Zhi# - Programming Language Inherent Support for Ontologies

Alexander Paar[1]

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
alexpaar@acm.org

Abstract. XML Schema Definition (XSD) and the Web Ontology Language (OWL) have been widely used to define programming language independent data types and to conceptualize knowledge. However, writing software that operates on XML instance documents and on ontological knowledge bases still suffers from a lack of compile time support for XSD and OWL. In this paper, a novel compiler framework is presented that facilitates the cooperative usage of external type systems with C#. For the resulting programming language Zhi#, XSD and OWL compiler plug-ins were implemented in order to provide static type checking for constrained atomic value types and ontologies. XSD constraining facets and ontological inference rules could be integrated with host language features such as method overwriting. Zhi# programs are compiled to conventional C# and are interoperable with .NET assemblies.

1 Introduction

In recent years, Semantic Web technologies such as RDF(S) [29] [11], DAML+OIL [21], and their common Description Logics [2] based successor OWL DL [30] have paved the way for standardized formal conceptualizations of all kinds of knowledge. However, processing ontological information programmatically is still laborious and error prone. From the author's experience, this is mainly caused by the lack of compile time support both for XML Schema Definition based type definitions, which may be the range of OWL data type properties, as well as for terminological knowledge in form of ontologies.

Ontology management systems merely provide APIs, which have to be used in an explicit manner. The burden to wisely manipulate ontological data is put on the programmer. This problem is even more evident since software systems are usually made up of two distinct class hierarchies, which comprise domain specific classes such as for example `Product`, `Price`, and `Invoice` in a business application and technical classes such as `System.IO.File` or `System.Console`.

In Zhi#, both class hierarchies can be devised using the most appropriate language. C# classes can be used to lay out the technical foundation while OWL can be cooperatively used for the domain specific part of an application.

Processing OWL data implicitly requires the use of possibly constrained XML Schema Definition [15] simple data types, which may be the range of OWL datatype properties. For example, an ontological concept `Person` may have defined a property `Age` of type `xsd#unsignedInt` [1]. This type could be mapped to the C# data type `System.UInt32`. If, however, the XML schema defined a further constrained simple data type such as `#unsignedIntLessThan110` in order to constrain possible `Age` values of a `Person` to reasonable values less than 110, there would be no appropriate C# data type with a value space that comprises integer values between 0 and 110. Instead, assignments to objects of type `System.UInt32` would have to be explicitly checked to be schema valid. An XML instance document - or in this case an instance of a constrained simple data type - is said to be schema valid if there is an XML schema given, and the content of the XML instance document - or of the data type object - conforms to the content model as defined in the schema. Up to now, schema validation has been particularly error prone since there is no isomorphic mapping between atomic XSD datatypes and programmatic data types.

Both the Web Ontology Language and XML Schema Definition stand for external type systems that may be used to define datatypes and to represent knowledge. Such data usually form the foundation of (business-) applications. Still, for application developers it uses to be particularly tedious and error prone to process external type definitions and data structures with one single general purpose programming language.

In this work, a novel compiler framework is presented that facilitates the integration of external type checking and compiler functionality with conventional ECMA standard C#. XSD and OWL type system and compiler components were implemented in order to make constrained atomic XSD data types and ontological concepts first class citizens of the resulting programming language Zhi#. This paper focuses on the following three theses.

Thesis 1 *Using an appropriate compiler framework, domain specific type checking and program transformation functionalities can be cooperatively added to a plain object oriented programming language.*

Thesis 2 *XML Schema Definition constraining facets can be made first class citizens of an object oriented programming language that supports value types.*

Thesis 3 *The integration of constrained data types and ontological reasoning technology with the compiler of a general purpose object oriented programming language facilitates the processing of ontologies and reduces the number of runtime validation errors for constrained atomic value types.*

Thesis 1 has been validated by a working implementation of a Zhi# compiler, which augments the complete safe (i.e. managed) fragment of ECMA 334 standard C# with compile time support both for constrained XML Schema Definition simple data types and Web Ontology Language OWL DL concepts.

[1] In this work, the XML prefixes `xs` and `xsd` are bound to the XML Schema Definition namespace `http://www.w3.org/2001/XMLSchema`

Based on the architectural model of the Zhi# compiler as elucidated in Section 3 XSD and OWL compiler components can be activated to operate separately with standard C# code or to cooperate. External types can be included using the novel keyword `import`, which permits the use of external types in a Zhi# namespace.

```
import XML xsd = http://www.w3.org/2001/XMLSchema;
import XML coke = http://www.choky-cola.com/schema;
import OWL ont = http://www.choky-cola.com/ontology;
```

The Zhi# compiler framework supports the usage of arithmetic (`+`, `-`, `*`, `/`), relational (`>=`, `>`, `==`, `<`, `<=`, `?=`, `?>`, `?<`, `??`, `$=`, `%%`, `%.`), and logical (`&&`, `||`) operators with external types. The operators `?=`, `?>`, `?<`, `??`, `$=`, `%%`, and `%.` were added to the grammar of C# in order to cover XSD constraining facets. Support for additional operators is conceivable but would require a recompilation of the Zhi# compiler framework (not necessarily of its plug-ins). The dot operator '`.`' can be used to access members of external types. Types of different type systems may be used cooperatively in one single statement. For example, a .NET `System.Int32` variable can be assigned the XML value `Age`, which may be defined as a property of the OWL individual `Person`.

```
int i = #ont#Person.Age;
```

Related approaches in the field of programming language support for external languages such as XSD or OWL simply map XSD types or OWL concepts to plain C# (or Java) types. Automatically generated code attempts to mimic the intented behavior (e.g., value space constraints) of mapped types at runtime. In contrast, the Zhi# compiler framework can be extended with relevant type checking and program transformation functionality in order to provide compile time support for external languages and type systems. In particular, a constrained types calculus[2] as elucidated in Section 4 was devised and implemented in order to cope well with XSD constraining facets and to prove Thesis 2. Finally, Thesis 3 will be validated by a case study that will demonstrate the practicability and the ease of use of the novel Zhi# programming language features.

2 Related Work

With major software companies that have come out strongly in favor of standardizing on XML Schema Definition, several approaches have emerged to define programming languages specifically for the XML domain. XDuce [22] is a functional programming language that is specifically designed for processing XML data. One can read an XML document as an XDuce value, extract information

[2] In this work, the term "constrained types" refers to atomic data types that represent a value space, which may be lessened (i.e. constrained) by explicitly defined constraining facets (e.g., `xsd:minExclusive`, `xsd:maxExclusive`). This is different to constrained based type inference algorithms found in the literature where constraints are not checked but rather recorded for later consideration.

from it or convert it to another format, and write out the result value as an XML document. The subsequent Xtatic project [18] aims to develop theoretical foundations and implementation techniques for a lightweight extension of C# tailored for native XML processing. In contrast to Zhi#, both approaches focus on content models of aggregated XML structures and lack support for XSD atomic types. In this way, they are similar to Xen and Cω, which are amalgamations of Microsoft's Common Language Runtime (CLR) [1], XML, and SQL programming languages. The JWIG development system [13] is a Java-based high-level language for the development of interactive Web services. JWig integrates the central features of the `<bigwig>` language [9] into Java by providing explicit support for Web service sessions and safe XHTML dynamic document construction. In particular, JWIG facilitates the construction of XHTML documents by introducing XML templates, which can contain inlined pieces of code, called code gaps. At runtime, code gaps can be substituted by other templates or literals. In the Xact project [26], JWig's validation algorithm is extended to implement further compile time guarantees, such that, dynamically transformed XML documents are valid according to a given XML schema. XL [17] adopts to XQuery [8] and combines imperative and declarative programming language features to facilitate the development of Web services. The XML Objects Programming Language [38] integrates XML and XPath [14] into the Java programming language with a main emphasis on valid updates for persistent XML objects.

In [3] a constraint algebra is proposed in which complex constraint expressions can be built up from primitive constraints using logical connectives like conjunction or disjunction. This algebra, however, has not been embedded with a type system. A constraint model for XML is presented in [16]. Just like the XML domain specific languages mentioned above, this model focuses on the content model of aggregated types and does not include value space constraints of atomic data types as they may be defined by XML Schema Definition.

To the best of the author's knowledge, all of these approaches lack support for constrained XSD atomic data types. Rather, the Zhi# approach presented in this paper is complementary to some technologies that do only support content models of XSD complex types.

While there have been no approaches to devise programming language inherent support for the Web Ontology Language, the problem to provide compile time support for OWL DL stems from practice. In the CHIL research project [24], which aims to introduce computers into a loop of humans interacting with humans, rather than condemning a human to operate in a loop of computers, a semantic middleware has been developed that fusions information provided by so called *perceptual components* in meaningful ways. Each perceptual component (e.g., image and speech recognizers, body trackers, etc.) contributes to the common domain of discourse. The Web Ontology Language OWL was decidedly used to replace previous domain models that had been based on particular programming languages. A major disadvantage of using an OWL API compared to previously used Java based domain models had been the lack of compile time support (i.e. static type checking). This lack of compile time support has lead to the development of code generation tools such as the Ontology Bean Generator [37] for the Java Agent Development Framework [40], which generates proxy classes in order to represent elements of an ontology. Similarly, in [25] Kalyanpur et al. 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 in [41] where a Java API for processing RDF data is automatically generated from an RDF schema. While this may still result in a significant number of automatically generated Java classes, the generated API operates on one single external RDF model, which may be handled by existing ontology management systems. Still, since RDFReactor operates on RDF triples, obsolete generated code may result in meaningless or even invalid modifications of the triple store.

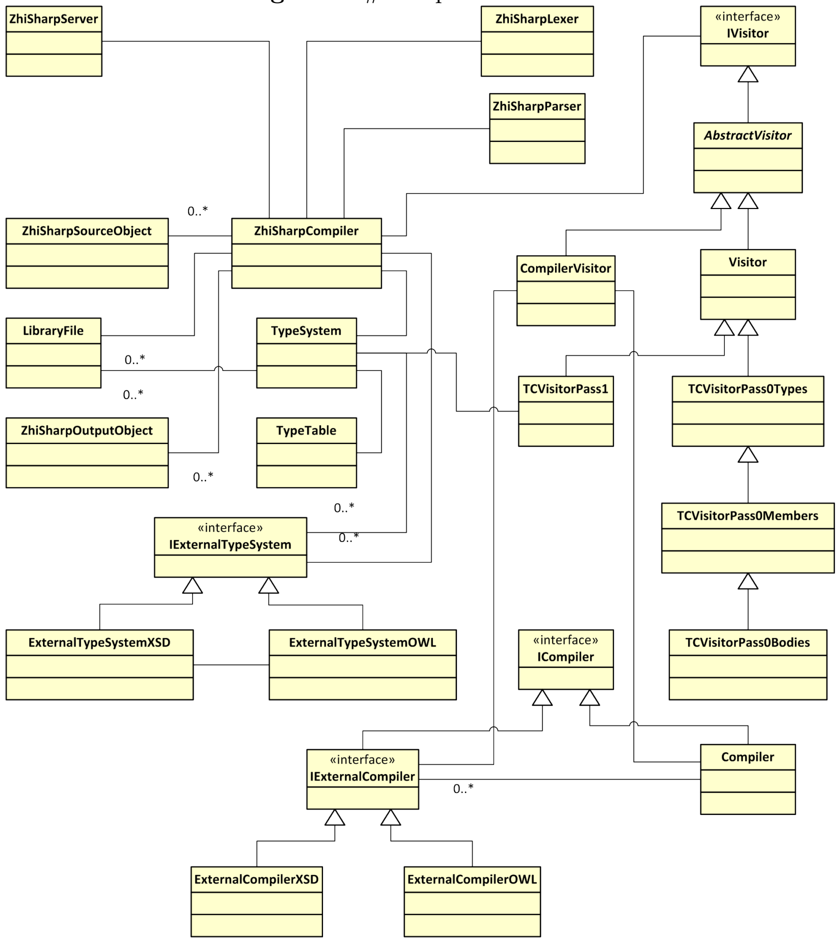
The Zhi# compiler framework as introduced in Section 3 is complementary to a long line of approaches to add *syntactic* extensibility to programming languages [10,27,42,12,4,6]. Most of these approaches support *syntactic safety*. Embedded code is checked at compile time to be syntactically correct. In [10] embedded code is used to compose XML documents. Proper program transformations and a properly defined underlying XML API guarantee that compositions (i.e. generated XML instance documents) are syntactically correct as well. However, nothing can be said about the validity of generated XML documents with respect to a given schema. This lack of type safety is inherent to most of the referenced approaches. In [4], argument and result types of embedded code are checked but there is no error trailing (i.e. type checking errors in the expanded code are not traced back to the unexpanded syntax).

In Zhi#, a host language (i.e. C#) can be augmented with external type systems (e.g., OWL, XSD). In contrast to a syntactic language extension, these type systems are fully integrated with the static type checking of the host language. Objects of such external type systems can be addressed following an object oriented notation. In particular, external objects are assumed to be organizable into taxonomies. Such a hierarchical organization provides for the reuse of methods and data that are located higher in the hierarchy. Objects are either atomic value types, whose value spaces may be constrained by means of constraining facets, or complex types, which can be seen as collections of named attributes. The author trusts that these two notations plus operator overloading plus some minor syntactical additions for importing external namespaces and referencing external types are sufficient for a variety of applications. In fact, Bravenboer and Visser state in [10] that in object oriented languages "language constructs are often sufficient for domain abstractions at the semantic level".

3 The Zhi# Compiler Framework

Fig. 1 depicts the architectural model of the Zhi# compiler framework on a class level. The ZhiSharpCompiler class aggregates all components of the framework. Zhi# source code and references to included .NET assemblies are passed into the ZhiSharpCompiler as ZhiSharpSourceObject and LibraryFile objects.

Fig. 1. Zhi# compiler framework



Zhi# source code is tokenized an transformed into an AST by the ZhiSharp-Lexer and ZhiSharpParser, which were automatically generated based on an ANTLR [36] grammar of the augmented C# language specification [19].

A TypeTable, which comprises type definitions of passed in Zhi# source code as well as referenced .NET assemblies, is build during the first two type checking passes by the TCVisitorPass0Types and TCVisitorPass0Members visitors.

Type names in method and property bodies are resolved by the `TCVisitor-Pass0Bodies` visitor. Next, the `TCVisitorPass1` visitor performs the actual type checking of the Zhi# code, which includes the application of type inference mechanisms of activated `IExternalTypeSystem`s. Note that the presence of implementations of such external type systems (e.g., `ExternalTypeSystem-XSD`) is transparent to the `TCVisitorPass1` visitor since only the generic (i.e. type system independent) interfaces of the `TypeSystem` and `TypeTable` classes are used. Also, external type systems may cooperate in order to resolve for example member access expressions where the member of an external type is defined in another external type system.

Finally, successfully type checked Zhi# code is transformed into conventional C# code by the `CompilerVisitor`. Again, a type system independent `Compiler` is used to make activated implementations of the `IExternalCompiler` interface such as `ExternalCompilerXSD` transparent to the Zhi# compiler framework.

Cooperation between external type systems is achieved through delegating type checking and compilation tasks to the external compiler component responsible for the external type of the left operand of a binary expression or to the external compiler component responsible for the external type of the right operand of a binary expression if the left operand is a .NET type.

```
#shippingAndHandling SaH = 7.5;
System.Decimal d = #ont#TVSet.hasPrice + SaH;
```

In the above listing, the evaluation of the dot operator in the expression `#ont#TVSet.hasPrice` is dispatched to the OWL DL compiler component. This component can process OWL DL types and is also aware of the the XSD compiler component. First, the RDF type of the individual `#ont#TVSet` is checked to allow for a datatype property `#ont#hasPrice`. Accordingly, the expression type of `#ont#TVSet.hasPrice` shall be evaluated to `#xsd#decimal{> 0}{%. 2}`[3] (a positive real number that has at most two fraction digits). The type of variable `SaH` shall be evaluated to `#xsd#decimal{>= 0}{%. 1}` (a non-negative real number that has at most one fraction digit). Next, the type of the binary expression `#ont#TVSet.hasPrice + SaH` is evaluated by the XSD compiler component to `#xsd#decimal{> 0}{%. 2}`. The variable declaration statement `System.Decimal d = #ont#TVSet.hasPrice + SaH` is type checked by the XSD component, too.

Analogously, each binary expression in the above code snippet is compiled by the Zhi# compiler component that is responsible for the respective type system. The preceding listing would be compiled to the following conventional C# code (for the sake of brevity, only the alias "ont" is used in the compiled code instead of the fully qualifying namespace).

```
RTSimpleType SaH =
    ZhiSharpRuntime.GetInitializedRTSimpleType(
      "#shippingAndHandling",
```

[3] In Zhi#, type names can be followed by a number of constraint notations of the form {constraint operator *literal*}.

```
      ZhiSharpRuntime.ToString("#shippingAndHandling", 7.5)
    );
System.Decimal d = Convert.ToDecimal(
    ZhiSharpRuntime.Addition("#xsd#decimal{> 0}{%. 2}",
      ZhiSharpRuntime.GetDatatypePropertyValue(
        ZhiSharpRuntime.GetIndividual("#ont#TVSet"),
        "#ont#hasPrice"
      ),
      SaH
    )
  );
```

Note that the above code uses static functions defined in a partial static class `ZhiSharpRuntime`. This class, which is needed at runtime, is the only part of the Zhi# compiler framework that may have to be recompiled when compiler components are added or removed. Also, all external type system functionality is used in a "pay as you go" manner. In Zhi#, there is no overhead for conventional C# code. The class `RTSimpleType` is explained in the next Section, which elucidates the constrained types calculus implemented by the Zhi# XSD compiler component.

4 XSD Compile Time Support

4.1 The λ_C Constrained Types Calculus

This subsection elucidates (in an incomplete manner) an extension of the typed lambda calculus with subtyping ($\lambda_{<:}$) for constrained atomic (or simple) data types. For the sake of brevity, the elucidation of XSD fundamental facets (see [7]) is not given in this paper. Also, only some λ_C-specific extensions of the conventional $\lambda_{<:}$-calculus are presented.

Atomic data types can only have atomic values, which are not allowed to be further fractionalized even though this may be technically possible (e.g., the XSD datatype `xsd#string` may be considered to be an atomic data type despite the fact that it comprises several distinguishable character information items). Special emphasis was placed on the ability to make it possible to cover constrained based type derivations as they are allowed for XML Schema Definition atomic data types.

Analogously to [7], atomic datatypes are derived from a number of unconstrained primitive data types $P_1, ..., P_n \in \mathcal{P}$. In particular, built-in XML Schema Definition data types such as `xsd#duration`, `xsd#dateTime`, and `xsd#decimal` are valid elements of $\mathcal{P}$ and will be denoted $P_{xsd\#duration}$, $P_{xsd\#dateTime}$, and $P_{xsd\#decimal}$, respectively.

A primitive data type P is a three-tuple consisting of a set of distinct values, called its value space $\upsilon(P)$ (e.g., the value space $\upsilon(P_{xsd\#boolean})$ is the set {true, false} to denote a logical true and a logical false), a set of lexical representations called its lexical space, and a set of fundamental facets that characterize properties of the value.

The value spaces of primitive XML Schema Definition data types $\mathcal{P}_{xsd}$ such as $P_{xsd\#duration}$, $P_{xsd\#dateTime}$, and $P_{xsd\#decimal}$ are given by the respective type definitions of XML Schema Definition built-in data types in [7].

The value space of a base type T may be pruned by the application of one or more constraints $c_i = \phi(TV)b_i\ ^{i \in 1..n}$. Each constraint c has a type variable TV, which is to be bound to a base type T, and a body b that possibly lessens the value space $\upsilon(TV)$ of TV. The letter ϕ is used as a binder for the base type parameter TV. A constraint $c = \phi(TV)b$ has a value space $\upsilon(c)$ comprising all elements of the value space $\upsilon(TV)$, which satisfy the comparison operations defined in the constraint body b. A constraint body $b = \{x | x \in \upsilon(TV)\} \bigcap_{k \in 1..m} \{x | x \prec literal_k\}$ defines the intersection of the value space of TV and those values which adhere to the comparison operations $x \prec literal_k$. Depending on the constraining facet '$\prec$', $literal_k$ is interpreted as a lexical representation of an element of $\upsilon(TV)$ or of another value space that is implicitly effective for the constraining facet (e.g., if TV is bound to the XSD data type $P_{xsd\#gYear}$ $literal_k$ is interpreted as a lexical representation of a gregorian calendar year while it is taken as an integer number for the constraining facets $c_{xsd:length}$ and $c_{xsd:totalDigits}$).

In order to capture the constraining facets as defined in the W3C Recommendation for XML Schema Definition [7] the operators given in Table 1 may be substituted for the operator placeholder '$\prec$'.

Table 1. Constraining facets

Zhi# comparison operator	**XSD constraining facet**
? =	$c_{xsd:length}$, $c_{?=}$
? >	$c_{xsd:minLength}$, $c_{?>}$
? <	$c_{xsd:maxLength}$, $c_{?<}$
??	$c_{xsd:pattern}$, $c_{??}$
\$ =	$c_{xsd:enumeration}$, $c_{\$=}$
<=	$c_{xsd:maxInclusive}$, $c_{<=}$
<	$c_{xsd:maxExclusive}$, $c_{<}$
>	$c_{xsd:minExclusive}$, $c_{>}$
>=	$c_{xsd:minInclusive}$, $c_{>=}$
%%	$c_{xsd:totalDigits}$, $c_{\%\%}$
%.	$c_{xsd:fractionDigits}$, $c_{\%.}$

In the λ_C-calculus, atomic types are inductively defined by their value spaces. Atomic types are derived through the application of value space constraints according to rule TD-CSTRAPP.

$$\frac{T' = T.c}{\upsilon(T') = \upsilon(c\{\{TV \leftarrow T\}\})} \text{ (TD-CSTRAPP)}$$

A main characteristic of object oriented languages is that an object can emulate another object that has fewer methods, since the former supports the entire protocol of the latter. Analogously, for constrained data types as defined in this work, the basic rules of subtyping are affective: reflexivity, transitivity, and subsumption. Additionally, width and depth subtyping rules apply to atomic data types that are constrained by one or more constraining facets.

A type A is considered to be a subtype of B if the value space of A is a subset of the value space of B (rule S-VSPACE). In particular, A and B must be derived from the same base type since otherwise their value spaces would be disjoint. The rule S-VSPACE subsumes the width and depth subtyping rules S-WIDTH and S-DEPTH.

$$\frac{\upsilon(A) \subseteq \upsilon(B)}{A <: B} \text{ (S-VSPACE)}$$

$$\frac{\upsilon(A) = \bigcap_{i \in 1..n+k}^{TV} c_i \quad \upsilon(B) = \bigcap_{i \in 1..n}^{TV} c_i}{A <: B} \text{ (S-WIDTH)}$$

$$\frac{\text{for each } i \;\; c_i <:: d_i \quad \upsilon(A) = \bigcap_{i \in 1..n}^{TV} c_i \quad \upsilon(B) = \bigcap_{i \in 1..n}^{TV} d_i}{A <: B} \text{ (S-DEPTH)}$$

A type A is considered to be a subtype of B if both types are derived from the same base type and fewer constraints are defined for type B than for A. The intuition that it is safe to add constraints to an atomic type is captured by the *width subtyping* rule S-WIDTH for constrained atomic types.

Also, constraints that are defined for atomic types may vary as long as the value spaces of each corresponding constraint are in the subset relation (i.e. the constraints are in the sub-constraint relation). The *depth subtyping* rule S-DEPTH for constrained atomic types expresses this notion.

Finally, rule S-APP captures the notion that a constraint application always makes a type more specific (i.e. constraint applications can only reduce the value space of a type).

$$\frac{A <: B}{A.c <: B} \text{ (S-APP)}$$

This form of type construction mimics the semantics of XML Schema Definition. In the λ_C-calculus it is also possible to infer transient constraints that may hold for the instances of constrained types within only a limited scope of a program.

A scope within a program shall be denoted by $\diamond$. Considering the *then* branch of an if-statement $if\ (a \prec literal)\ then \diamond$ it is possible to add the constraint $c_{\prec literal}$ to the type of variable a. The constraint $c_{\prec literal}$ holds for the instance a within scope $\diamond$ until a is assigned a value (i.e. in a programming language with side effects a must also not be referenced by method invocations). These two intuitions of adding constraints to the type of a variable for a limited scope and eventually removing them upon an assignment to the variable are captured by the rules TI-IFADD and TI-ASSIGNREM, where Γ and $\Gamma_\diamond$ shall be a typing context and a transient typing context for a limited scope $\diamond$, respectively (i.e., $\Gamma_{\diamond\diamond}$ shall be a sub-scope of $\Gamma_\diamond$).

$$\frac{\Gamma \vdash a : A \qquad if\ (\bigwedge_{i \in 1..n} (a \prec_i literal_i))\ then\ \diamond}{\Gamma_\diamond \vdash a : A' \qquad \Gamma_\diamond \vdash A' = \bigcap_{i \in 1..n}^{A} (c_i = \phi(TV)\{x | x \in \upsilon(TV)\} \cap \{x | x \prec_i literal_i\})}$$

(TI-IFADD)

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash a : A \qquad \Gamma_\diamond \vdash a : A' \qquad \Gamma_\diamond \vdash A' = \bigcap_{i \in 1..n}^{A} c_i \qquad a := t}{\Gamma_{\diamond\diamond} \vdash a : A}$$

(TI-ASSIGNREM)

The following example may illustrate the effect of the TE-IFADD rule on the type checking of λ_C-programs. Under the assumption $\vdash a : A_{xsd\#int}$ the λ_C-application $(\lambda x : Age.x)(a)$ would fail to typecheck according to rule S-VSPACE since $\upsilon(A_{xsd\#int}) \nsubseteq \upsilon(Age)$. The application would, however, typecheck if it occurred within the *then*-branch of an *if*-statement such as $if\ ((a >= 0) \wedge (a < 110))\ then\ (\lambda x : Age.x)(a)$ since the transient type $A'_{xsd\#int}$ of a could be inferred to be $\bigcap^{A_{xsd\#int}} c_{>=0} \cap c_{<110}$. Using the rule S-VSPACE one can conclude that $A'_{xsd\#int}$ is a valid subtype of Age.

4.2 XSD Aware Compilation

The λ_C-type system was fully implemented in C# as an XSD plug-in for the Zhi# compiler framework; a Java based implementation has also been embedded with the CHIL Knowledge Base Server [34], which is an adapter for OWL DL ontology management systems.

The XSD plug-in can be used in the Zhi# compiler framework to enable *XSD aware compilation*. XSD atomic data types defined in XML schemas are made public as native data types in Zhi# programs using the novel keyword `import`, which works analogously for XML data types like the C# `using` keyword for .NET programming language type definitions. It permits the use of XSD atomic data types in a Zhi# namespace, such that one does not have to qualify the use of a type in that namespace.

```
<xsd:schema xmlns:xsd="..."
    targetNamespace="http://www.choky-cola.com">
 <xsd:simpleType name="productID">
  <xsd:restriction base="xsd:int">
   <xsd:minInclusive value="4000"/>
   <xsd:maxExclusive value="8000"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

```
10 import XML coke = http://www.choky-cola.com;
20 namespace MyBusinessApplication {
30   class MyClass {
40     #coke#productID pID;
50   }
60 }
```

Thus, it is possible to use a `productID` data type, which shall be defined in a namespace `http://www.choky-cola.com`, as shown in the code snippet above. In the given example, the value space of the XSD type `productID` is limited to integer values greater or equal to 4000 and less than 8000. Accordingly, the following assignment in line 30 would result in error messages at compile time. The same kinds of errors would occur if XSD values are assigned to incompatible .NET variables. In contrast to other approaches, the type incompatibility is traced to the actual value space constraints.

```
10 int i = ...;
20 #coke#productID pID;
30 pID = i;
```

This error trailing makes it particularly easy for developers to handle assignments to or from constraint datatypes. In the following listing, the value space of the source operand is inferred based on control and data flow analysis. Using the λ_C-calculus rules TI-IFADD and TI-ASSIGNREM the value space of variable `i` is inferred to be a subset of the value space of `pID` within the scope of the `if`-statement. Several novel comparisons operators as listed in Table 1 were introduced in the Zhi# programming language in order to make it particularly easy to test for XSD specific constraining facets. Additionally, the value of the `Length` property defined for the .NET value type `System.String` can be used to check for compliance with the XSD constraining facets `xsd:length`, `xsd:minLength`, and `xsd:maxLength`.

```
30 if ((i >= 4000) && (i < 8000)) {
40   pID = i;
50 }
```

Zhi#'s XSD component implements the relational operators `>=`, `>`, `==`, `<`, `<=`, `?=`, `?>`, `?<`, `??`, `$=`, `%%`, and `%.`, and the arithmetic operators `+`, `-`, `*`, and `/`. Relational operators can be used to constrain the possible value space of atomic objects for limited scopes in a program as shown in the listing above. Operator overloading can be used to enable combinations and comparisons of external XSD objcets with .NET reference types.

Types of binary expressions where objects are combined using arithmetic operators are inferred at compile time. For objects `a`, `b`, and `c` of type `xsd#non-NegativeInteger`, the assignment `a = b + c` would successfully typecheck while `a = b - c` results in compile time errors since the type of the binary expression `b - c` would be inferred to be `xsd#decimal{%. 0}` (i.e. positive and negative real numbers with no fraction digits).

5 OWL Compile Time Support

5.1 A Formally Specified OWL API

In recent years, the Web Ontology Language OWL has been widely adopted as the lingua franca of the Semantic Web. However, there are no widely accepted standards yet that define APIs to manage ontological data. Processing ontological information still suffers from the heterogeneity imposed by the plethora of available ontology management systems and OWL APIs [23,32,28,39,33,31,5]. The author devised a Floyd-Hoare logic based formal specification of an OWL API in order to make it possible to consistently adapt off-the-shelf ontology management systems. The methods of this CHIL [24] OWL API are specified as Hoare triples [20] of the following form.

$$\begin{array}{rc}
\{P\}\text{:} & \{\eta(\mathcal{O}) \wedge (\bigwedge \mathcal{O} \vdash ((\mathcal{SHOIN}(\mathbf{D})\ \textit{semantics}) \dot{\vee}\ \texttt{Exception}))\} \\
Q\text{:} & \mathcal{O}, \mathtt{m}(p_1, ..., p_n) \\
\{R\}\text{:} & \{(\bigwedge \mathcal{O} \vdash (\mathcal{SHOIN}(\mathbf{D})\ \textit{semantics})) \wedge (\textit{return value semantics})\}
\end{array}$$

Fig. 2. OWL API Hoare triple schema

Preconditions guarantee the ontology to be consistent and particular axioms and facts to hold; the program terminates with the given exceptions otherwise. The comma operator applies a method on the ontology. If the program terminates, particular axioms and facts and return value semantics can be asserted. The CHIL OWL API comprises 33 methods for telling the TBox, 8 methods for telling the ABox, 19 methods for asking the TBox, and 9 methods for asking the ABox of OWL DL knowledge bases. It was fully implemented in an adapter software referred to as the CHIL Knowledge Base Server, which can expose the functionality of off-the-shelf ontology management systems via an XML-over-TCP remoting protocol. Client libraries are available for several programming languages, among those .NET.

5.2 OWL Aware Compilation

Based on the formally specified OWL API an OWL plug-in for the Zhi# compiler framework was developed that enables the usage of OWL concepts, roles, and individuals in Zhi# programs. In particular, the OWL plug-in can be used cooperatively with the XSD plug-in in order to facilitate the handling of OWL datatype properties. The following ontology snippet is an excerpt from the larger CHIL ontology that covers the domain of multi-model applications in human interaction scenarios.

$$\begin{array}{l}
\textit{Room} \sqsubseteq \top, \textit{ActivityLevel} \sqsubseteq \top, \textit{Meeting} \sqsubseteq \textit{Occurrence} \sqsubseteq \top, \\
\geq 1 \textit{hasActivityLevel} \sqsubseteq \textit{Meeting}, \top \sqsubseteq \forall \textit{hasActivityLevel}.\textit{ActivityLevel}, \\
\geq 1 \textit{scheduledAt} \sqsubseteq \textit{Occurrence}, \top \sqsubseteq \forall \textit{scheduledAt}.\textit{Room}, \textit{hosts} \sqsubseteq \textit{scheduledAt}^{-}, \\
\textit{ActivityLevel}(\textit{low}), \textit{ActivityLevel}(\textit{high}), \\
\textit{ActiveMeeting} \sqsubseteq \textit{Meeting} \sqcap \exists \textit{hasActivityLevel}.\textit{high}
\end{array}$$

In Zhi#, it is possible to *schedule* a *review meeting* in *room 248* as shown in lines 20 - 30 in the code snippet below.

```
10 import OWL ont = http://chil.server.de/ontology;
15 class C { static void Main() {
20   #ont#Room r = new #ont#Room("#ont#room248");
25   #ont#Meeting m = new #ont#Meeting("#ont#reviewMeeting");
30   m.#ont#scheduledAt = r;
35   foreach (#ont#Occurrence o in r.#ont#hosts) {
40     System.Console.WriteLine(r + " hosts " + o + "!");
45   }
50   m.#ont#hasActivityLevel = #ont#high;
55   if (m is #ont#ActiveMeeting) {
60     System.Console.WriteLine(m + " is an active meeting!");
65   }
70 }}
```

The query in line 35 benefits from ontological reasoning in two ways. First, the *review meeting* is classified not only as a *meeting* but also as an *occurrence*. Second, *room 248 hosts* the *review meeting* since the role *hosts* was declared as the inverse of *scheduledAt*.

In line 55 the *review meeting* is at runtime inferred to be an *active meeting* based on the complex concept definition of an *active meeting* in the ontology and based on the value of its property *hasActivityLevel*.

The given examples may illustrate the usefulness to have in a program the same convenient and compact notations used to describe data in an ontology.

5.3 OWL and XSD

Recently, several approaches have emerged to refine on the integration of external type systems such as XSD and the Web Ontology Language OWL. In [35], Pan and Horrocks present an extension of OWL DL, called OWL-Eu, which allows for class descriptions based on (customized) datatypes. Especially, customized datatypes can be used in datatype exists restrictions ($\exists T.u$) and datatype value restrictions ($\forall T.u$). It can be imagined to use the Zhi# framework with an OWL-Eu API since Zhi#'s XSD component already provides type checking and type inference capabilities that are needed to cope well with OWL-Eu's datatype expressions. Zhi#'s OWL component would have to be extended in order to track local range restrictions that can be inferred for properties of individuals that are of a particular type. Given complex concept descriptions the OWL type checking component could augment the type information of ontological roles accordingly.

6 Conclusion and Outlook

Based on ECMA standard C# a compiler framework was devised that facilitates the cooperative usage of external type systems in the host language. XML

Schema Definition and Web Ontology Language OWL DL plug-ins were developed in order to provide a significant degree of compile time support for constrained atomic datatypes and ontologies. A complete tool suite was implemented including a Zhi# Code DOM and an Eclipse project type with program visualizers and an editor with syntax highlighting and auto-completion.

It could be shown that it is possible to integrate external type systems with an object oriented host language. The notion of atomic value types was extended with constraint based type derivation, which was fully integrated with host language features such as method overwriting. Type inference is used to infer the most specific constraints that hold for an atomic value type within a limited scope of a program. Ontological inference rules are used for static type checking. It is conceivable to add further type systems (e.g., SQL) with the Zhi# compiler framework.

Currently, a controlled experiment is planned to find out empirically whether the Zhi# programming language facilitates the processing of XML and OWL data. In course of this study, several CHIL demo applications [34] will be rewritten in Zhi# in order to evaluate the gained improvements of the proposed approach in terms of standard software metrics (i.e. productivity, error density).

The current state of ontological reasoning technology may not yet be sufficient to manage very large ontological knowledge bases. However, significant performance gains have been achieved in recent years. These developments are likely to continue since the advantages of the convenient and compact OWL notation have been widely acknowledged. As a consequence, it appears to be reasonable to make this important form of ontologies available in a widely used programming language.

In the long run, the author plans to investigate the interplay of closed world semantics of an ontology with the type checking provided in the Zhi# programming language in order to enable an even higher degree of static type checking for ontologies.

7 Acknowledgements

This work is part of the FP6 CHIL project (FP6-506909) [24], partially funded by the European Commission under the Information Society Technology (IST) program. The author acknowledges valuable help and contributions from all partners of the project.

References

1. Common Language Infrastructure (CLI). Technical report, ECMA International, June 2006. Internet: http://www.ecma-international.org/publications/standards/Ecma-335.htm.
2. F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. F. Patel-Schneider. *The Desciption Logic Handbook*. Cambridge University Press, 2003.

3. F. Bacchus and T. Walsh. A constraint algebra. In *1st International Workshop on Constraint Propagation and Implementation*, 2004.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.
5. S. Bechhofer. The DIG Description Logic Interface: DIG/1.0. Technical report, University of Manchester, Oxford Road, Manchester M13 9PLA, 2002.
6. A. Begel and S. L. Graham. Language analysis and tools for ambiguous input streams. *Electronic Notes in Theoretical Computer Science*, 110:75–96, 2004.
7. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004. Internet: http://www.w3.org/TR/xmlschema-2/.
8. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium (W3C), June 2006. Internet: http://www.w3.org/TR/xquery/.
9. C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.
10. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.
11. D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/rdf-schema/.
12. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, 1994.
13. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.
14. J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium (W3C), November 1999. Internet: http://www.w3.org/TR/xpath.
15. D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004. Internet: http://www.w3.org/TR/xmlschema-0/.
16. W. Fan, G. M. Kuper, and J. Simeon. A unified constraint model for XML. In *World Wide Web Conference Series*, 2001.
17. D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for Web service specification and composition. *Computer Networks*, 42(5):641–660, 2003.
18. V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 151–175. Springer, 2003.
19. A. Hejlsberg, S. Wiltamuth, and P. Golde. C# Language Specification. Technical report, ECMA International, June 2006. Internet: http://www.ecma-international.org/publications/standards/Ecma-334.htm.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.
21. I. Horrocks, F. van Harmelen, and P. Patel-Schneider. DAML+OIL. Technical report, DARPA's Information Exploitation Office and Euro-

pean Union's Information Society Technologies, March 2001. Internet: http://www.daml.org/2001/03/daml+oil-index.html.
22. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
23. HP Labs. Jena Semantic Web Framework, 2004. Internet: http://www.hpl.hp.com/semweb/jena.htm.
24. Information Society Technology integrated project 506909. Computers in the human interaction loop, 2004. Internet: http://chil.server.de.
25. 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, *SEKE*, pages 98–103, 2004.
26. C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, 2004.
27. B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.
28. J. Lee, R. Goodwin, R. Akkiraju, Y. Ye, and P. Doshi. Ibm ontology management system, 2003. Internet: http://www.alphaworks.ibm.com/tech/snobase/.
29. F. Manola and E. Miller. RDF Primer. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/rdf-primer/.
30. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/owl-features/.
31. R. Möller and V. Haarslev. RACER: Renamed ABox and Concept Expression Reasoner, 2004. Internet: http://www.sts.tuharburg.de/ r.f.moeller/racer/.
32. B. Motik. KAON2, 2006. Internet: http://kaon2.semanticweb.org.
33. Open RDF. Sesame RDF Database, 2006. Internet: http://www.openrdf.org.
34. A. Paar, J. Reuter, J. Soldatos, K. Stamatis, and L. Polymenakos. A formally specified ontology management api as a registry for ubiquitous computing systems. *The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies (Applied Intelligence)*, 2007. to appear.
35. J. Z. Pan and I. Horrocks. Owl-eu: Adding customised datatypes into owl. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(1):29–39, January 2006.
36. T. Parr. Another tool for language recognition, 2005. Internet: http://www.antlr.org.
37. Protégé Wiki. Ontology bean generator, 2007. Internet: http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator.
38. H. Schuhart and V. Linnemann. Valid updates for persistent XML objects. In *Datenbanksysteme für Business, Technologie und Web*, volume 65 of *Lecture Notes in Informatics*, pages 245–264. Gesellschaft für Informatik, 2005.
39. Stanford University School of Medicine. Protégé knowledge acquisition system, 2006. Internet: http://protege.stanford.edu.
40. Telecom Italia. Java agent development framework, 2007. Internet: http://jade.cselt.it/.
41. M. Völkel. Rdfreactor – from ontologies to programatic data access. In *Proc. of the Jena User Conference 2006*. HP Bristol, May 2006.
42. D. Weise and R. F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.