM SIGPLAN Notices*, New York, NY, USA, September 2006. ACM Press. 13
21. Stanford University School of Medicine. Protégé knowledge acquisition system, 2006. 1
22. Telecom Italia. Java Agent Development Framework (JADE), 2007. 12
23. S. R. Thatte. Quasi-static typing. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 367–381, New York, NY, USA, January 1990. ACM Press. 13
24. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. *ACM SIGPLAN Notices*, 43(1):395–406, January 2008. 13
25. M. Völkel. RDFReactor – from ontologies to programmatic data access. In *1st Jena User Conference (JUC)*. HP Bristol, May 2006. 12