Grammar-based analysis of string expressions

Conference Paper · January 2005

DOI: 10.1145/1040294.1040300 · Source: DBLP

1 author:

Peter Thiemann
University of Freiburg
237 PUBLICATIONS **1,914** CITATIONS

SEE PROFILE

Grammar-Based Analysis of String Expressions

Peter Thiemann
Universität Freiburg
Georges-Köhler-Allee 079
D-79110 Freiburg, Germany
thiemann@informatik.uni-freiburg.de

ABSTRACT

We specify a polymorphic type system for an applied lambda calculus that refines the string type with a subtype hierarchy derived from language containment. It enables us to find a language for each string-type expression such that the value of the expression is a member of that language. Type inference for this system infers language inclusion constraints that can be viewed as a context-free grammar with a nonterminal for each string-valued expression.

Then we present two algorithms that solve language inclusion constraints with respect to a fixed context-free reference grammar. The solutions are sound but incomplete because the general problem of context-free language inclusion is undecidable. Both algorithms are derived from Earley's parsing algorithm for context-free languages.

Taking the two parts together enables us to answer questions like: Is the value of a string-type expression derivable from a given nonterminal in the reference grammar?

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms

Algorithms, Languages, Theory

Keywords

string expression analysis, type inference, constraints

1. INTRODUCTION

String values have lost their innocence and are being used in many unforeseen contexts. Particularly in scripting languages, strings serve as a universal data structure to communicate values and commands between separate program components. For example, strings are used to build output in XHTML format, to compose SQL statements, and to build XPath and JavaScript expressions which are passed to their respective interpreters or stored in an event attribute. Hence, strings often carry a hidden meaning which is not reflected in their type.

In all of the above cases, strings must adhere to a fixed external format. Examples are XHTML strings, SQL statements, and XPath and JavaScript expressions. In each case, there is a context-free grammar that defines the language of admissible strings. Also in each case a runtime error ensues, if the actual string value is not correctly formed. Ill-formed XHTML may make the web browser or a subsequently used XML processor choke. Ill-formed SQL statements are either rejected by the database or give rise to erroneous results. Finally, illegal expressions passed to an XPath processor or a JavaScript interpreter give rise to runtime exceptions.

The commonality of the examples is their wide-spread use in currently deployed web applications. For example, the JDBC interface to databases [11] requires SQL queries to be passed as strings. Also, the generation of XHTML is often a matter of constructing suitable strings and just writing them to the output. Many applications deal with XPath and JavaScript expressions in the same way.

Judging from the many errors of this kind encountered by even a casual web surfer, we are facing a serious software maintenance problem. How are we ever going to be sure that all those carelessly concatenated strings adhere to their desired format?

Our answer to this question is program analysis and string expression analysis in particular. String expression analysis has been considered before by two groups of researchers [9, 1]. Both of them model the possible values of a string expression by a regular language and rely on the decidability of inclusion for regular languages (see Sec. 9 for a detailed comparison). Regular languages yield highly useful results but they cannot guarantee that a string value is an element of a context-free language. For example, XHTML strings, SQL statements, XPath and JavaScript expressions are all definable by context-free grammars and cannot be described by regular languages in an entirely satisfactory manner.

However, the previous approaches do not generalize to context-free languages. Even though a context-free grammar can be constructed that describes the values of all string expression [1], it is not possible to test the language of such a grammar against a context-free reference language describing the desired values because language inclusion is undecidable for context-free languages [4].

The present work introduces a novel type-based framework for string expression analysis which guarantees that all

TLDI'05, January 10, 2005, Long Beach, California, USA.

values of a string expression are elements of a context-free language given by some reference grammar G. The framework is sound but incomplete because it "replaces" the undecidable inclusion property with the decidable derivability property.

Our string expression analysis is phrased as a constrained type system where the constraints are *parsing constraints* with respect to the reference grammar G. Intuitively, a parsing constraint expresses the derivability relation of G which is defined on words over terminal and nonterminal symbols of G. Since derivability is decidable for context-free grammars, the constraints can be solved effectively. In particular, we exhibit a generalization of Earley's parsing algorithm for context-free grammars [2] for solving the constraints. Thanks to the type-based formulation, the analysis is polymorphic and applies to a polymorphic base language.

There are two caveats with our framework. First, a lot depends on the proper choice of the grammar to ensure that Earley's parser can solve all constraints. In particular, grammars that are left- or right-recursive will not parse constraints successfully if the program builds the string under analysis the other way round (see Section 8.1 for discussion). The chances for a successful parse are much higher with an ambiguous grammar that does not prefer any particular direction of building lists.

Second, since the analysis works directly on the level of characters, the grammar must be given at the same level. A grammar that assumes a preceding scanner and uses tokens as terminal symbols will not work. Instead, the grammar must specify the syntax down to the character level, similar to a grammar for scannerless parsing.

In the paper, we first define a polymorphic string expression analysis as a reference analysis in Sec. 2. The reference analysis does not restrict the constraints to derivations of a context-free grammar, but rather interprets them as inclusions of arbitrary languages. Stating the analysis in this way simplifies the proof of soundness and makes it straightforward to transfer results for constrained type systems (for example, the type reconstruction algorithm) to the specific setting as shown in Sec. 3. The next steps starting with Sec. 4 restrict the notion of constraint model to those which verify the inclusion constraints by parsing with respect to the reference grammar and investigate the constraint simplification facilitated by that restriction. In particular, Sec. 5 defines constraint simplification for parsing constraints using a generalization of Earley's parsing algorithm. Sec. 6 discusses a naive algorithm to translate inclusion constraints to assignment constraints and shows how to resolve them. Sec. 7 refines the naive algorithm to an inference algorithm that discovers assignments during parsing. We conclude by discussing some examples and extensions in Sec. 8, a detailed comparison with related work in Sec. 9, and some final remarks in Sec. 10.

2. POLYMORPHIC STRING EXPRESSION ANALYSIS

The task of a string analysis is to determine for each occurrence of a string type in a type derivation a set of strings such that each value computed by the corresponding expression is a member of this set. This section develops a reference specification of a string analysis that essentially infers a context-free grammar from a given program such that each

Alphabet	T		
Symbols	$\mathtt{a}, \mathtt{b}$	$\in$	T
Words	w	$\in$	T^*
Constants	c	$\in$	$\{w \in T^*\} \cup \{\cdot, \mathtt{if}\ \ \}$
Expressions	e	$::=$	$c \mid x \mid e\ (e) \mid \mathtt{rec}\ f(x)\ e \mid$
			$\mathtt{let}\ x = e\ \mathtt{in}\ e$
Values	v	$::=$	$\mathtt{rec}\ f(x)\ e \mid c$
Ev. Contexts	E	$::=$	$[\,] \mid E\ (e) \mid e\ (E)$

Beta reduction

$$(\mathtt{rec}\ f(x)\ e)\ (v) \quad \longrightarrow \quad e[x \mapsto v, f \mapsto \mathtt{rec}\ f(x)\ e]$$

Delta reduction

$$\begin{array}{lcl} \mathtt{if}\ (\mathtt{a}\mathtt{b}_1 \ldots \mathtt{b}_m)\ e_1\ e_2 & \longrightarrow & e_1 \\ \mathtt{if}\ \varepsilon\ e_1\ e_2 & \longrightarrow & e_2 \\ \mathtt{a}_1 \ldots \mathtt{a}_n \cdot \mathtt{b}_1 \ldots \mathtt{b}_m & \longrightarrow & \mathtt{a}_1 \ldots \mathtt{a}_n \mathtt{b}_1 \ldots \mathtt{b}_m \end{array}$$

Reduction

$$\frac{e \longrightarrow e'}{E[e] \mapsto E[e']}$$

Figure 1: Syntax and dynamic semantics

Types	τ	$::=$	$\alpha \mid \mathtt{Str}(\varphi) \mid \tau \to \tau$
Language Variables	φ	$\in$	Φ
Constrained Types	ρ	$::=$	$C \Rightarrow \tau$
Constraints	C	$::=$	$\mathsf{true} \mid \tau \dot{\leq} \tau \mid r \dot{\subseteq} \varphi \mid C \wedge C$
String Type Indices	r	$::=$	$\varphi \mid \varepsilon \mid \mathtt{a} \mid r \cdot r$
Type Schemes	σ	$::=$	$\forall \tilde{\varphi} \tilde{\alpha}.\rho$
Type Environments	Γ	$::=$	$\emptyset \mid \Gamma(x : \sigma)$

Figure 2: Type language

$$L_\Theta(\varphi) = \Theta(\varphi) \qquad L_\Theta(\varepsilon) = \{\varepsilon\} \qquad L_\Theta(\mathtt{a}) = \{\mathtt{a}\}$$
$$L_\Theta(r_1 \cdot r_2) = \{w_1 \cdot w_2 \mid w_1 \in L_\Theta(r_1), w_2 \in L_\Theta(r_2)\}$$

Figure 3: Meaning of string type indices

string value in the program can be derived from that grammar. The analysis is phrased as a type system where the string type is refined with a suitable index structure.

The language we consider is an applied lambda calculus with recursive functions, a conditional that checks emptiness of a string, and operations for string construction (string constants and string concatenation). Strings are sequences of terminal symbols drawn from a finite set T with ε denoting the empty string. Figure 1 defines the syntax and the semantics of the calculus. The semantics is given in small step operational style with evaluation contexts [12]. The contexts do not fix the evaluation order entirely, they admit both call-by-value and call-by-name reduction. As usual, $\overset{*}{\mapsto}$ denotes the reflexive transitive closure of the reduction relation $\mapsto$. All string-related operations (conditional, string constants, and string concatenation) are included in the calculus via constants with delta reductions.

Figure 2 defines the syntax of types. A type is either a type variable, a string type, or a function type. Each string type is indexed with a language variable φ. A language variable stands for a language over T. Types may be constrained by a conjunction of subtyping constraints $\tau \dot{\leq} \tau'$ and inclusion constraints $r \dot{\subseteq} \varphi$ where r is a sequence of terminal symbols and language variables and $\cdot$ is regarded as an associative operation. Constrained types may be abstracted over a sequence of type and language variables (using the shorthand $\tilde{\varphi}$ for $\varphi_1 \ldots \varphi_n$ for some n). The function $fv(\cdot)$ is applicable to all kinds of constraints and type phrases. It extracts the set of free type and language variables, with the standard definition.

The type algebra is many-sorted. Besides the usual kind of types, there is another kind of string indices. In this view, a language variable is just type variable with a different kind.

There is a subtyping relation which is structural and fully determined by the inclusion constraints on the string type indices. Since string type indices (and hence types, constrained types, constraints, and type schemes) may contain language variables, the meaning of a type phrase is relative to an assignment $\Theta : \Phi \to \mathcal{P}(T^*)$ that maps a language variable to a set of strings. In particular, Figure 3 defines $L_\Theta(r)$ to be the set of strings denoted by r under assignment Θ.

A model for a constraint C is pair (S, Θ) where S is a substitution and Θ an assignment (written $S, \Theta \models C$) subject to the following conditions.

1. The substitution S maps type variables to types and language variables to language variables. S must be idempotent.
2. The assignment Θ maps language variables to sets of strings.
3. Θ must be defined on all language variables in C and $S(C)$ and it must be compatible with S: for all φ, $\Theta(\varphi) = \Theta(S(\varphi))$.
4. $S, \Theta \models C$ must be derivable using the inference rules in Fig. 4. The last rule (application of the substitution) need only be used at most once in a derivation because substitutions are idempotent.

$$S, \Theta \models \mathsf{true}$$
$$S, \Theta \models \alpha \dot{\leq} \alpha$$
$$\frac{\Theta(\varphi_1) \subseteq \Theta(\varphi_2)}{S, \Theta \models \mathtt{Str}(\varphi_1) \dot{\leq} \mathtt{Str}(\varphi_2)}$$
$$\frac{S, \Theta \models \tau_2' \dot{\leq} \tau_2 \quad S, \Theta \models \tau_1 \dot{\leq} \tau_1'}{S, \Theta \models \tau_2 \to \tau_1 \dot{\leq} \tau_2' \to \tau_1'}$$
$$\frac{L_\Theta(r) \subseteq \Theta(\varphi)}{S, \Theta \models r \dot{\subseteq} \varphi}$$
$$\frac{S, \Theta \models C_1 \qquad S, \Theta \models C_2}{S, \Theta \models C_1 \wedge C_2}$$
$$\frac{S, \Theta \models S(C)}{S, \Theta \models C}$$

Figure 4: Constraint satisfaction

$$\frac{(x : \sigma) \in \Gamma}{C, \Gamma \vdash x : \sigma} \qquad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \dot{\leq} \tau'}{C, \Gamma \vdash e : \tau'}$$
$$\frac{C, \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\ (e_2) : \tau_1}$$
$$\frac{C, \Gamma(f : \tau_2 \to \tau_1)(x : \tau_2) \vdash e : \tau_1}{C, \Gamma \vdash \mathtt{rec}\ f(x)\ e : \tau_2 \to \tau_1}$$
$$\frac{C, \Gamma \vdash e : \sigma \qquad C, \Gamma(x : \sigma) \vdash e' : \tau'}{C, \Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau'}$$
$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \{\tilde{\alpha}, \tilde{\varphi}\} \cap (fv(C) \cup fv(\Gamma)) = \emptyset}{C, \Gamma \vdash e : \forall \tilde{\alpha}\tilde{\varphi}.D \Rightarrow \tau}$$
$$\frac{C, \Gamma \vdash e : \forall \tilde{\alpha}\tilde{\varphi}.D \Rightarrow \tau \qquad S = [\tilde{\alpha} \mapsto \tilde{\tau}, \tilde{\varphi} \mapsto \tilde{\varphi}'] \qquad C \Vdash S(D)}{C, \Gamma \vdash e : S(\tau)}$$

Figure 5: Typing rules

We define constraint entailment $C \Vdash C'$ in the usual way by $C \Vdash C'$ iff $(S, \Theta) \models C$ entails $(S, \Theta) \models C'$.

The rules in Figure 5 define the deduction system for the typing judgment $C, \Gamma \vdash e : \sigma$, which relates a constraint, a type environment, and an expression to a type scheme. Intuitively, an expression of type $\mathtt{Str}(\varphi)$ under constraint C can only take on string values drawn from $\Theta(S(\varphi))$ for some S, Θ where $S, \Theta \models C$. The rules for the lambda calculus and the subsumption rule are taken from the logical presentation of the system HM(X) [8]. The only difference is the introduction rule for polymorphism. The HM(X) rule states that the abstracted constraint D must be solvable by requiring $\exists \tilde{\alpha}\tilde{\varphi}.D$. Omitting this constraint effectively defer checking of solvability to the points of use. All operations for strings (constants w, string concatenation $\cdot$, and the conditional `if`) are made available through an initial type environment which defines

$$\begin{array}{ll} w & : \forall \varphi.(w \dot{\subseteq} \varphi) \Rightarrow \mathtt{Str}(\varphi) \\ \cdot & : \forall \varphi_1, \varphi_2, \varphi.(\varphi_1 \cdot \varphi_2 \dot{\subseteq} \varphi) \Rightarrow \mathtt{Str}(\varphi_1) \to \mathtt{Str}(\varphi_2) \to \mathtt{Str}(\varphi) \\ \mathtt{if} & : \forall \alpha, \varphi.\mathtt{Str}(\varphi) \to \alpha \to \alpha \to \alpha \end{array}$$

Subtyping is a necessary ingredient of the system. It is required to approximate the branches of conditionals, to de-

fine recursive functions, and to use functions in more than one context.

Our use of a many-sorted type algebra (the sorts are types and languages, as evidenced by type and language variables) is only a minor extension fo HM(X). All results for HM(X) carry over to the many-sorted case as already observed and stated by the HM(X) authors [8, Example 8].

It is a standard exercise to prove that this system is sound by proving type preservation and progress.

Theorem 1 *Suppose that $C, \emptyset \vdash e : \tau$ and there exist S, Θ such that $S, \Theta \models C$. Then either e diverges or there is some v such that $e \overset{*}{\mapsto} v$ and $C, \emptyset \vdash v : \tau$.*

One part of the progress property is the establishment of a canonical forms lemma. While this lemma is usually not of much interest, we show it here because it states the connection between constraints, indexed string types, and the resulting values.

Lemma 1 *Suppose that $C, \emptyset \vdash v : \tau$.*

1. *If $\tau = \tau_2 \to \tau_1$, then $v = \texttt{rec}\ f(x)\ e$, for some f, x, and e.*
2. *If $\tau = \texttt{Str}(\varphi)$, then $v = w$, for some $w \in T^*$, and $w \in \Theta(S(\varphi))$, for all S, Θ, such that $S, \Theta \models C$.*

From this specification of the type system, we proceed first, in Section 3, to an algorithmic version of the system. This system attempts to reduce subtyping constraints as much as possible but it defers all inclusion constraints $r \dot{\subseteq} \varphi$. It essentially extracts constraints that correspond to a context-free grammar for describing all string sets that occur in a type. Subsequent sections investigate further simplification rules for solving inclusion constraints.

3. ALGORITHMIC STRING EXPRESSION ANALYSIS

To arrive at a grammar describing the string sets in the types, we transform the logical presentation of the type system in the previous section to a syntax-based, algorithmic formulation. This transformation is exactly as prescribed by HM(X) [8] and its result amounts to a type reconstruction algorithm for polymorphic string analysis.

Stating the algorithmic system requires a term constraint system. Hence, constraints C receive three new alternatives, two substitutions and failure, all of which only arise during constraint simplification. Here is the complete list of alternatives for C:

- $\alpha \doteq \tau$ (substitution for type variables),
- $\varphi \doteq \varphi'$ (substitution for language variables),
- $\tau \dot{\leq} \tau'$ (subtyping),
- $r \dot{\subseteq} \varphi$ (language inclusion, $r \in (\Phi \cup T)^*$),
- $C' \wedge C''$ (conjunction of constraints),
- true and fail.

$$\frac{S(\alpha) = \tau}{S, \Theta \models \alpha \doteq \tau} \qquad S, \Theta \models \varphi \doteq \varphi$$

Figure 7: Constraint satisfaction, part 2

The notion of a model of a constraint $S, \Theta \models C$ needs to be extended accordingly. Figure 7 contains the additional inference rules. To preserve the form of substitution constraints, we define $S(\alpha \doteq \tau) = \alpha \doteq S(\tau)$, that is, the substitution only applies to the right hand side of a substitution constraint. Despite the symmetric symbol $\doteq$, the constraint $\alpha \doteq \tau$ denotes a left-to-right substitution, *i.e.*, the two sides must not be exchanged.

The type reconstruction algorithm in Fig. 6 is taken almost verbatim from the HM(X) paper [8]. The only difference is in the rules *(rec')* and *(if')*: HM(X) does not consider recursion and conditional as builtin features but rather stipulates that they are included via typed constants (as with string constants and concatenation). It defines the type reconstruction judgment

$$S, C, \Gamma \vdash' e : \tau$$

where the type environment Γ and the expression e are the inputs and the substitution S, the constraint C, and the type τ are the outputs. The function *normalize* simplifies constraints and performs all necessary unifications. Section 3.1 explains it and states some of its properties. If a constraint normalizes to fail (perhaps due to a failing unification), then the inference rule calling *normalize* fails, too, and no derivation exists. To enable a uniform treatment we consider a substitution as a conjunction of substitution constraints and vice versa. We convert between the two representations as convenient.

Due to our reliance on the HM(X) framework, soundness and completeness of type reconstruction with respect to the logical system is immediate from the corresponding results for HM(X) [8, Theorem 11 and 12].

Theorem 2 (Soundness of reconstruction) *Suppose that $S, C, \Gamma \vdash' e : \tau$. Then $S(C), S(\Gamma) \vdash e : S(\tau)$.*

The statement of the completeness theorem requires further auxiliary notions, so we refrain from including it at this point.

3.1 Constraint Normalization

The function *normalize* maps a constraint to a pair of a substitution and a constraint. It is defined by exhaustive application of the normalization rules in Figure 8. The rules are given in the style of constraint-handling rules [3]. A single rule application to constraint C works as follows.

- Split C into $C = C_0 \wedge C_1$ taking advantage of associativity and commutativity of $\wedge$ as well as of the unit element true.
- Applying rule $C_1 \Leftrightarrow C_2$ replaces constraint C_1 by constraint C_2. The final constraint is $C_0 \wedge C_2$.
- Rule $C_1 \Rightarrow C_2$ is only applicable if C_2 is not yet present in C. In this case the rule adds constraint C_2 and yields the final constraint $C_0 \wedge C_1 \wedge C_2$.

$$\textit{(var')}\quad \frac{(x:\sigma)\in\Gamma \quad \sigma=\forall\tilde{\alpha}\tilde{\varphi}.C'\Rightarrow\tau' \quad \tilde{\beta},\tilde{\varphi}' \textbf{ fresh} \quad S_0=[\tilde{\alpha}\mapsto\tilde{\beta},\tilde{\varphi}\mapsto\tilde{\varphi}'] \quad (S,C)=\mathit{normalize}(S_0(C'))}{S|_{fv(\Gamma)},C,\Gamma\vdash' x:S(\tau')}$$

$$\textit{(rec')}\quad \frac{S,C,\Gamma(f:\alpha\to\alpha')(x:\alpha)\vdash' e:\tau \quad \alpha,\alpha' \textbf{ fresh} \quad (S',C')=\mathit{normalize}(S\wedge C\wedge\tau\dot{\leq}\alpha')}{S'|_{fv(\Gamma)},C',\Gamma\vdash' \texttt{rec } f(x)\ e:S'(\alpha\to\tau)}$$

$$\textit{(app')}\quad \frac{S,C,\Gamma\vdash' e:\tau \qquad S',C',\Gamma\vdash' e':\tau' \quad \alpha \textbf{ fresh} \quad (S'',C'')=\mathit{normalize}(S\wedge S'\wedge C\wedge C'\wedge\tau\dot{\leq}\tau'\to\alpha)}{S''|_{fv(\Gamma)},C'',\Gamma\vdash' e\ (e'):S''(\alpha)}$$

$$\textit{(let')}\quad \frac{S_0,C_0\wedge C_1,\Gamma\vdash' e:\tau \qquad S',C',\Gamma(x:\forall\tilde{\alpha}\tilde{\varphi}.C_1\Rightarrow\tau)\vdash' e':\tau' \quad \{\tilde{\alpha},\tilde{\varphi}\}=fv(\tau)\setminus fv(\Gamma) \quad fv(C_0)\subseteq fv(\Gamma) \quad (S'',C'')=\mathit{normalize}(S_0\wedge S'\wedge C_0\wedge C')}{S''|_{fv(\Gamma)},C'',\Gamma\vdash' \texttt{let } x=e \texttt{ in } e':S''(\tau')}$$

Figure 6: Constraint-based Type Reconstruction Rules

$\texttt{Str}(\varphi)\dot{\leq}\texttt{Str}(\varphi')$	$\Leftrightarrow$	$\varphi\dot{\subseteq}\varphi'$
$\texttt{Str}(\varphi)\dot{\leq}\tau\to\tau'$	$\Leftrightarrow$	fail
$\texttt{Str}(\varphi)\dot{\leq}\alpha$	$\Rightarrow$	$\alpha\doteq\texttt{Str}(\varphi')$ **if** φ' **fresh**
$\tau\to\tau'\dot{\leq}\texttt{Str}(\varphi')$	$\Leftrightarrow$	fail
$\tau\to\tau'\dot{\leq}\tau_1\to\tau_1'$	$\Leftrightarrow$	$\tau_1\dot{\leq}\tau\wedge\tau'\dot{\leq}\tau_1'$
$\tau\to\tau'\dot{\leq}\alpha$	$\Rightarrow$	$\alpha\doteq\alpha'\to\alpha''$ **if** α',α'' **fresh**
$\alpha\dot{\leq}\texttt{Str}(\varphi')$	$\Rightarrow$	$\alpha\doteq\texttt{Str}(\varphi)$ **if** φ **fresh**
$\alpha\dot{\leq}\tau_1\to\tau_1'$	$\Rightarrow$	$\alpha\doteq\alpha'\to\alpha''$ **if** α',α'' **fresh**
$\alpha\dot{\leq}\alpha$	$\Leftrightarrow$	true
$\alpha\dot{\leq}\alpha'\wedge\alpha'\dot{\leq}\alpha''$	$\Rightarrow$	$\alpha\dot{\leq}\alpha''$
$\alpha\dot{\leq}\alpha'\wedge\alpha'\dot{\leq}\alpha$	$\Leftrightarrow$	$\alpha\doteq\alpha'$
$\alpha\doteq\alpha$	$\Leftrightarrow$	true
$\varphi\doteq\varphi$	$\Leftrightarrow$	true
$\varphi\dot{\subseteq}\varphi$	$\Leftrightarrow$	true
$\varphi\dot{\subseteq}\varphi'\wedge\varphi'\dot{\subseteq}\varphi$	$\Leftrightarrow$	$\varphi\doteq\varphi'$
$C@(C'\wedge r\dot{\subseteq}\varphi\wedge r_1\varphi r_2\dot{\subseteq}\varphi')$	$\Rightarrow$	$r_1rr_2\dot{\subseteq}\varphi'$ **if** $\varphi\notin\mathit{reach}(C,\varphi)$
$C\wedge\varphi\doteq\varphi'$	$\Leftrightarrow$	$C[\varphi\mapsto\varphi']\wedge\varphi\doteq\varphi'$
$C\wedge\alpha\doteq\tau$	$\Leftrightarrow$	$C[\alpha\mapsto\tau]\wedge\alpha\doteq\tau$ **if** $\alpha\notin fv(\tau)$
$C\wedge\alpha\doteq\tau$	$\Leftrightarrow$	fail **if** $\alpha\in fv(\tau)\wedge\tau\neq\alpha$
$C\wedge$ true	$\Leftrightarrow$	C
$C\wedge$ fail	$\Leftrightarrow$	fail

Figure 8: Constraint normalization rules

- The rules from the second group must be applied to the entire constraint C (that is, $C_0=\mathsf{true}$). The notation $C@(C')$ is an as-pattern: it binds the entire constraint to C and matches parts of C according to C'.

- The side condition on the first rule of the second group is required to ensure termination. It is similar to an occurs check and relies on the function $\mathit{reach}(C,\varphi)$ which yields the set of language variables reachable in one or more steps from φ in the dependency graph induced by the constraint C. The dependency graph has node set Φ and a directed edge $\varphi\to\varphi'$ for each inclusion $r_1\varphi' r_2\dot{\subseteq}\varphi\in C$.

When no more rule is applicable, the resulting normalized constraint is split into the substitution constraints of the form $\alpha\doteq\tau$ and $\varphi\doteq\varphi'$ (that make up the substitution) and the rest.

Lemma 2 *The rules in Figure 8 are sound and complete: Suppose that C rewrites to C'. Then $S,\Theta\models C$ if and only if there are extensions S' of S and Θ' of Θ such that $S',\Theta'\models C'$.*

Lemma 3 *The rules in Figure 8 are confluent and terminating.*

Exhaustive application of the propagation rules yields a constraint in normal form.

Definition 1 *A constraint C is in* normal form *if C is either* true *or* fail *or all conjuncts in C have one of the forms*

1. *$\alpha\doteq\tau$ and $\alpha\notin fv(\tau)$ and α does not occur in the rest of C,*
2. *$\alpha\dot{\leq}\alpha'$ where α and α' are different,*
3. *$\varphi\doteq\varphi'$ where φ and φ' are different,*
4. *$r\dot{\subseteq}\varphi$ and, for all possible partitions $r_1\varphi' r_2=r$, either $\mathrm{reach}(C,\varphi')=\emptyset$, $\varphi'\in\mathrm{reach}(C,\varphi')$, or there exists some r' such that $r_1r'r_2\dot{\subseteq}\varphi\wedge r'\dot{\subseteq}\varphi'\in C$.*

Lemma 4 *If no application of a rule from Fig. 8 changes C, then C is in normal form.*

3.2 Cropping

A constraint in normal form can become quite large due to the resolution of transitivity. However, many of the constraint's conjuncts only serve to define and propagate intermediate results which may be dropped after propagation is finished. The following example illustrated this point.

Example 1 *Consider the term $\lambda x.\mathtt{a}\cdot x$. It gives rise to the following constrained type.*

$$\begin{array}{ll} & \mathtt{a} \dot{\subseteq} \varphi_1 \\ \wedge & \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4 \\ \wedge & \mathtt{Str}(\varphi_2) \to \mathtt{Str}(\varphi_3) \to \mathtt{Str}(\varphi_4) \dot{\leq} \mathtt{Str}(\varphi_1) \to \alpha_2 \\ \wedge & \alpha_2 \dot{\leq} \alpha_1 \to \alpha_3 \\ \Rightarrow & \alpha_1 \to \alpha_3 \end{array}$$

Normalization and expansion of the substitutions leads to

$$\begin{array}{ll} & \mathtt{a} \dot{\subseteq} \varphi_1 \wedge \varphi_1 \dot{\subseteq} \varphi_2 \wedge \mathtt{a} \dot{\subseteq} \varphi_2 \\ \wedge & \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4 \wedge \mathtt{a} \cdot \varphi_3 \dot{\subseteq} \varphi_4 \wedge \varphi_2 \cdot \varphi_3' \dot{\subseteq} \varphi_4 \wedge \mathtt{a} \cdot \varphi_3' \dot{\subseteq} \varphi_4 \\ \wedge & \varphi_4 \dot{\subseteq} \varphi_4' \wedge \varphi_3' \dot{\subseteq} \varphi_3 \wedge \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4' \wedge \mathtt{a} \cdot \varphi_3 \dot{\subseteq} \varphi_4' \\ \wedge & \varphi_2 \cdot \varphi_3' \dot{\subseteq} \varphi_4' \wedge \mathtt{a} \cdot \varphi_3' \dot{\subseteq} \varphi_4' \\ \Rightarrow & \mathtt{Str}(\varphi_3') \to \mathtt{Str}(\varphi_4') \end{array}$$

To avoid the explosion of constraints, we propose *cropping* to get rid of as many language variables as possible. More concretely, let V be a set of language variables and C be a constraint. With V defining the set of variables that are mentioned in the result type or the typing environment, we are interested in a conjunct C' of C such that any solution of C' extends to a solution of C.

Hence, we are looking for $C = C' \wedge C''$ such that $S, \Theta \models C$ if and only if there exist S', Θ' such that $S', \Theta' \models C'$ and $S|_V = S'|_V$ and $\Theta|_V = \Theta'|_V$. To find C', we assume that C is in normal form and set

$$\begin{array}{rl} V' = V & \cup\{\varphi' \in \mathit{reach}(C, V) \mid \mathit{reach}(C, \varphi') = \emptyset\} \\ & \cup\{\varphi' \in \mathit{reach}(C, V) \mid \varphi' \in \mathit{reach}(C, \varphi')\} \end{array}$$

Then we select those constraints that only mention variables in V' and keep all subtyping constraints.

$$\begin{array}{rcl} C' & = & \bigwedge\{r \dot{\subseteq} \varphi \in C \mid \mathit{fv}(r \dot{\subseteq} \varphi) \subseteq V'\} \\ & \wedge & \bigwedge\{\tau \dot{\leq} \tau' \in C\} \end{array}$$

Lemma 5 *Let C and C' be as outlined in the preceding paragraph.*

1. *If $S, \Theta \models C$, then there exists S', Θ' with $S', \Theta' \models C'$ and $S|_V = S'|_V$ and $\Theta|_V = \Theta'|_V$.*
2. *If $S', \Theta' \models C'$, then there exists S and Θ with $S, \Theta \models C$ and $S'|_V = S|_V$ and $\Theta'|_V = \Theta|_V$.*

From now on, we assume that normalization also crops the constraint in each type reconstruction rule with respect to the variables in the type environment and the result type after applying the computed substitution.

Example 2 *If we apply cropping to the previously constructed constrained type, the resulting constraint is much smaller:*

$$\mathtt{a} \cdot \varphi_3' \dot{\subseteq} \varphi_4' \Rightarrow \mathtt{Str}(\varphi_3') \to \mathtt{Str}(\varphi_4')$$

At this point, each inclusion constraint like $r \dot{\subseteq} \varphi$ may be considered as context-free production with the language variables as nonterminals. Thus we have arrived at a similar situation as Christensen et al [1], who extract a context-free grammar with extended right-hand sides from the results of a flow analysis.

However, we take a different approach from now on. While Christensen et al apply the Mohri-Nederhof algorithm [7] to transform the resulting grammar into a regular grammar, we attempt to simplify the inclusion constraints directly using the reduction relation of a context-free reference grammar.

$$S, \Theta \models_G \mathsf{true} \qquad \frac{S(\alpha) = \tau}{S, \Theta \models_G \alpha \doteq \tau} \qquad \frac{\varphi = \varphi'}{S, \Theta \models_G \varphi \doteq \varphi'}$$

$$\frac{\Theta(\varphi') \stackrel{*}{\Rightarrow}_G \Theta(\varphi)}{S, \Theta \models_G \mathtt{Str}(\varphi) \dot{\leq} \mathtt{Str}(\varphi')}$$

$$\frac{S, \Theta \models_G \tau_1' \dot{\leq} \tau_1 \quad S, \Theta \models_G \tau_2 \dot{\leq} \tau_2'}{S, \Theta \models_G \tau_1 \to \tau_2 \dot{\leq} \tau_1' \to \tau_2'} \qquad S, \Theta \models_G \alpha \dot{\leq} \alpha$$

$$\frac{\Theta(\varphi) \stackrel{*}{\Rightarrow}_G \Theta(r)}{S, \Theta \models_G r \dot{\subseteq} \varphi} \qquad \frac{S, \Theta \models_G C \quad S, \Theta \models C'}{S, \Theta \models_G C \wedge C'}$$

$$\frac{S, \Theta \models_G S(C)}{S, \Theta \models_G C}$$

Figure 9: Constraint satisfaction wrt. reference grammar G

4. SIMPLIFICATION WITH RESPECT TO A REFERENCE GRAMMAR

The normalization rules from Figure 8 only compute the transitive closure of inclusion constraints but do not attempt to simplify them in any way. To obtain a useful notion of simplification for inclusion constraints, recall that a string analysis problem is always posed in a particular context. Let's assume that such a context demands that a certain string-type expression must evaluate to (a string containing) a valid XPath expression, an SQL statement, or an XML fragment. In each of these cases, the set of legal strings is a context-free language, specified by a context-free reference grammar $G = (N, T, P, S)$, where N is the set of nonterminal symbols, T is the set of terminal symbols, $P \subseteq N \times (N \cup T)^*$ the set of productions, and $S \in N$ the start symbol.

Hence, we are no longer interested in *all* models of a constraint, but rather in those models (S, Θ) where the assignment Θ maps language variables to languages *defined in terms of the reference grammar G*. There are several choices for defining a language in terms of G. For now, we will concentrate on the languages $L_A(G) = \{w \in T^* \mid A \stackrel{*}{\Rightarrow} w\}$, that is, the set of strings derivable from $A \in N$ using the productions of G (where $\Rightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$ is the usual derivation relation on strings of grammar symbols of G and $\stackrel{*}{\Rightarrow}$ its reflexive transitive closure).

Since a language assignment Θ is now restricted to map language variables to one of $L_A(G)$, where A is a nonterminal of G, each possible Θ is completely defined by a mapping $\hat{\Theta} : \Phi \to N$ from language variables to nonterminals. Solving the inclusion constraint $r \dot{\subseteq} \varphi$ in this setting means to look for a nonterminal assignment $\hat{\Theta} : \Phi \to N$ of language variables to nonterminals N such that there exists a derivation $\Theta(\varphi) \stackrel{*}{\Rightarrow} \Theta(r)$, where $\Theta(\varphi) = L_{\hat{\Theta}(\varphi)}(G)$. In the following, we will overload Θ to stand for both, the nonterminal assignment $\hat{\Theta}$ and its induced language assignment.

To cater for this notion of model with respect to the reference grammar G requires a refined definition. The new definition replaces the subset relation on languages by the G-derivability between strings of grammar symbols.

Definition 2 *Let S be a substitution and $\Theta : \Phi \to N$ be a mapping that assigns nonterminals to language variables.*

(S, Θ) is a G-model of constraint C if $S, \Theta \models_G C$ is derivable using the rules in Figure 9.

4.1 Parsing Assertions

Usually, there is no need for an overall solution of the constraints in terms of the reference grammar. Only a few places in a program require strings to be in a specific format. A program makes the demand for a certain format explicit by using *parsing assertions*. A parsing assertion is an expression that states that the string value of the expression must be derivable from a certain nonterminal symbol $A \in N$ of G:

$$e ::= \ldots \mid e \sqsubset A \qquad\qquad E ::= \ldots \mid E \sqsubset A$$

Its semantics is defined by the following notion of reduction

$$\mathtt{a}_1 \ldots \mathtt{a}_n \sqsubset A \quad \longrightarrow \quad \mathtt{a}_1 \ldots \mathtt{a}_n \qquad \textbf{if } A \stackrel{*}{\Rightarrow}_G \mathtt{a}_1 \ldots \mathtt{a}_n$$

and its type is defined by

$$\cdot \sqsubset A : \forall\varphi.(\varphi \mathrel{\dot\subseteq} A) \Rightarrow \mathtt{Str}(\varphi) \to \mathtt{Str}(\varphi)$$

where $\varphi \mathrel{\dot\subseteq} A$ is a *containment constraint* that is not touched by constraint propagation (recall that inclusion constraints always have a language variable on the right side). Constraint satisfaction extends to containment constraints by

$$\frac{A \stackrel{*}{\Rightarrow}_G \Theta(\varphi)}{S, \Theta \models_G \varphi \mathrel{\dot\subseteq} A}$$

Soundness and completeness of constraint normalization is not affected by the addition of this constraint because it is not involved in the propagation rules.

As an example for a use of a parsing assertion, consider a program in an extended version of our calculus that has access to a Lisp interpreter via a builtin function `eval`. `Eval` takes a string containing an sexpression and returns the result value as a string, too. If the reference grammar derives Lisp sexpressions from nonterminal A, then the parsing assertion in the expression

$$\mathtt{eval}(e \sqsubset A)$$

informs the analysis that all string values of e must be derivable from A, *i.e.*, they must be sexpressions.

4.2 Searching for Solutions

Finding assignments that solve containment constraints is simple, but we still have to find assignments that solve inclusion constraints. A brute force approach to solving $C = r \mathrel{\dot\subseteq} \varphi$ would be to enumerate all possible assignments of nonterminals to the language variables in C and then verify that $\Theta(\varphi) \stackrel{*}{\Rightarrow} \Theta(r)$. It turns out that the derivation relation $\stackrel{*}{\Rightarrow}$ is a decidable relation on $(N \cup T)^*$ for any context-free language as we will argue in the next section[1]. Surprisingly, a generalization of Earley's algorithm for parsing context-free languages [2] is applicable to deciding $\stackrel{*}{\Rightarrow}$ on $(N \cup T)^*$ for arbitrary G and is even amenable to improve on the brute force enumeration of all assignments. The next section recalls the essentials of that algorithm and generalizes it to perform verification of such an assignment.

5. EARLEY'S PARSING ALGORITHM

Earley's algorithm solves the word problem "$a_1 \ldots a_n \in L(G)$?" for a context-free grammar G. The algorithm works with an arbitrary context-free grammar and runs in time $O(n^3)$ in the worst case.

This section first presents the original algorithm and then its adaptation to words over $(N \cup T)$. By convention we let $A, B, \ldots$ range over nonterminals N, $\alpha, \beta, \ldots$ range over words of grammar symbols $(N \cup T)^*$, and X, X_i range over grammar symbols $(N \cup T)$. We write $A \to \alpha$ for a production of G.

5.1 Original Algorithm

Earley's algorithm [2] assigns a set E_i of Earley items to each position $i \in \{0, \ldots, n\}$ of an input word $a_1 \ldots a_n$ where 0 denotes the position right in front of the first symbol. An Earley item (without lookahead) has the form $[A \to \alpha \bullet \beta, j]$ where $A \to \alpha\beta$ is a production and $0 \le j \le n$. The dot corresponds to the current position in the input word, whereas the index j denotes the position in the input where matching of the right hand side of the production started. The item states that the word between position j and the current position is derivable from α. To avoid introducing a new separate start symbol, degenerate items of the form $[\to \alpha \bullet \beta, j]$ are also used. The sets E_i are the smallest sets closed under application of the rules in Fig. 10, for all $0 \le i \le n$. The input word is accepted if $[\to S\bullet, 0] \in E_n$.

5.2 Adaption to Sentential Forms

Every string of terminal and nonterminal symbols derivable from a context-free grammar's start symbol is a *sentential form*. Since the set of all sentential forms is also a context-free language (just extend the set of terminal symbols by copies A' of all nonterminal symbols A and add productions $A \to A'$), Earley's algorithm may also be used to recognize the language of sentential forms of G.

Instead of constructing the modified grammar explicitly and then running the algorithm on the modified grammar, the algorithm can be modified to directly recognize the language of sentential forms. In fact, the modified algorithm will be able to decide if $\gamma \stackrel{*}{\Rightarrow}_G \delta$ where $\gamma, \delta \in (N \cup T)^*$.

The only difference with respect to the standard formulation of Earley's algorithm is that nonterminal symbols may be scanned (shifted), too.

To solve queries of the form "$\gamma \stackrel{*}{\Rightarrow}_G X_1 \ldots X_n$?", Earley's rules generalize as shown in Fig. 11. The resulting algorithm is correct in the following sense.

Lemma 6 *$\gamma \stackrel{*}{\Rightarrow}_G X_1 \ldots X_n$ if and only if $[\to \gamma\bullet, 0] \in E_n$.*

Lemma 7 *The generalized Earley algorithm terminates after $O(n^3)$ steps.*

Proof. Analogous to Earley's proof [2]. □

The generalized Earley algorithm can verify if a nonterminal assignment Θ solves a constraint $r \mathrel{\dot\subseteq} \varphi$ by parsing the string $\Theta(r) \in (N \cup T)^*$ with the initialization $[\to \bullet\Theta(\varphi), 0] \in E_0$.

Definition 3 *We write $\Theta \vdash_G r \mathrel{\dot\subseteq} \varphi$ if parsing is successful, that is, if $fv(r \mathrel{\dot\subseteq} \varphi) \subseteq dom(\Theta)$ and $\Theta(\varphi) \stackrel{*}{\Rightarrow}_G \Theta(r)$.*

Example 3 *Consider the constraint $\mathtt{a} \cdot \varphi \mathrel{\dot\subseteq} \varphi'$ from Example 2 and the grammar G given by the productions $S \to \varepsilon \mid$*

[1] Multi-step derivability on $(N \cup T)^*$ is also decidable for context-sensitive languages, but the complexity is overwhelming.

init		$[\rightarrow \bullet S, 0] \in E_0$.
scan	$[A \rightarrow \alpha \bullet a\beta, j] \in E_i$ and $a_{i+1} = a$ implies	$[A \rightarrow \alpha a \bullet \beta, j] \in E_{i+1}$.
pred	$[A \rightarrow \alpha \bullet B\beta, j] \in E_i$ and $B \rightarrow \gamma \in P$ implies	$[B \rightarrow \bullet\gamma, i] \in E_i$.
red	$[B \rightarrow \gamma\bullet, j] \in E_i$ and $[A \rightarrow \alpha \bullet B\beta, k] \in E_j$ implies	$[A \rightarrow \alpha B \bullet \beta, k] \in E_i$.

Figure 10: Earley's algorithm

init		$[\rightarrow \bullet\gamma, 0] \in E_0$.
scan	$[A \rightarrow \alpha \bullet X\beta, j] \in E_i$ and $X_{i+1} = X$ implies	$[A \rightarrow \alpha X \bullet \beta, j] \in E_{i+1}$.
pred	$[A \rightarrow \alpha \bullet B\beta, j] \in E_i$ and $B \rightarrow \gamma \in P$ implies	$[B \rightarrow \bullet\gamma, i] \in E_i$.
red	$[B \rightarrow \gamma\bullet, j] \in E_i$ and $[A \rightarrow \alpha \bullet B\beta, k] \in E_j$ implies	$[A \rightarrow \alpha B \bullet \beta, k] \in E_i$.

Figure 11: Generalized Earley algorithm

AS and $A \rightarrow \mathtt{a} \mid AA$. *There are three successful parses for* $\Theta \vdash_G \mathtt{a} \cdot \varphi \mathrel{\dot\subseteq} \varphi'$*:*

$$\begin{array}{lcl} \Theta_1 & = & [\varphi \mapsto A, \varphi' \mapsto A] \\ \Theta_2 & = & [\varphi \mapsto A, \varphi' \mapsto S] \\ \Theta_3 & = & [\varphi \mapsto S, \varphi' \mapsto S] \end{array}$$

Since subtyping is defined structurally in terms of the language inclusion relation, the algorithm yields a decidable sufficient condition for testing the subtyping relation relative to a given nonterminal assignment. It remains to find suitable nonterminal assignments.

6. ASSIGNMENT CONSTRAINTS

Solving an inclusion constraint is essentially a matter of searching for suitable nonterminal assignments such that the constraint can be verified by parsing. For each inclusion constraint, there may be multiple nonterminal assignments such that the constraint is parsable. Hence, we propose to replace a solved inclusion constraint by an assignment constraint that collects all those assignments that lead to a successful parse.

Furthermore, we are only interested in solving those constraints, which are subject to a parsing assertion. These constraints can be extracted from a normal form constraint of the form $\varphi_1 \mathrel{\dot\subseteq} A_1 \wedge \cdots \wedge \varphi_n \mathrel{\dot\subseteq} A_n \wedge C$ by keeping those inclusions in C that only involve language variables $\varphi' \in \mathit{reach}(C, \varphi_1 \ldots \varphi_n)$ such that either $\mathit{reach}(C, \varphi') = \emptyset$ or $\varphi' \in \mathit{reach}(C, \varphi')$.

Then we consider each of the inclusions in turn. For each of them, there may be many nonterminal assignments Θ that make the constraint true. Fortunately, the number of nonterminal assignments is finite because the constraint contains a finite number of language variables and because there are finitely many nonterminal symbols.

The algorithm enumerates all nonterminal assignments, checks each of them if it leads to a successful parse using the algorithm from Section 5.2, and collects the successful ones in an *assignment constraint*, a disjunction of assignments $(\Theta_1 \vee \cdots \vee \Theta_k)$. By construction, all assignments in a single disjunction have the same domain. The definition of a solution extends to assignment constraints by

$$\frac{(\exists i)\ \Theta_i \subseteq \Theta}{S, \Theta \models_G (\Theta_1 \vee \cdots \vee \Theta_k)}$$

Formally, the inclusion $r \mathrel{\dot\subseteq} \varphi$ is replaced by $(\Theta_{i_1} \vee \cdots \vee \Theta_{i_k})$ if

- $\Phi_0 = \mathit{fv}(r \mathrel{\dot\subseteq} \varphi)$ is the set of language variables occurring in the inclusion constraint;
- $\Theta_1, \ldots, \Theta_n$ are *all possible* nonterminal assignments with $\mathit{dom}(\Theta_i) = \Phi_0$; and
- $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ such that

$$i \in I \Leftrightarrow \Theta_i \vdash_G r \mathrel{\dot\subseteq} \varphi$$

Lemma 8 *Under the listed conditions* $S, \Theta \models_G r \mathrel{\dot\subseteq} \varphi$ *if and only if* $S, \Theta \models_G (\Theta_{i_1} \vee \cdots \vee \Theta_{i_k})$.

Analogously, the containment $\varphi \mathrel{\dot\subseteq} A$ is replaced by $(\Theta_{i_1} \vee \cdots \vee \Theta_{i_k})$ if

- $\Phi_0 = \mathit{fv}(r \mathrel{\dot\subseteq} A)$ is the set of language variables occurring in the containment constraint;
- $\Theta_1, \ldots, \Theta_n$ are *all possible* nonterminal assignments with $\mathit{dom}(\Theta_i) = \Phi_0$; and
- $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ such that

$$i \in I \Leftrightarrow A \stackrel{*}{\Rightarrow}_G \Theta_i(\varphi).$$

Lemma 9 *Under the listed conditions* $S, \Theta \models_G \varphi \mathrel{\dot\subseteq} A$ *if and only if* $S, \Theta \models_G (\Theta_{i_1} \vee \cdots \vee \Theta_{i_k})$.

Simply turning $\varphi \mathrel{\dot\subseteq} A$ into an assignment $[\varphi \mapsto A]$ is incomplete. To see this, consider the productions $A \rightarrow B, B \rightarrow b, C \rightarrow B$ and the constraint $\varphi \mathrel{\dot\subseteq} A \wedge \varphi \mathrel{\dot\subseteq} \varphi'$. Assigning A to φ prevents the assignment of C to φ', thus losing solutions.

Example 4 *Continuing Example 3 further, the constraint* $\mathtt{a} \cdot \varphi \mathrel{\dot\subseteq} \varphi'$ *would be replaced by an assignment constraint representing the three possible solutions* Θ_1*,* Θ_2*, and* Θ_3*. That is*

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee [\varphi \mapsto S, \varphi' \mapsto S])$$

Exhaustive application of the enumeration algorithm to all inclusions and containments results in a conjunction of assignment constraints. Reduction of such a conjunction is by pairwise combination of assignment constraints:

$$(\Theta_1 \vee \cdots \vee \Theta_k) \wedge (\Theta'_1 \vee \cdots \vee \Theta'_l) \Leftrightarrow \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Theta_i \bowtie \Theta'_j$$

The join operator $\bowtie$ checks if a pair of assignments is compatible, that is, if the assignments agree on the intersection

of their domains. If the pair of assignments is compatible, then they are merged to form a new assignment. Otherwise, the pair is dropped using the rule $x \vee \mathsf{fail} = x$.

$$\Theta \bowtie \Theta' = \begin{cases} \Theta \cup \Theta' & \textbf{if} \quad (\forall \varphi \in dom(\Theta) \cap dom(\Theta')) \\ & \qquad \Theta(\varphi) = \Theta'(\varphi) \\ \mathsf{fail} & \textbf{otherwise} \end{cases}$$

The resulting constraint is either fail or it is a single assignment constraint enumerating all possible solutions.

The reduction of conjunctions is also sound and complete.

Lemma 10 *$S, \Theta \models_G (\Theta_1 \vee \cdots \vee \Theta_k) \wedge (\Theta'_1 \vee \cdots \vee \Theta'_l)$ if and only if $S, \Theta \models_G \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Theta_i \bowtie \Theta'_j$.*

Example 5 *Starting from the assignment constraint of the preceding example*

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee [\varphi \mapsto S, \varphi' \mapsto S])$$

we add a further assignment constraint, say, $([\varphi \mapsto A])$. The result of joining these two assignment constraints is

$$\begin{array}{l} ([\varphi \mapsto A, \varphi' \mapsto A] \bowtie [\varphi \mapsto A] \vee \\ [\varphi \mapsto A, \varphi' \mapsto S] \bowtie [\varphi \mapsto A] \vee \\ [\varphi \mapsto S, \varphi' \mapsto S] \bowtie [\varphi \mapsto A]) \end{array}$$

which simplifies to

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee \mathit{fail})$$

and to

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S])$$

Since the above constraint reduction rules may be applied exhaustively to transform all inclusion and containment constraints into a single assignment constraint, there is an algorithm to solve a constraint C in normal form. Together with the prior result that constraint normalization transforms a constraint to normal form, we have the following.

Lemma 11 *There is an algorithm that decides for given C and S if there is a Θ such that $S, \Theta \models_G C$.*

There are a few obvious ways to improve the performance of the naive algorithm. One of them is to associate with each language variable a set of nonterminals which have led to successful parses and thus are assigned to the language variable in an assignment constraint. Another, more efficient way is to further extend the parsing algorithm to infer the equivalent assignment constraint while constructing the Earley sets. The next section explores this possibility.

7. PARSING WITH UNKNOWNS

The final generalization of Earley's algorithm yields an algorithm suitable for parsing strings r which may contain language variables. The algorithm thus interleaves the construction of an assignment constraint with parsing.

The resulting procedure (shown in Fig. 12) maintains the set of states E_i^Θ where $0 \leq i \leq |r|$ and Θ is a partial assignment with $dom(\Theta) \subseteq fv(r)$. In the definition, X ranges over $\Phi \cup N \cup T$, $X_1 \ldots X_n$ is the subject string, and Θ is assumed to be the identity function on terminal and nonterminal symbols. As usual, the algorithm amounts to exhaustive application of the rules. Again, the algorithm terminates after a finite number of steps because the number of state sets

init $[\rightarrow \bullet\varphi, 0] \in E_0^\emptyset$.

scan $[A \rightarrow \alpha \bullet X\beta, j] \in E_i^\Theta$ and $\Theta(X) = \Theta(X_{i+1})$
implies $[A \rightarrow \alpha X \bullet \beta, j] \in E_{i+1}^\Theta$.

inst-n $[A \rightarrow \alpha \bullet \varphi\beta, j] \in E_i^\Theta$ and $\varphi \notin dom(\Theta)$ and $\Theta(X_{i+1}) = B$
implies $[A \rightarrow \alpha \bullet \varphi\beta, j] \in E_i^{\Theta[\varphi \mapsto B]}$.

inst-n $[A \rightarrow \alpha \bullet X\beta, j] \in E_i^\Theta$ and $X_{i+1} = \varphi \notin dom(\Theta)$ and $\Theta(X) = B$
implies $[A \rightarrow \alpha \bullet X\beta, j] \in E_i^{\Theta[\varphi \mapsto B]}$.

inst-n $[A \rightarrow \alpha \bullet \varphi'\beta, j] \in E_i^\Theta$ and $X_{i+1} = \varphi$ and $\varphi, \varphi' \notin dom(\Theta)$ and $B \in N$
implies $[A \rightarrow \alpha \bullet \varphi'\beta, j] \in E_i^{\Theta[\varphi \mapsto B]}$.

inst-t $[A \rightarrow \alpha \bullet \varphi\beta, j] \in E_i^\Theta$ and $X_{i+1} = a \in T$ and ($a \in$ *first*(B) or B nullable) and $\varphi \notin dom(\Theta)$
implies $[A \rightarrow \alpha \bullet \varphi\beta, j] \in E_i^{\Theta[\varphi \mapsto B]}$.

pred $[A \rightarrow \alpha \bullet X\beta, j] \in E_i^\Theta$ and $\Theta(X) = B \in N$ and $B \rightarrow \gamma \in P$
implies $[B \rightarrow \bullet\gamma, i] \in E_i^\Theta$.

red $[B \rightarrow \gamma\bullet, j] \in E_i^\Theta$ and $[A \rightarrow \alpha \bullet X\beta, k] \in E_j^{\Theta'}$ and $\Theta'(X) = B$ and $\Theta' \subseteq \Theta$
implies $[A \rightarrow \alpha X \bullet \beta, k] \in E_i^\Theta$.

Figure 12: Earley parsing for strings with language variables

is finite (since Θ ranges over a finite set of assignments) and the number of elements in each set is finite by the same argumentation as before.

Each successful parse for $X_1 \ldots X_n \subseteq \varphi$ leads to a state E_n^Θ containing $[\rightarrow \varphi\bullet, 0]$. Hence, the algorithm is sound and complete with respect to parses of the grammar G.

Lemma 12 *The algorithm in Fig. 12 produces a state E_n^Θ containing $[\rightarrow \varphi\bullet, 0]$ if and only if $\Theta \vdash_G X_1 \ldots X_n \dot{\subseteq} \varphi$.*

8. EXAMPLES AND DISCUSSION

We have implemented the algorithm in Fig. 12 to evaluate if it leads to practically useful results. In this section, we examine its results using two example grammars. The first grammar defines a tokenized fragment of XHTML, while the second one is more realistic in defining Lisp sexpressions down to the single character.

8.1 Example: Tokenized XHTML

As an example, we have used the grammar in Fig. 13, which defines a fragment of XHTML. In the grammar, the `typewriter` font indicates terminal symbols and upper case letters are used for nonterminal symbols.

Here is typical function to append to the body of an enumerated list

$$f_1 = \lambda x.(\texttt{ CDATA }) \cdot x$$

As in our earlier example, the constrained type of the function is

$$(\texttt{ CDATA }) \cdot \varphi \dot{\subseteq} \varphi' \Rightarrow \texttt{Str}(\varphi) \rightarrow \texttt{Str}(\varphi')$$

$S \to$ `<html>` $H\ B$ `</html>`
$S \to H\ B$
$H \to$ `<head>` T `</head>`
$T \to$ `<title> CDATA </title>`
$B \to$ `<body>` C `</body>`
$L \to$ `<li>` C `</li>`
$L \to L\ L$
$C \to$
$C \to$ `CDATA`
$C \to C\ C$
$C \to$ `<p>` C `</p>`
$C \to$ `<b>` C `</b>`
$C \to$ `<tt>` C `</tt>`
$C \to$ `<em>` C `</em>`
$C \to$ `<i>` C `</i>`
$C \to$ `<u>` C `</u>`
$C \to$ `<s>` C `</s>`
$C \to$ `<ol>` L `</ol>`
$C \to$ `<ul>` L `</ul>`
$C \to$ `<dl>` D `</dl>`
$D \to$ `<dt>` C `</dt>`
$D \to$ `<dd>` C `</dd>`
$D \to D\ D$

Figure 13: Grammar for XHTML fragment

Solving this constraint yields only one assignment: $[\varphi \mapsto L, \varphi' \mapsto L]$ so that the constrained type of f_1 with respect to the XHTML grammar is

$$[\varphi \mapsto L, \varphi' \mapsto L] \Rightarrow \texttt{Str}(\varphi) \to \texttt{Str}(\varphi')$$

Since the nonterminal L constructs list bodies, it (and the function f_1) must be used in a string context which wraps `<ol>` or `<ul>` around it.

The slightly more general function

$$f_2 = \lambda x.\lambda y.\texttt{<dd>} \cdot y \cdot \texttt{</dd>} \cdot x$$

has the constrained type

$$(\texttt{<dd>} \cdot \varphi' \cdot \texttt{</dd>} \cdot \varphi \dot{\subseteq} \varphi'') \Rightarrow \texttt{Str}(\varphi) \to \texttt{Str}(\varphi') \to \texttt{Str}(\varphi'')$$

Solving the constraint also yields only one assignment: $[\varphi \mapsto D, \varphi' \mapsto C, \varphi'' \mapsto D]$ so that f_2's constrained type is

$$[\varphi \mapsto D, \varphi' \mapsto C, \varphi'' \mapsto D] \Rightarrow \texttt{Str}(\varphi) \to \texttt{Str}(\varphi') \to \texttt{Str}(\varphi'')$$

Hence, the variable $x : \texttt{Str}(\varphi)$ must contain a string derivable from D, the contents of a definition list, the variable $y : \texttt{Str}(\varphi')$ must contain mixed content, and the result of the function is again a string derivable from D, suitable for the contents of a definition list.

In present example, it does not matter for the final assignment constraint whether the functions construct their result in a left- or right-recursive manner. Both f_1 and f_2 are right-recursive but their left-recursive counterparts (*e.g.*, $f_1' = \lambda x.x \cdot (\texttt{ CDATA })$ for the first function) yield identical assignment constraints. The constrained types before solving the inclusion constraints are different, of course, as can be seen for f_1':

$$\varphi \cdot (\texttt{ CDATA }) \dot{\subseteq} \varphi' \Rightarrow \texttt{Str}(\varphi) \to \texttt{Str}(\varphi')$$

The constrained type *after* solving inclusions turns out to be the same as before:

$$[\varphi \mapsto L, \varphi' \mapsto L] \Rightarrow \texttt{Str}(\varphi) \to \texttt{Str}(\varphi')$$

However, the indifference with respect to left- and right-recursion is due to our careful crafting of the list constructing rules in the grammar. If the grammar were left- or right-biased like the typical input for an LL- or LALR-parser generator then only the corresponding version of the function would lead to an assignment, the other version would be rejected.

8.2 Scannerless Parsing of Sexpressions

Since the analyzed program generates each string character by character, the reference grammar must be defined in terms of single characters, too. It is not possible to parse in terms of tokens because there is no lexical analysis. Hence,

SEXP → *B0 AEXP B0*
SEXP → *B0 PEXP B0*
AEXP → `a`
PEXP → `(` *BSERIES* `)`
BSERIES → *B0*
BSERIES → *B0 SERIES B0*
SERIES → *SERIES_YY B1 SERIES_YY*
SERIES → *SERIES_YN B0 SERIES_NY*
SERIES_YY → *AEXP*
SERIES_YY → *PEXP*
SERIES_YY → *SERIES_YY B1 SERIES_YY*
SERIES_YY → *SERIES_YN B0 SERIES_NY*
SERIES_YN → *PEXP*
SERIES_YN → *SERIES_YY B1 SERIES_YN*
SERIES_YN → *SERIES_YN B0 SERIES_NN*
SERIES_NY → *PEXP*
SERIES_NY → *SERIES_NY B1 SERIES_YY*
SERIES_NY → *SERIES_NN B0 SERIES_NY*
SERIES_NN → *PEXP*
SERIES_NN → *SERIES_NY B1 SERIES_YN*
SERIES_NN → *SERIES_NN B0 SERIES_NN*
B0 →
B0 → *B1*
B1 → ␣
B1 → *B1 B1*

Figure 14: Scannerless grammar for some sexpressions

a grammar for scannerless parsing [10] is required for performing string analysis.

Unfortunately, a scannerless grammar obscures some of the structure because it must be explicit with issues like the handling of whitespace, the maximum munch rule, and so on. In an experiment with a several scannerless grammars for Lisp-sexpressions, we found that some effort was required to get the grammar into a form suitable for dealing with left- and right-recursive constraints. Figure 14 shows an example grammar. The grammar still leaves some room for improvement because it insists on putting more spaces than strictly necessary. With the example grammar, we can verify that the constraints (using ␣ to indicate a space character)

$$\varphi.\texttt{␣␣(a)} \quad \dot{\subseteq} \quad \varphi \tag{1}$$
$$\texttt{(a)␣␣.}\varphi \quad \dot{\subseteq} \quad \varphi \tag{2}$$
$$\varphi.\texttt{(a)} \quad \dot{\subseteq} \quad \varphi \tag{3}$$
$$\texttt{(a).}\varphi \quad \dot{\subseteq} \quad \varphi \tag{4}$$

have the following four solutions

- $\varphi \mapsto SERIES\_NN$: constraints (1), (2), (3), and (4)
- $\varphi \mapsto SERIES\_YN$: constraints (1), (2), and (3)
- $\varphi \mapsto SERIES\_NY$: constraints (1), (2), and (4)
- $\varphi \mapsto SERIES\_YY$: constraints (1) and (2)

9. RELATED WORK

Using formal languages to define refined type systems has received much attention stimulated by the search for expressive type systems for XML processing. The most influential results in this area are by Hosoya and Pierce [6, 5]. They propose to integrate regular expressions in a type language for describing the structure of XML documents. This design yields an expressive type system with a decidable, semantic subtyping relation. A number of subsequent papers by different authors extends their work in various directions.

Regular expression types have been applied to strings by Tabuchi et al [9]. In their work, the index of a string type and the description of the string output are both regular expressions. Compared to our system, that work has two additional features and two restrictions. First, it abstracts the output of a computation by a regular expression. Second, it attempts to derive tight bounds for substrings extracted via regular pattern matching. While our system could be easily extended with output tracking, it is not clear how to integrate regular pattern matching. On the downside, the system by Tabuchi et al does not come with a type inference algorithm and it is not clear if such an algorithm exists whereas our system has an inference algorithm that computes principal types. Finally, regular expression types describe string values with a regular language whereas our system can assert containment of a string value in a context-free language.

Another approach to analyzing string values is by Christensen et al [1]. From a Java program, they extract a context-free grammar $\mathcal{G}$ such that each string expression is described by one nonterminal symbol. They apply an algorithm from natural language processing to compute a regular grammar $\mathcal{R}$ such that for each nonterminal N in $\mathcal{G}$ there is a nonterminal N' in $\mathcal{R}$ such that $L_{\mathcal{G}}(N) \subseteq L_{\mathcal{R}}(N')$. That is, the grammar $\mathcal{R}$ is an overapproximation of $\mathcal{G}$.

Their framework can assert that the value of a string expression is always contained in a prescribed *regular language* (since containment of regular languages is decidable). While such a check is sufficient for validating URL syntax, attribute values, and many uses of reflection (as indicated in sections 8.2 and 8.3 of their paper), a check against a regular language is not sufficient for performing syntax analysis (validating expressions, asserting well-formedness of XML strings). In contrast, our type system guarantees that the value of a string expression is contained in a prescribed *context-free language*. Hence, it is suitable for syntax analysis tasks.

Finally, their system is defined and implemented to provide sensible abstractions for all of Java's string manipulation, whereas the scope of our system is much more modest, an applied lambda calculus with only the essential operations for string construction.

Earley's algorithm [2] has been the basis for many extended parsing algorithms. It yields an efficient parsing scheme without requiring a transformation on the underlying context-free grammar. The algorithm has recently received growing interest for parsing ambiguous grammars, for example, in the context of program renovation.

10. CONCLUSION

We have outlined a path to obtaining sound and precise solutions for analyzing the set of values of a string expression. Our approach relies on a generalization of Earley's parsing algorithm to decide inclusion constraints with language variables. The main novelty of our work is the guarantee that the value of a string expression is an element of a given context-free language.

It is not our intention to advocate using the language and analysis in this paper to write programs that produce only well-formed strings. Domain specific abstract datatypes with suitable APIs are much better suited for such a task. Rather, we see the present work as a starting point to investigate powerful analyses for existing programs in widely disseminated scripting language to gain confidence that those programs only feed well-formed strings to (say) SQL processors.

Initial experiments with an implementation of the solver for inclusion constraints have yielded encouraging results. To obtain results for real programs, we are currently experimenting with different approaches to analyzing string matching constructs and with generalizations to the notion of a nonterminal assignment. We are also looking at automatic ways of transforming token-based grammars so that they are no longer biased towards left- or right-recursion and to make them suitable for scannerless parsing. Once these problems have satisfactory solutions, we intend to construct a full implementation of the type inference algorithm outlined in the paper.

Acknowledgment

Thanks to the anonymous reviewers for their extensive comments that helped to improve the presentation significantly.

11. REFERENCES

[1] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Radhia Cousot, editor, *Proc. International Static Analysis Symposium, SAS'03*, number 2694 in Lecture Notes in Computer Science, pages 1–18, San Diego, CA, USA, June 2003. Springer-Verlag.

[2] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[3] Thom Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science. Springer, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).

[4] John E. Hopcroft. On the equivalence and containment problems for context-free languages. *Math. Systems Theory*, 3(2):119–124, 1969.

[5] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In Hanne Riis Nielson, editor, *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 67–80, London, England, January 2001. ACM Press.

[6] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In Philip Wadler, editor, *Proc. International Conference on Functional Programming 2000*, pages 11–22, Montreal, Canada, September 2000. ACM Press, New York.

[7] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Kluwer Academic Publishers, Dordrecht, 2001.

[8] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[9] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In Gilles Barthe and Peter Thiemann, editors, *Proceedings of Workshop on Types in Programming (TIP02)*, volume 75 of *ENTCS*, pages 1–19, 2003.

[10] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[11] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC(TM) API Tutorial and Reference: Universal Data Access for the Java(TM) 2 Platform*. Addison-Wesley, 1999.

[12] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.