# Aufgabe 2: "XMΛ Typechecker"

## 1 Introduction

In recent years, the Extensible Markup Language XML[1] has been widely adopted as a format to exchange data. In order to restrict the contents of XML instance documents to particular data types and content models, XML Schema Definition XSD[2] provides a number of fundamental- and constraining facets. Based on predefined built-in data types and lexical- and value space restrictions it is possible to derive application specific atomic XML data types. For example, in order to represent a company's product IDs ranging from 0 to 999 it is possible to derive an atomic data type `#productID` from the built-in XSD data type `xsd#nonNegativeInteger` by restricting the value space of non-negative integers to numbers less than 1000.

Obeying such lexical– and value space constraints, which may be imposed on XSD simple data types, is still error-prone and laborious. Validating XML instance documents according to given XML Schema Definitions requires significant amounts of code. Since corporations from all sectors have braced to store and process data using XML based technologies, providing programming language inherent support for XML Schema Definition data types may be a business model for an emerging start-up company...

You are with *XMΛ Inc.* - a small startup founded by enthusiastic programmers, who aim to augment widely used programming languages such as Java or C# with inherent support for XML Schema Definition data types.

## 2 Your Task

The extension of conventional object oriented programming languages with support for XML- or SQL data types has been the subject of several collegiate and industrial programming projects. However, most approaches are either restricted to type checking of compound data types[3] or do not make XSD constraints an inherent feature of the programming language.

In order to enable first assessments how a programming language with inherent support for constrained atomic data types such as `#productID` could behave in terms of evaluation and subtyping, you decide to base your research on this topic on the well known lambda-calculus[4] [Chu41]. In particular, your plan is

[1] http://www.w3.org/XML

[2] http://www.w3.org/XML/Schema

[3] Compound XSD datatypes are described in the *XML Schema Part 1: Structures* W3C recommendation. Such complex data types, however, can be ignored for now. This task description considers only constrained atomic data types as described in the *XML Schema Part 2: Datatypes* W3C recommendation!

[4] http://en.wikipedia.org/wiki/Lambda_calculus

to implement an interpreter and a typechecker for an extended version of the simply typed lambda-calculus with subtyping, which should include support for constrained atomic types as described by [BM04].

## 3 Considerations

You may start off developing a - possibly graphical - user interface and a parser that can be used to enter typed lambda-calculus ($\lambda_{<:}$) terms for which the syntax is given in Tables 1, 2, 3, and 4 and in [Pie02]. The $\lambda_{<:}$ abstract syntax trees can be used to develop a *beta-reduction* processor that evaluates $\lambda_{<:}$ terms such as $(\lambda x : T.x)y$ or $(\lambda x : T.x(\lambda x : T.x))(ur)$, which evaluate to $y$ and $ur(\lambda x : T.x)$, respectively.

Before the evaluation of $\lambda_{<:}$ terms, you should check that input terms are *well-typed*. Well-typed terms do not "go wrong". In particular, you can be sure that your beta-reduction processor will never reach a "stuck state".

$$\dfrac{\dfrac{}{\{x : \{a : T, b : T\}, y : \{m : T\}\} <: \{x : \{a : T, b : T\}\}} \quad \dfrac{\dfrac{}{\{a : T, b : T\} <: \{a : T\}}}{\{x : \{a : T, b : T\}\} <: \{x : \{a : T\}\}}}{\{x : \{a : T, b : T\}, y : \{m : T\}\} <: \{x : \{a : T\}\}}$$

Figure 1: Exemplary subtyping derivation

Given derivation statements such as in Figure 1 where the transitivity rule S-TRANS is used to combine the width- and depth subtyping rules S-RCDWIDTH and S-RCDDEPTH for record data types, your system should be able to check whether given types are in the subtype relation. In order to include constrained XML Schema Definition data types in your system you are to add constrained types to $\lambda_{<:}$. Constraining facets that you may want to consider are given in section 4.3 of the *XML Schema Part 2: Datatypes* W3C recommendation.

For an introduction to the simply typed lambda-calculus and some common extensions such as atomic base types, record data types, and subtyping you may refer to chapters 2, 3, 5, 8, 9, 11, and 15 in [Pie02].

Hint: You may model XSD constraints similarly like fields of record data types. In this way, constrained atomic types are tuples comprising a value space (e.g., integer numbers) and a set of constraints (e.g., minInclusive, maxInclusive) that lessen the value space. For example, for the two given types $T_1$ and $T_2$ shown below your system should be able to answer the question "Is $T_1$ a subtype of $T_2$?".

$T_1 \equiv \{xsd\#integer, \{minInclusive = 0, maxExclusive = 1000\}\}$

$T_2 \equiv \{xsd\#nonNegativeInteger, \{maxInclusive = 999\}\}$

Table 1: Typed lambda-calculus ($\lambda_{<:}$) syntax

| ***Syntax*** | |
|---|---|
| t ::= | *terms:* |
| $x$ | *variable* |
| $\lambda x : T.t$ | *abstraction* |
| $tt$ | *application* |
| $\{l_i = t_i \ ^{i \in 1..n}\}$ | *record* |
| $t.l$ | *projection* |
| v ::= | *values:* |
| $\lambda x : T.t$ | *abstraction value* |
| $\{l_i = v_i \ ^{i \in 1..n}\}$ | *record value* |
| T ::= | *types:* |
| $Top$ | *maximum type* |
| $A$ | *base type* |
| $T \to T$ | *type of functions* |
| $\{l_i : T_i \ ^{i \in 1..n}\}$ | *type of records* |
| $\Gamma$ ::= | *contexts:* |
| $\varnothing$ | *empty context* |
| $\Gamma, x : T$ | *term variable binding* |

Table 2: Typed lambda-calculus ($\lambda_{<:}$) evaluation

| ***Evaluation*** | $t \to t'$ |
|---|---|
| $\dfrac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$ | (E-APP1) |
| $\dfrac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'}$ | (E-APP2) |
| $(\lambda x : T_{11}.t_{12}) v_2 \to [x \mapsto v_2] t_{12}$ | (E-APPABS) |
| $\{l_i = v_i \ ^{i \in 1..n}\}.l_j \to v_j$ | (E-PROJRCD) |
| $\dfrac{t_1 \to t_1'}{t_1.l \to t_1'.l}$ | (E-PROJ) |
| $\dfrac{t_j \to t_j'}{\{l_i = v_i \ ^{i \in 1..j-1}, l_j = t_j, l_k = t_k \ ^{k \in j+1..n}\} \to \{l_i = v_i \ ^{i \in 1..j-1}, l_j = t_j', l_k = t_k \ ^{k \in j+1..n}\}}$ | (E-RCD) |

Table 3: Typed lambda-calculus ($\lambda_{<:}$) typing

| ***Typing*** | $\Gamma \vdash t : T$ |
|---|---|
| $\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$ | (T-VAR) |
| $\dfrac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$ | (T-ABS) |
| $\dfrac{\Gamma \vdash t_1 : T_{11} \to T_{12} \; \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$ | (T-APP) |
| $\dfrac{\text{for each } i \; \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i \; ^{i \in 1..n}\} : \{l_i = T_i \; ^{i \in 1..n}\}}$ | (T-RCD) |
| $\dfrac{\Gamma \vdash t_1 : \{l_i : T_i \; ^{i \in 1..n}\}}{\Gamma \vdash t_1 . l_j : T_j}$ | (T-PROJ) |
| $\dfrac{\Gamma \vdash t : S \; S <: T}{\Gamma \vdash t : T}$ | (T-SUB) |

Table 4: Typed lambda-calculus ($\lambda_{<:}$) subtyping

| ***Subtyping*** | $S <: T$ |
|---|---|
| $S <: S$ | (S-REFL) |
| $\dfrac{S <: U \; U <: T}{S <: T}$ | (S-TRANS) |
| $S <: Top$ | (S-TOP) |
| $\dfrac{T_1 <: S_1 \; S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$ | (S-ARROW) |
| $\{l_i : T_i \; ^{i \in 1..n+k}\} <: \{l_i : T_i \; ^{i \in 1..n}\}$ | (S-RCDWIDTH) |
| $\dfrac{\text{for each } i \; S_i <: T_i}{\{l_i : S_i \; ^{i \in 1..n}\} <: \{l_i : T_i \; ^{i \in 1..n}\}}$ | (S-RCDDEPTH) |
| $\dfrac{\{k_j : S_j \; ^{j \in 1..n}\} \text{ is a permutation of } \{l_i : T_i \; ^{i \in 1..n}\}}{\{k_j : S_j \; ^{j \in 1..n}\} <: \{l_i : T_i \; ^{i \in 1..n}\}}$ | (S-RCDPERM) |

# References

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium (W3C), 2004. Internet: http://www.w3.org/TR/xmlschema-2/, 2004.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[Pie02] Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

informatiCup
Der GI-Wettbewerb für Studierende

## Kontakt und Einreichung

Gesellschaft für Informatik e.V. (GI) · Ludger Porada · Ahrstraße 45 · 53175 Bonn
E-Mail: ludger.porada@gi-ev.de · http://www.informaticup.de

## Termine

- 20. Dezember 2006: Einsendeschluss der Beiträge
- 28. Februar 2007: Benachrichtigung der Teilnehmerinnen/Teilnehmer
- 30. – 31. März 2007: Endrunde
- 31. März 2007: Siegerehrung

## FAQ zur Aufgabenstellung und Einreichung

**Wer darf teilnehmen?**
Teilnehmen dürfen Studierende bis zum 5. Semester (Bachelor- oder Diplomstudium). Unser Wettbewerb richtet sich ausschließlich an Gruppen von mindestens zwei bis maximal vier Personen. Wenn Sie interessiert sind, suchen Sie Mitstreiterinnen und Mitstreiter. Die Teilnahme am Wettbewerb ist kostenlos.

**Wieviele Aufgaben müssen wir lösen?**
Für die Teilnahme am InformatiCup muss nur eine Aufgabe gelöst werden.

**Was müssen wir als Lösung einreichen?**
Eine gelöste Aufgabe umfasst:

- ein lauffähiges Programm
- eine Installationsanleitung
- ggf. eine Bedienungsanleitung
- eine schriftliche Ausarbeitung wie in den Aufgaben gefordert

**Wie können wir eine Lösung einreichen?**
In jeder Form, per Brief, auf Papier, mit CD, per E-Mail.

**Welche Programmiersprachen dürfen wir verwenden?**
Es darf jede Programmiersprache verwendet werden, die für die Lösung einer Aufgabe geeignet erscheint.

**Welche Bibliotheken dürfen wir verwenden?**
Es darf jede kostenlos verfügbare Bibliothek verwendet werden. Die Bibliothek muss nicht als Quelltext verfügbar sein.

**Wie werden die eingereichten Lösungen bewertet?**
Bei der Bewertung der eingereichten Lösungen werden sowohl der theoretische Lösungsansatz (z.B. Komplexität verwendeter Algorithmen) als auch die praktische Umsetzung (z.B. Bedienbarkeit und Zuverlässigkeit der Software) bewertet.