

Static Typing with Value Space-based Subtyping

Alexander Paar
Department of Computer Science
University of Pretoria
Pretoria 0002, South Africa
alexpaar@acm.org

Stefan Gruner
Department of Computer Science
University of Pretoria
Pretoria 0002, South Africa
sgruner@cs.up.ac.za

ABSTRACT

Numerous programming and schema languages contain the notion of value types. However, support for value space-based subtyping is spotty. This paper presents a formal type system for atomic value types as an extension of the simply typed lambda calculus with subtyping. In the λ_C -calculus, value types can be derived through the application of value space constraints. Type inference rules can be used to infer transient value space constraints that hold for limited scopes of a program. The type system of the λ_C -calculus is proved to be sound. The λ_C -calculus was fully implemented in Java and C#. The presented approach was successfully validated to subsume XML Schema Definition type construction and to facilitate the integration of XSD data types with the C# programming language.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*data types and structures, frameworks*; D.3.3 [Programming Languages]: Processors—*compilers*

General Terms

Static typing of value types with constraints

Keywords

Value types, constraints, static typing, type inference

1. INTRODUCTION

Many widely used programming languages such as, for example, Java and C# implement *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. For pragmatic reasons, programming languages make a distinction between class types, user-defined value types, and a limited set of built-in atomic value types (e.g., data types

int, *short*). In contrast to class types, in Java and C#, value types cannot be used for type derivation. Instead, subtype relationships between built-in data types are provided extra by the programming language compiler to take into account obvious subsumption relationships between the data types' value spaces. For example, a Java or C# *short* value can be used in all places where an *int* is admissible.

The XML Schema Definition (XSD) [1] 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. Value spaces can be defined through the application of value space *constraints*. For example, an XSD data type *age* can be derived from the built-in data type *xsd#nonNegativeInteger* through the application of the constraint *less than 110*. XML Schema Definition introduces 19 built-in data types and 11 constraining facets that can be used for type construction.

Because of their limited set of built-in data types Java and C# only provide a limited set of isomorphic mappings from XSD data types to programmatic data types. In general, compilers for programming languages such as Java or C# are unaware of the value space-based subtyping that is used by XML Schema Definition. This impedance mismatch has been widely acknowledged in the literature [12, 20, 10] and in practice leads to brittle workarounds such as the generation of programming language proxy classes for XSD data types or the use of assertions to enforce XSD value space constraints within program code.

The contribution of this paper is a formal type system for constrained¹ atomic value types as an extension of the simply typed lambda calculus with subtyping (λ_C).

The purpose of the λ_C -calculus is manifold. First, it can be used to formalize the W3C Recommendation for XSD data types, which up to now only provides natural language specifications of fundamental type system properties such as type derivation and subsumption. Second, implementations of the λ_C -calculus can be used for XML schema validation. Third, the λ_C -calculus provides the formal foundation for the integration of value space-based type construction, type derivation, and type inference into general purpose programming languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Name Date and Location.

Copyright Year ACM CRDATA ...\$10.00.

¹Note that, in this work, the term "constrained types" refers to atomic data types that represent a value space, which may be constrained by explicitly defined constraining facets (e.g., *xsd:minExclusive*, *xsd:maxExclusive*). This is different to constraint-based type inference algorithms found in the literature where constraints are not checked but rather recorded for later consideration.

2. THE λ_C -CALCULUS

The type system of the λ_C -calculus provides a formalization of the construction, derivation, and inference of atomic data types. 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 data type *xsd#string* is considered an atomic data type despite the fact that it comprises several distinguishable characters). Table 1 lists the notations that are used in this work. In particular, variables S , T , and U stand for atomic types; superscripted variables S^\rightarrow , T^\rightarrow , and U^\rightarrow are used in all places where both atomic as well as function types are permissible. Throughout this paper, XML Schema Definition will be used to illustrate the practicality of the λ_C -calculus.

We begin with the introduction of unconstrained primitive base types $P_1, \dots, P_n \in \mathcal{P}$. Built-in XSD data types such as *xsd#duration*, *xsd#dateTime*, and *xsd#decimal* are valid elements of \mathcal{P} and will be denoted $P_{\text{xsd}\#\text{duration}}$, $P_{\text{xsd}\#\text{dateTime}}$, and $P_{\text{xsd}\#\text{decimal}}$, respectively.

A primitive base type P is a triple consisting of a set of distinct values denoted $v(P)$, called its *value space* (e.g., the value space $v(P_{\text{xsd}\#\text{boolean}})$ is the set $\{\text{true}, \text{false}\}$ 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 space (e.g., cardinalities, order relations). Primitive base types can only have atomic values. Function $v(P)$ is defined semantically for each primitive base type under consideration. Each value in the value space $v(P)$ of a data type P may be denoted by more than one literal of its lexical space (e.g., “1” and “1.0” are both valid lexical representations of the same $P_{\text{xsd}\#\text{float}}$ value 1).

2.1 Facets

2.1.1 Fundamental Facets

A *fundamental facet* is an abstract property that semantically characterizes the values in a value space. The λ_C -calculus includes the fundamental facets *equality*, *partial/total order*, *cardinality*, *boundedness*, *numeric*, and *date-time*.

Every value space supports the notion of *equality*, with the following rules. For any a and b in the value space, either a is equal to b , denoted $a = b$, or a is not equal to b , denoted $a \neq b$. There is no pair a and b from the value space such that both $a = b$ and $a \neq b$. For all a in the value space, $a = a$. For any a and b in the value space, $a = b$ if and only if $b = a$. For any a , b , and c in the value space, if $a = b$ and $b = c$, then $a = c$. For any a and b in the value space, if $a = b$, then a and b cannot be distinguished (i.e. equality is identity). In spite of the fact that value spaces of particular built-in data types (e.g., $P_{\text{xsd}\#\text{float}}$ and $P_{\text{xsd}\#\text{double}}$) may intuitively appear to be compatible, they will be treated as pairwise disjoint for primitive base types $P_i, P_j \in \mathcal{P}$ where $i \neq j$ and $i, j \in 1..n$ (i.e. P_i and P_j do not share any values).

Value spaces may either be *partially ordered* (i.e. the order relation is reflexive, transitive, and antisymmetric) or *totally ordered* (i.e. the order relation is transitive, antisymmetric, and total). A data type is said to be partially ordered or totally ordered if there exists a partial order-relation or a total order-relation defined for its value space, respectively. Data types whose value spaces have upper and lower bounds defined are said to be *bounded*; *unbounded* if no such bounds

exist. Value spaces may be finite or countably infinite. A data type is said to have the *cardinality* of its value space. The cardinality of a value space is the number of elements in the value space. A data type is said to be *numeric* if its values are conceptually quantities in some mathematical number system. A data type is said to be a *date-time* data type if its values denote dates and times as in the ISO 8601 standard of date and time-related data exchange [8].

2.1.2 Constraining Facets

In the λ_C -calculus, atomic data types are computational structures, where the only operations are constraint applications. A *constraining facet* is an optional property that can be applied to an atomic data type to constrain its value space. A value space $v(T)$ is the set of values for a given atomic data type T .

The value space of a base type T can be restricted by the application of one or more constraints $c_i = \phi(TV)b_i$, $i \in 1..n$. Each constraint c_i has a type variable TV , which is to be bound to a base type T , and a body b that possibly constrains the value space $v(T)$. The letter ϕ is used as a binder for the base type parameter TV . A constraint $c = \phi(TV)b$ where the type variable TV is bound to a base type T (denoted by $c\{\{TV \leftarrow T\}\}$) has a value space $v(c\{\{TV \leftarrow T\}\})$ comprising all elements of the value space of the base type T that satisfy the comparison operations defined in the constraint body b . A constraint body $b = \{x|x \in v(TV)\} \bigcap_{k \in 1..m} \{x|x \prec \text{literal}_k\}$ defines the intersection of the value space of TV and those values that satisfy the properties $x \prec \text{literal}_k$, $k \in 1..m$.

Depending on the constraining facet ‘ \prec ’, *literal_k* is interpreted as a lexical representation of an element of $v(TV)$ or of another value space that is implicitly effective for the constraining facet. For example, if TV is bound to the XSD data type $P_{\text{xsd}\#\text{gYear}}$, *literal_k* is interpreted as a lexical representation of a Gregorian calendar year while it is taken as an integer number for the constraining facet $c_{\text{xsd}\#\text{length}}$, which defines the length of string data types. The comparison operators given in Table 2 can be substituted for the operator placeholder ‘ \prec ’ in constraint bodies to capture the constraining facets as defined in the W3C Recommendation for XML Schema Definition.

Table 2: Comparison operators for XSD

Comparison operator	XSD constraining facet
?=	$c_{\text{xsd}\#\text{length}} (c_{=?})$
?>	$c_{\text{xsd}\#\text{minLength}} (c_{>?>})$
?<	$c_{\text{xsd}\#\text{maxLength}} (c_{>?<})$
??	$c_{\text{xsd}\#\text{pattern}} (c_{??})$
=\$	$c_{\text{xsd}\#\text{enumeration}} (c_{\$=})$
<=	$c_{\text{xsd}\#\text{maxInclusive}} (c_{<=})$
<	$c_{\text{xsd}\#\text{maxExclusive}} (c_{<})$
>	$c_{\text{xsd}\#\text{minExclusive}} (c_{>})$
>=	$c_{\text{xsd}\#\text{minInclusive}} (c_{>=})$
%%	$c_{\text{xsd}\#\text{totalDigits}} (c_{\% \%})$
%.	$c_{\text{xsd}\#\text{fractionDigits}} (c_{\% .})$

Constraints where both the comparison operator and the (single) literal of the constraint body are given are denoted by a c subscripted with the operator symbol or the name of the constraining facet followed by the literal (e.g., $c_{[< 5]}$ and $c_{[\text{xsd}\#\text{maxExclusive} 5]}$ denote the exclusive upper bound ‘five’).

The λ_C -calculus comprises formal subsumption rules not

Table 1: Terminology of the λ_C -calculus

$P_1, \dots, P_n \in \mathcal{P}$	primitive base types (e.g., $P_{\text{xsd}\#\text{boolean}}, P_{\text{xsd}\#\text{string}}$)
S, T, U	atomic types
$S \rightarrow, T \rightarrow, U \rightarrow$	atomic or function types
$v(T)$	value space of type T
$c = \phi(TV)b$	constraint c with base type parameter TV and body b
$b = \{x x \in v(TV)\} \bigcap_{k \in 1..m} \{x x \prec \text{literal}_k\}$	constraint body b with base type parameter TV
$T.c$	constraint application
$\bigcap_{i \in 1..n}^T c_i \equiv T.c_1 \dots .c_n$	multiple constraint application
$c\{\{TV \leftarrow T\}\}$	type variable binding in constraint c
$v(c\{\{TV \leftarrow T\}\})$	value space of bound constraint c
\mapsto	substitution
$<$	subtype of
$<::$	subconstraint of

only for the actual data types but also for the constraints that are used for type construction. Thus, both data types and constraints are first-class citizens in the calculus, which facilitates value space-based subtyping as will be shown below. A constraint $c_1 = \phi(TV_1)b_1$ is a sub-constraint of a constraint $c_2 = \phi(TV_2)b_2$ if the base type parameters TV_1 and TV_2 are bound to the same base type T and the value space $v(c_1\{\{TV_1 \leftarrow T\}\})$ is subsumed by $v(c_2\{\{TV_2 \leftarrow T\}\})$ as shown in Table 3. Variable T is implicitly all-quantified for all types on which both c_1 and c_2 can be applied. Note that type/constraint compatibility can only be defined semantically. Hence, in the rest of the paper, for all applications of constraints on types user-defined compatibility is assumed. For example, Section 4.1.5 in part 2 of the W3C recommendation for XML Schema Definition [1] defines the set of applicable constraints on XSD data types.

Table 3: S-CstrVSpace

$$\frac{v(c_1\{\{TV \leftarrow T\}\}) \subseteq v(c_2\{\{TV \leftarrow T\}\})}{c_1 <:: c_2}$$

Rule S-CSTRVSPACE subsumes the subconstraint axiom S-CSTRWIDTH and the subconstraint rule S-CSTRDEPTH. The *width subconstraint* axiom S-CSTRWIDTH (see Table 4) captures the intuition that one wants to consider the constraint $c_1 = \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x|x \prec \text{literal}_i\}$ with $n+k$ predicates defined to be a subconstraint of less restrictive $c_2 = \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec \text{literal}_i\}$.

Axiom S-CSTRWIDTH applies only to constraints where the common properties $x \prec \text{literal}_i$ $i \in 1..n$ are identical and fewer properties are defined in the body of constraint c_2 than in the body of c_1 . It is also safe to allow the comparison operands literal_i $i \in 1..n$ to vary as long as the value spaces of each corresponding comparison operation are in the subset relation. The *depth subconstraint* rule S-CSTRDEPTH as shown in Table 5 captures this notion.

As shown below, a more restrictive constraint $\text{succ}(c)$ can be computed by replacing the literal l in the body of constraint c by its successor $\text{succ}(l)$ or predecessor $\text{pred}(l)$ depending on the used comparison operator and bound base type such that, the resulting value space of the constrained type becomes more specific. Note that here the variable l stands for literal values while the variable c denotes a con-

straint (i.e. the definition of $\text{succ}(c)$ is not circular).

$$c = \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec l\}$$

$$c_{\text{pred}} \equiv \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec \text{pred}(l)\}$$

$$c_{\text{succ}} \equiv \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec \text{succ}(l)\}$$

$$\text{succ}(c) = \begin{cases} c_{\text{pred}} & \text{iff } v(c_{\text{pred}}\{\{TV \leftarrow T\}\}) \subset v(c\{\{TV \leftarrow T\}\}) \\ c_{\text{succ}} & \text{iff } v(c_{\text{succ}}\{\{TV \leftarrow T\}\}) \subset v(c\{\{TV \leftarrow T\}\}) \end{cases}$$

The successor $\text{succ}(c)$ of a constraint c is undefined if there are no constraints c_{pred} and c_{succ} whose values spaces – when applied on a base type T – are subsumed by $v(c\{\{TV \leftarrow T\}\})$. In this case, it is not possible to derive further subtypes using constraint c . As a result, the constraint’s host type to which the base type parameter TV is bound is *final over constraint* c (e.g., a constraint that defines the exact length of strings cannot be used to derive subtypes).

2.2 Type Derivation

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 written $T.c$ for an atomic type T and a constraint c . The notation $\bigcap_{i \in 1..n}^T c_i$ is a shorthand for the subsequent application of multiple constraints $c_1 \dots c_n$ on type T (i.e. $T.c_1 \dots .c_n$). The order in which constraints are applied has no significance and can be permuted by implementations for optimization purposes.

Constraint applications on atomic types can be reduced to set operations on their value spaces. Let $T_n \equiv \bigcap_{i \in 1..n}^{T_0} c_i$ be an atomic type, which shall be derived from a given base type T_0 based on a number of constraints c_1, \dots, c_n . The effective value space $v(T_n)$ of the derived type T_n can be computed by reducing the sequence of constraint applications $T_0.c_1 \dots .c_n$ to the successive application of constraints c_k on type T_{k-1} for $k = 1, \dots, n$ such that, $T_k = T_{k-1}.c_k$. Accordingly, $v(T_k) = v(c_k\{\{TV \leftarrow T_{k-1}\}\})$.

A value space expression $v(c_k\{\{TV \leftarrow T_{k-2}.c_{k-1}\}\})$ may be rewritten as $v(c_k\{\{TV \leftarrow T_{k-2}\}\}) \cap v(c_{k-1}\{\{TV \leftarrow T_{k-2}\}\})$ as long as type definitions within a derivation chain are not subject to change. If in a derivation tree a type definition T is substituted by a more specific type defini-

Table 4: S-CstrWidth

$$\phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x|x \prec y_i\} <:: \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\}$$

Table 5: S-CstrDepth

$$\frac{\text{for each } i \quad \{x|x \prec y_i\} \subseteq \{x|x \prec z_i\}}{\phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\} <:: \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec z_i\}}$$

tion T' (denoted by $T \mapsto T'$), all dependent type definitions change accordingly as defined by rule TD-SUBS (note that T and T' must be derived from the same primitive base type). For example, if in a derivation tree $T_3 <: T_2 <: T_1$, where $T_2 \equiv T_1.c_{11}$ and $T_3 \equiv T_2.c_2$, $T_2 \mapsto T_2.c_{12}$, then $T_3 \mapsto T_3.c_{12}$.

Table 6: Td-Subs

$$\frac{T \mapsto T'}{T.c \mapsto T'.c}$$

2.3 Subtyping

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 effective: 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 S is considered to be a subtype of T (denoted by $S <: T$) if the value space of S is a subset of the value space of T as shown in Table 7. In particular, S and T must be derived from the same primitive base type since otherwise their value spaces would be disjoint. The non-algorithmic rule S-VSPACE subsumes the width and depth subtyping rules S-WIDTH and S-DEPTH.

A type S is considered to be a subtype of U if both types, S and U , are derived from the same base type T through the application of constraints and fewer constraints are defined for type U than for S . 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 as shown in Table 8.

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 denoted by $c <:: d$). The *depth subtyping* rule S-DEPTH for constrained atomic types as shown in Table 9 expresses this notion.

The subtyping rule S-APP captures the notion that a constraint application always makes a type more specific, which is required for the soundness proof of the λ_C -type system.

Table 7: S-VSpace

$$\frac{v(S) \subseteq v(T)}{S <: T}$$

Table 8: S-Width

$$\frac{S = \bigcap_{i \in 1..n+k}^T c_i \quad U = \bigcap_{i \in 1..n}^T c_i}{S <: U}$$

Table 9: S-Depth

$$\frac{\text{for each } i \quad c_i <:: d_i \quad S = \bigcap_{i \in 1..n}^T c_i \quad U = \bigcap_{i \in 1..n}^T d_i}{S <: U}$$

In the λ_C -calculus, the subtyping rule S-ARROW (see section *Subtyping* in Table 14) for function types is the standard one with covariant return types and contravariant function parameters (i.e. a function type is contravariant in its domain and covariant in its range). For constrained atomic data types that are embedded with a conventional object-oriented programming language, the subtyping rule for function types could follow the same scheme. However, while covariant return types and contravariant formal parameter types are generally safe, different programming languages support different variance policies. Although the calculus presented in this work uses a specific implementation of the subtyping rule for function types, different behaviors of host programming languages may be adopted.

2.4 Properties of the λ_C -Type System

The most fundamental property of type systems is *safety* (also called *soundness*). The soundness of a type system can be shown by proving the *progress* and *preservation* theorems, which say that a well-typed term is either a value or it can take a step of evaluation and if a well-typed term takes a step of evaluation then the resulting term is also well-typed, respectively. In [19], proofs of the *progress* and *preservation* theorems for the simply typed lambda-calculus with subtyping ($\lambda_{<}$) are given. Both proofs are based on straightforward induction on a derivation of $t : T$. Both properties are preserved in the λ_C -calculus because the type construction mechanisms do not pertain to the original $\lambda_{<}$ -typing and evaluation rules. What remains is to prove the subsumption property for constrained types.

Table 10: S-App

$$\frac{S <: T}{S.c <: T}$$

Intuitively, $S <: T$ can be read “every value described by S is also described by T ”. According to rule S-TRANS (see section *Subtyping* in Table 14), type derivations must only yield types whose value spaces are more specific than those of their base types. The proof of the following lemma shows that this subsumption property is always satisfied for type derivations in the λ_C -calculus.

Lemma 2.1 (Subsumption) *If $S' <: S$ and $S <: T$ then $S' <: T$.*

Proof: There is almost nothing to show since in the λ_C -calculus atomic types can only be derived through constraint applications.

Case $S' = S.c$, where $c = \phi(TV)\{x|x \in v(TV)\} \cap \{\dots\}$. For a type derivation $S' = S.c$, where $c = \phi(TV)\{x|x \in v(TV)\} \cap \{\dots\}$ is an arbitrary constraint, the value space $v(S') = v(c\{\{TV \leftarrow S\}\})$ reduces to $\{x|x \in v(S)\} \cap \{\dots\}$. Consequently, all elements of $v(S')$ are also elements of $v(S)$ and the subtyping rule S-VSPACE can be applied to conclude that $S' <: T$. \square

2.5 Type Inference

In the purely functional λ_C -calculus, atomic types can be derived through explicit applications of value space constraints. This form of type construction mimics the semantics of XML Schema Definition. This section indicates how type inference of constrained λ_C -data types can be accomplished in an imperative programming language.

The λ_C -calculus given in Table 14 is purely functional. In particular, variables are immutable and there are no assignments, which are commonly considered harmful in the functional programming community. On the other hand, assignments add a great deal of convenience to a programming language by allowing for the implementation of mutable, implicit, distributed state. Several approaches in the literature [16] therefore propose extensions of functional programming languages with constructs and reduction rules that represent mutable variables and assignments. The type construction mechanism of the λ_C -type system facilitates the inference of transient value space constraints that hold for mutable variables within only a limited scope of an imperative program.

We introduce the typing rule T-REL, which provides for the use of relational expressions with *if*-statements. Relational expressions relate variables with expressions that denote elements of the value space of the primitive base type of the variable type.

Table 11: T-Rel

$$\frac{a : A \quad A <: P \quad v_e \in v(P)}{a < e : Bool}$$

Next we define the notion of a program *scope*. A scope within a program is denoted by \diamond , a sub-scope of \diamond is denoted by $\diamond\diamond$. A program scope may, for example, be the *then*-branch of an *if*-statement; a sub-scope of this scope may be the remainder of the *then*-branch after a variable assignment.

Considering the *then*-branch of an *if*-statement of the form *if* ($a < e$) *then* \diamond it is safe to add the constraint $c_{[< e]}$ to the type of variable a . The constraint $c_{[< e]}$ 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 as shown in Tables 12 and 13, where Γ and Γ_\diamond are a typing context and a transient typing context for a limited scope \diamond , respectively (i.e. Γ_∞ is the typing context of the sub-scope $\diamond\diamond$).

Table 12: Ti-IfAdd

$$\frac{\Gamma \vdash a : A \quad \text{if} \left(\bigwedge_{i \in 1..n} (a <_i e_i) \right) \text{ then } \diamond}{\Gamma_\diamond \vdash a : \bigcap_{i \in 1..n} c_i \quad ; c_i = \{x|x \in v(A)\} \cap \{x|x <_i v_{e_i}\}^{i \in 1..n}}$$

Table 13: Ti-AssignRem

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A \quad \Gamma_\diamond \vdash a : \bigcap_{i \in 1..n} c_i \quad a := t}{\Gamma_\infty \vdash a : A}$$

The following example illustrates the effect of the TI-IFADD rule on the type checking of programs written in a prospective extension of the λ_C -calculus with imperative constructs. Under the assumptions $\vdash x : Age$ and $\vdash a : P_{\text{xsd}\#\text{nonNegativeInteger}}$, where the value space of type *Age* includes integer numbers from 0 to 110, the following assignment will fail to type-check according to rule S-VSPACE since $v(P_{\text{xsd}\#\text{nonNegativeInteger}}) \not\subseteq v(Age)$. The assignment does, however, type-check if it occurs within the *then*-branch of an *if*-statement as shown below.

```
x := a
```

```
if (a <= 110) then x := a endif
```

For the scope of the *then*-branch the type of a can be inferred to be $P_{\text{xsd}\#\text{nonNegativeInteger}.c_{[\leq 110]}}$. Using the subtyping rule S-VSPACE one can conclude that the inferred type $P_{\text{xsd}\#\text{nonNegativeInteger}.c_{[\leq 110]}}$ is a subtype of *Age*.

In practice, implementations of the λ_C -type system such as the one of the Zhi# programming language (see Section 3) may consider only ANDed top-level relational expressions in *if*-statements for type inference. In this way, the type inference algorithm is rather conservative (i.e. incomplete), which is likely to be sufficient in practice since the effects of type inference should still be comprehensible for the programmer. For fundamental reasons, the proposed kind of type inference will always be incomplete since otherwise the inference algorithm would solve the Halting Problem (for the same reason, there is no “optimal” compiler).

3. VALIDATION

The λ_C -calculus was fully implemented in Java and C#. The extensible architectural model of this implementation, which is elucidated in detail in [17], allows for the implementation of arbitrary type systems that are based on the notion of value space-based subtyping and constraint-based type derivation (e.g., XSD, OCL, SQL).

In particular, the Java and C# λ_C -frameworks were used to implement the XML Schema Definition type system. The λ_C -based XSD processors facilitate reading and writing of XSD data type definitions, XSD data types can be constructed in-memory using an object-oriented API, and data values can be validated to conform to the content model of given XSD data type definitions.

3.1 The CHIL Knowledge Base Server

The Java implementation of the λ_C -based XSD processor was employed in the CHIL Knowledge Base Server in order to augment its API for the Web Ontology Language (OWL) [11] with extensive support for XSD data types, which are the range of OWL datatype properties. The CHIL OWL API was successfully used in the CHIL research project [7] to adapt the Jena Semantic Web Framework [6] and to compensate for Jena’s limited support for XSD data types [18]. The CHIL Knowledge Base Server is available online².

3.2 The Zhi# Programming Language

The C# implementation of the λ_C -based XSD processor was used for the XSD plug-in for the Zhi# programming language [17]. Zhi# is a proper superset of C# 1.0 and is extensible with respect to external type systems. The Zhi# compiler framework³ provides two extension points for semantic analysis and program transformation that can be implemented by compiler plug-ins to make external type definitions available in Zhi# program code. Thus, external type definitions can be seamlessly used with features of the C# programming language such as, for example, method overriding and user defined operators. In addition, compiler plug-ins can contribute type inference for atomic data types.

3.2.1 Implementation of XSD data types

The following code snippet illustrates the use of XSD data types in Zhi# program code. In line 1, XSD built-in data types are imported. In line 5, λ_C -based validation is used by the Zhi# compiler (i.e. the XSD plug-in) to reject the assignment of a signed 32-bit integer value to an *xsd#positiveInteger* variable. In line 6, using rule T1-IFADD, the value space of variable *i* is inferred to be an integer number *greater than or equal to one* and *less than 2^{31}* , where the former constraint only holds within the limited scope of the *if*-statement and rule T1-ASSIGNREM is used to remove this transient constraint in line 8. Type checks can be deferred to runtime by the use of downcasts as shown in line 9.

```

1 import XML xsd = http://www.w3.org/2001/XMLSchema;
2 class Program {
3     public static void Main() {
4         int i = new Random().Next(); //  $-2^{31} \leq i < 2^{31}$ 
5         #xsd#positiveInteger pi = i; // Error!
6         if (i >= 1) {
7             pi = i; //  $1 \leq i < 2^{31}$ 
8         }
9         #xsd#positiveInteger pi = //  $-2^{31} \leq i < 2^{31}$ 
10         (#xsd#positiveInteger) i; // Warning...
11     }
12 }
```

In [17], the superiority of an integration of XSD data types into a programming language by means of the λ_C -calculus is shown by contradistinction with the four mapping options

²<http://chilkbs.sourceforge.net>

³<http://zhisharp.sourceforge.net>

for atomic XSD data types as proposed by Lämmel and Meijer [10]. Static typing of XSD data types provably reduces the number of possible runtime validation errors.

3.2.2 Implementation of OCL invariants

Although developed in the first place to facilitate the handling of XSD data types, the usefulness of the λ_C -type system is not limited to XML Schema Definition. This section outlines the implementation of OCL invariants with λ_C -data types in a Zhi# program.

The Object Constraint Language (OCL) [13] has been included with the specification of the Unified Modeling Language (UML) [14, 15] since UML version 1.1. OCL provides means for textually specifying constraints, which apply to model elements, that cannot otherwise be expressed by the diagrammatic notations of UML. A very frequent constraint-/model element combination is the use of OCL invariants to tag UML class attributes. Invariants are described using an expression that evaluates to true if the invariant is met. For example, given a class *Person*, the following invariant definitions restrict the age of a person to values *greater than or equal to zero* and *less than or equal to 110*; also, persons can only have names that are not longer than 40 characters.

```

context Person
inv age >= 0 && age <= 110
inv name.size() <= 40
```

There is a plethora of approaches that implement OCL invariants by means of runtime checks that are, for example, facilitated by means of aspect-oriented programming (AOP) [9] based tools and techniques. In contrast, in the Zhi# programming language, OCL invariants can be directly translated into λ_C -data types. Thus, OCL invariants can be substituted by context-free type definitions and runtime invariant checks can be substituted by compile time analysis. Also, no additional tool suite (e.g., AOP compiler) is entailed by the Zhi# solution.

4. RELATED WORK

Jeffrey S. Foster et al. [4] developed the CQUAL tool that can be used to extend standard types with flow-sensitive type qualifiers. The formal foundation of CQUAL is constituted by the type system of the call-by-value lambda calculus that was extended with pointers and type qualifier annotations. Type inference checks that given annotations are correct, where CQUAL’s flow sensitivity is restricted to the decorating type qualifiers. In the λ_C -calculus, types are explicitly constructed using value space constraints, which may be modified entirely by also flow-sensitive type inference. CQUAL’s type qualifiers are rather complementary to the actual type information in order to, for example, annotate objects with required state information (e.g., to describe an *open File*) and there are only ad hoc user-defined subtype relations between type qualifiers while there are formal subtyping rules for value space constraints in the λ_C -calculus.

Brian Chin et al. introduced semantic type qualifiers [2] to support user-defined type refinements, which ensure additional invariants of interest (e.g., *nonnull*, *nonzero*). Just like in the λ_C -calculus, value-qualified types are always considered to be subtypes of their associated unqualified types (cf. the width subtyping rules in the λ_C -calculus). Still, in contrast to the λ_C -type system there is no support for explicit subtype declarations between user-defined qualifiers.

An important difference is that in the work of Brian Chin et al. associated type rules of type qualifiers are defined for particular code patterns (i.e. knowledge is assumed about the syntax of the programming language), which is not required for constrained type definitions in Zhi#. On the other hand, Brian Chin et al. provide an automatic soundness checker to prove the declared invariants for type qualifier definitions. In their subsequent CLARITY approach [3], qualifier rules can be automatically inferred from given invariants. Both CQUAL and CLARITY were instantiated in frameworks for user-defined type qualifiers in C programs.

The JQUAL tool [5] adds user-defined type qualifiers to Java. JQUAL supports subtyping orders of type qualifiers, which, however, must be supplied manually.

5. CONCLUSION & OUTLOOK

The author devised an extension of the simply typed lambda calculus with subtyping ($\lambda_{<}$) for constrained atomic data types. The λ_C -calculus provides for static typing of constrained atomic data types. Types are derived through the application of value space constraints. A soundness proof of the λ_C -calculus was obtained. The λ_C -calculus was fully implemented in Java and C# for the XML Schema Definition type system. λ_C -based XSD processors were used to augment an API for the Web Ontology Language with support for XSD data types and to integrate XSD data types into the C# programming language. The λ_C -calculus has shown to be a viable solution to making value space-based subtyping and constraint-based type derivation available for API and programming language-centric use alike.

Current work includes the integration of type inference with the λ_C -calculus based on an extension of the calculus with assignments and mutable variables.

The λ_C -calculus is not limited to XSD data types but can be used for the implementation of any type system where types are inductively defined through their value spaces. Currently, a tool is being developed that generates λ_C -type definitions from OCL invariant expressions. Thus, OCL invariants are translated into Zhi# program code that can be checked at compile time as opposed to runtime.

6. REFERENCES

- [1] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004.
- [2] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–95, New York, NY, USA, June 2005. ACM Press.
- [3] B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In P. Sestoft, editor, *15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 264–278, Berlin / Heidelberg, Germany, March 2006. Springer Verlag.
- [4] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *ACM SIGPLAN Notices*, pages 1–12, New York, NY, USA, June 2002. ACM Press.
- [5] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 42 of *ACM SIGPLAN Notices*, pages 321–336, New York, NY, USA, October 2007. ACM Press.
- [6] HP Labs. Jena Semantic Web Framework, 2004.
- [7] Information Society Technology integrated project 506909. Computers in the Human Interaction Loop (CHIL), 2004.
- [8] International Organization for Standardization. Representation of dates and times (ISO 8601). Technical report, International Organization for Standardization, December 2004.
- [9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, June 1997.
- [10] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch (changing lead into gold). In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, page 80. Springer-Verlag, June 2007.
- [11] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), February 2004.
- [12] E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL)*. Forschungszentrum Jülich GmbH, 2003.
- [13] Object Management Group. Object Constraint Language, version 2.0. Technical report, Object Management Group, May 2006.
- [14] Object Management Group. Unified Modeling Language, infrastructure. Technical report, Object Management Group, February 2009.
- [15] Object Management Group. Unified Modeling Language, superstructure. Technical report, Object Management Group, February 2009.
- [16] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 43–56, New York, NY, USA, 1993. ACM Press.
- [17] A. Paar. *Zhi# – Programming Language Inherent Support for Ontologies*. PhD thesis, Universität Karlsruhe (TH), Am Fasanengarten 5, 76137 Karlsruhe, Germany, July 2009.
- [18] A. Paar, J. Reuter, J. Soldatos, K. Stamatis, and L. Polymenakos. A formally specified ontology management API as a registry for ubiquitous computing systems. *Applied Intelligence*, 30(1):37–46, February 2009.
- [19] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, 2002.
- [20] D. Thomas. The impedance imperative: Tuples + objects + infosets = too much stuff! *Journal of Object Technology*, 2(5):7–12, September-October 2003.

APPENDIX

A. THE λ_C -CALCULUS

Table 14: The λ_C -Calculus (extends the $\lambda_{<}$ -calculus)

Syntax	
$t ::=$ x $\lambda x : T^{\rightarrow}.t$ tt	<i>terms:</i> <i>variable</i> <i>abstraction</i> <i>application</i>
$v ::=$ $\lambda x : T^{\rightarrow}.t$ true false	<i>values:</i> <i>abstraction value</i> <i>true value</i> <i>false value</i>
$T ::=$ $T^{\rightarrow} \rightarrow T^{\rightarrow}$ Top Bool $P_{\text{xsd}\#\text{boolean}}, \dots, P_{\text{xsd}\#\text{string}}$ $T.c$	<i>types:</i> <i>type of functions</i> <i>maximum type</i> <i>type of booleans</i> <i>primitive base types</i> <i>constraint application</i>
$c ::=$ $c = \phi(TV)\{x x \in v(TV)\} \bigcap_{k \in 1..m} \{x x < \mathit{literal}_k\}$	<i>constraints:</i> <i>constraint</i>
$\Gamma ::=$ \emptyset $\Gamma, x : T^{\rightarrow}$	<i>contexts:</i> <i>empty context</i> <i>term variable binding</i>
Evaluation	$t \rightarrow t'$
$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$	(E-APP1)
$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$	(E-APP2)
$(\lambda x : T_{11}^{\rightarrow}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$	(E-APPABS)
$\mathit{if\ true\ then\ } t_2 \mathit{\ else\ } t_3 \rightarrow t_2$	(E-IFTRUE)
$\mathit{if\ false\ then\ } t_2 \mathit{\ else\ } t_3 \rightarrow t_3$	(E-IFFALSE)
$\frac{t_1 \rightarrow t'_1}{\mathit{if\ } t_1 \mathit{\ then\ } t_2 \mathit{\ else\ } t_3 \rightarrow \mathit{if\ } t'_1 \mathit{\ then\ } t_2 \mathit{\ else\ } t_3}$	(E-IF)
Typing	$\Gamma \vdash t : T^{\rightarrow}$
$\frac{x : T^{\rightarrow} \in \Gamma}{\Gamma \vdash x : T^{\rightarrow}}$	(T-VAR)
$\frac{\Gamma, x : T_1^{\rightarrow} \vdash t_2 : T_2^{\rightarrow}}{\Gamma \vdash \lambda x : T_1^{\rightarrow}.t_2 : T_1^{\rightarrow} \rightarrow T_2^{\rightarrow}}$	(T-ABS)
	over, please

Table 14: The λ_C -Calculus (extends the $\lambda_{<}$ -calculus)

$\frac{\Gamma \vdash t_1 : T_{11}^{\rightarrow} \rightarrow T_{12}^{\rightarrow} \quad \Gamma \vdash t_2 : T_{11}^{\rightarrow}}{\Gamma \vdash t_1 t_2 : T_{12}^{\rightarrow}}$	(T-APP)
$\frac{\Gamma \vdash t : S^{\rightarrow} \quad S^{\rightarrow} <: T^{\rightarrow}}{\Gamma \vdash t : T^{\rightarrow}}$	(T-SUB)
$\mathit{true} : \mathit{Bool}$	(T-TRUE)
$\mathit{false} : \mathit{Bool}$	(T-FALSE)
$\frac{t_1 : \mathit{Bool} \quad t_2 : T^{\rightarrow} \quad t_3 : T^{\rightarrow}}{\mathit{if } t_1 \mathit{ then } t_2 \mathit{ else } t_3 : T^{\rightarrow}}$	(T-IF)
Subtyping	$S^{\rightarrow} <: T^{\rightarrow}$
$S^{\rightarrow} <: S^{\rightarrow}$	(S-REFL)
$\frac{S^{\rightarrow} <: U^{\rightarrow} \quad U^{\rightarrow} <: T^{\rightarrow}}{S^{\rightarrow} <: T^{\rightarrow}}$	(S-TRANS)
$S^{\rightarrow} <: \mathit{Top}$	(S-TOP)
$\frac{T_1^{\rightarrow} <: S_1^{\rightarrow} \quad S_2^{\rightarrow} <: T_2^{\rightarrow}}{S_1^{\rightarrow} \rightarrow S_2^{\rightarrow} <: T_1^{\rightarrow} \rightarrow T_2^{\rightarrow}}$	(S-ARROW)
$\frac{v(S) \subseteq v(T)}{S <: T}$	(S-VSPACE)
$\frac{S = \bigcap_{i \in 1..n+k}^T c_i \quad U = \bigcap_{i \in 1..n}^T c_i}{S <: U}$	(S-WIDTH)
$\frac{\text{for each } i \ c_i <:: d_i \quad S = \bigcap_{i \in 1..n}^T c_i \quad U = \bigcap_{i \in 1..n}^T d_i}{S <: U}$	(S-DEPTH)
$\frac{S <: T}{S.c <: T}$	(S-APP)
$\frac{v(c_1 \{\{TV \leftarrow T\}\}) \subseteq v(c_2 \{\{TV \leftarrow T\}\})}{c_1 <:: c_2}$	(S-CSTRVSPACE)
$\frac{\phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x x \prec y_i\} <:: \phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x \prec y_i\}}{\text{for each } i \ \{x x \prec y_i\} \subseteq \{x x \prec z_i\}}$	(S-CSTRWIDTH)
$\frac{\text{for each } i \ \{x x \prec y_i\} \subseteq \{x x \prec z_i\}}{\phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x \prec y_i\} <:: \phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x \prec z_i\}}$	(S-CSTRDEPTH)
Type derivation	$T \mapsto T'$
$\frac{T \mapsto T'}{T.c \mapsto T'.c}$	(TD-SUBS)