Type-Based Specialization of XML Transformations

Kazutaka Matsuda
The University of Tokyo/JSPS Research Fellow
kztk@ipl.t.u-tokyo.ac.jp

Zhenjiang Hu
National Institute of Informatics
hu@nii.ac.jp

Masato Takeichi
The University of Tokyo
takeichi@mist.i.u-tokyo.ac.jp

Abstract

It is often convenient to write a function and apply it to a specific input. However, a program developed in this way may be inefficient to evaluate and difficult to analyze due to its generality. In this paper, we propose a technique of new specialization for a class of XML transformations, in which no output of a function can be decomposed or traversed. Our specialization is type-based in the sense that it uses the structures of input types; types are described by regular hedge grammars and subtyping is defined set-theoretically. The specialization always terminates, resulting in a program where every function is fully specialized and only accepts its rigid input. We present several interesting applications of our new specialization, especially for injectivity analysis.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

1. Introduction

It is often convenient to write a function and apply it to a specific input. For example, a user may apply function *half* defined by $\mathit{half}(x) = \lfloor x/2 \rfloor$ to even numbers, or more generally, apply function $f :: A \to B$ to an input of type A' where A' is a subtype of A.

We focus on subtyping in XML transformations (functions). For example, consider an input document of a type where an element `<person>` must contain exactly one occurrence of `<email>`, `<tel>`, `<studentID>`, and `<postal_address>` in any order. Writing a function that exactly accepts such data is cumbersome because it would consist of twenty-four branches, i.e., the number of permutations of four elements $4! = 24$. One, however, can easily define the function as

$$\begin{array}{ll}
\mathit{transformPerson}(\texttt{<person>}(x)) & \hat{=}\ \mathit{foreachF}(x) \\
\mathit{foreachF}(\varepsilon) & \hat{=}\ \varepsilon \\
\mathit{foreachF}(x :: \mathit{Elem} \mathbin{.} r :: \mathit{Elem}^*) & \hat{=}\ f(x) \mathbin{.} \mathit{foreachF}(r) \\
f(\texttt{<email>}(x)) & \hat{=}\ \ldots \\
f(\texttt{<tel>}(x)) & \hat{=}\ \ldots \\
f(\texttt{<studentID>}(x)) & \hat{=}\ \ldots \\
f(\texttt{<postal\_address>}(x)) & \hat{=}\ \ldots
\end{array}$$

if the order of the four does not count. Here, we use ε to represent the empty sequence of XML elements, A^* for the Kleene closure of A, *Elem* for a type corresponding to any XML element, `<t>`(x) for `<t>`x`</t>`, and the operator "**.**" to concatenate two sequences. Function *transformPerson* above accepts the input of the type where `<person>` contains `<email>`, `<tel>`, `<studentID>`, and `<postal_address>` without any restrictions. The input type of *transformPerson* is a supertype of the type of input to which a user wants to apply the function. That is, since the application of *transformPerson* to the type of input uses subtyping, the application is only valid with subtyping in the sense of type correctness. Many modern XML transformation languages such as XDuce [16], CDuce [1], and Xtatic [18] support subtyping. Also, writing a function like that above supported by subtyping sometimes considered to be advantageous because a function defined in this way is reusable and robust to changes in input type, i.e., changes in the schema of an input XML document.

Despite the convenience of writing flexible programs with subtyping, a program defined in this way might be inefficient to evaluate and difficult to analyze.

First, a specific function can be implemented more efficiently than a general function. Consider the following function.

$$\begin{array}{ll}
f(\varepsilon) & \hat{=}\ \varepsilon \\
f(\texttt{<a>}(x :: \mathit{String}) \mathbin{.} r) & \hat{=}\ f(r) \mathbin{.} \texttt{<a>}(x) \\
f(r \mathbin{.} \texttt{<b>}(y)) & \hat{=}\ \ldots
\end{array}$$

Function f reverses an input when it only consists of `<a>` elements, and does something when it ends in `<b>`. To directly evaluate f, we must examine whether an input is a sequence of `<a>`s or is a sequence that ends in `<b>`, which takes $\mathrm{O}(n)$ time for each recursive call of f where n is the length of the input sequence. However, when a set of inputs of f is restricted to sequences of `<a>`s, we do not need to examine the whole input sequence for each recursion; it is sufficient to check whether the input list is empty or not, which takes a constant time. Actually, CDuce employs such a type-based technique of optimization [10]; if the system can establish that the inputs of f are restricted to sequences of `<a>`s, the restricted version of f above runs in $\mathrm{O}(n)$ time in CDuce, where n is the length of the input sequence of `<a>`s. However, as we do not know the inputs that have been restricted to sequences of `<a>`s by f itself for f above, we cannot enjoy this kind of type-based optimization.

Second, in program analysis, we want to analyze the behavior of a general function restricted to specific inputs but not the general function itself. For example, consider injectivity analysis, which is important in program inversion [9, 12, 13, 14], program bidirectionalization [24], and the validation of bidirectional transformation [4]. However, sometimes the injectivity of $f|_{A'}$, which reads the function f whose domain is restricted to A', is different from that of $f : A \to B$. For example, consider the following function.

$$\begin{array}{ll}
\mathit{unifyAddress}(\texttt{<email>}(x)) & \hat{=}\ \texttt{<address>}(x) \\
\mathit{unifyAddress}(\texttt{<tel>}(x)) & \hat{=}\ \texttt{<address>}(x)
\end{array}$$

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-327-3/09/01…$5.00

Transformation $\mathit{unifyAddress}$ produces an `<address>` element from either an `<email>` element that describes an email address or a `<tel>` element that describes a telephone number. In many situations, a set of strings describing email addresses is disjoint from a set of strings describing telephone numbers. Consequently, restricted to type T defined by

$$T = \texttt{<email>}(\mathit{EmalText}) \mid \texttt{<tag>}(\mathit{TelNumber})$$

where $\mathit{EmailText}$ and $\mathit{TelNumber}$ are disjoint, $\mathit{unifyAddress}|_T$ becomes injective. Another example is precise type inference. For function $f : A \to B$ and type $A' \subseteq A$, the range of function $f|_{A'}$ would be smaller than B. For example, the output type of $\mathit{unifyAddress}$ is $\texttt{<address>}(\top)$ where $\top$ represents anything, while the output type of $\mathit{unifyAddress}|_T$ is $\texttt{<address>}(\mathit{EmailText} \mid \mathit{TelNumber})$. Precise type inference is not only useful for precise type checking [23], but also for program inversion [9, 13, 14] in the derivation of deterministic inverses.

We rephrase the above problems with subtyping as follows. For general function $f :: A \to B$ and a specific input of type A' where $A' \subseteq A$, although we want to analyze $f|_{A'}$, the only information available from a program is that of f and A' but not that of $f|_{A'}$.

To solve the problem, we propose a new technique of specialization for a class of XML transformation languages, in which no output of a function can be examined by a pattern. The specialization generates function definitions of $f|_A$ from function f and type A. For example, for function $\mathit{unifyAddress}$ and type T above, our specialization generates the following.

$$\begin{array}{lcl} \mathit{unifyAddress}|_T(\texttt{<email>}(x :: \mathit{EmailText})) & \hat{=} & \texttt{<address>}(x) \\ \mathit{unifyAddress}|_T(\texttt{<tel>}(x :: \mathit{TelNumber})) & \hat{=} & \texttt{<address>}(x) \end{array}$$

The method of specialization we propose is type-based in the sense that the specialization uses the structures of types of specific inputs, i.e., a type is described by a regular hedge grammar [6] and subtyping is defined set-theoretically. Types defined by regular hedge grammar are commonly used in modern XML processing languages such as XDuce, CDuce, and Xtatic, and a set of valid XML documents/elements described by core parts of many XML schema languages such as DTD[1], XML Schema[2], and Relax NG[3] can be expressed by the type [28]. The three main contributions made by this paper can be summarized as follows.

- *Type-based specialization that generates $f|_T$ from f and T is proposed.* The type-based specialization we propose converts a program to that in which every function, including a recursively-defined function, is fully specialized and only accepts its rigid input. After specialization, we can safely ignore the types of arguments of a function call in program analysis because the called function is already specialized with respect to the types.
- *Our specialization always terminates.* Despite the strong property that the specialization fully specializes every function, the specialization always terminates. In addition, our specialization runs without human interaction.
- *Several applications demonstrate the effectiveness of our specialization.* We demonstrate the effectiveness of our new specialization by applying it to several examples, including type inference, injectivity analysis, and inversion. The property that the specialization fully specializes every function plays an important role in these applications.

[1] `http://www.w3.org/TR/REC-xml/#dt-doctype`

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

[3] `http://relaxng.org/`

We have also implemented a prototype system of our proposed specialization, which is available at `http://www.ipl.t.u-tokyo.ac.jp/~kztk/sp/`.

This paper is organized as follows. Section 2 roughly explains the idea behind our type-based specialization. Section 3 describes the language used to describe XML transformations. Section 4 describes the new method of type-based specialization for the language in Section 3, and gives a proof for termination. Section 5 presents several interesting applications of the specialization. Section 6 discusses extensions of the proposed approach. Section 7 concludes the paper and discusses future directions.

Related Work

Our specialization is a modest extension of the pre-processing method used in the exact type checking for a tree transduction [23]. The main differences between our method and theirs are as follows: (1) we permit concatenations in patterns, e.g., $x :: \texttt{<a>}^* \,\textbf{.}\, y :: \texttt{<b>}$, and (2) our specialization returns deterministic programs.

In specialization, we use a technique similar to type inference of a variable in a pattern where the pattern is restricted to a type. Our type inference technique in specialization is more precise for the restricted target language than that of existing approaches [1, 15, 16]. The preciseness of type inference is important for the fully specialized results of the specialization.

Our target language discussed in Section 3 is a simple extension of Wadler's treeless and affine language [32], to which concatenation operator "**.**" have been added in both patterns and expressions. The extension is also inspired by tree transducers with look-ahead [7] and patterns in XDuce [16]. Actually, type T in variable pattern $x :: T$ can be viewed as "look-ahead" in treeless language, and a pattern in our target language is a subset of patterns in XDuce. The specialization we propose can be applied to a wider class of languages, e.g., allowing accumulation parameters as macro forest transducers [30], which will be explained in Section 6. It is worth noting that, with the extension in Section 6, the language to which the specialization can be applicable is strictly more expressive than deterministic macro forest transducers [30], i.e., a model of recursive XML transformations. Many XSLT[4] programs can be expressed by a composition of deterministic macro forest transducers [8, 25, 21]. A transformation defined by a composition of deterministic macro forest transducers can be written by a deterministic macro forest transducer if the output data size is always proportional to the input data size [20].

In the applications presented in Section 5, we use type inference algorithm in [23], a variant of injectivity checking used in XSugar [3, 4], and a naive inversion by swapping left-hand sides with right-hand sides. A formal discussion on the inversion by swapping left-hand sides with right-hand sides for treeless language is found in [29].

There has been research on type-based optimization for pattern-matching [10, 19]. Although they carry out type-based "specialization" as we do, there are many differences between their specialization and ours. First, our purpose is to derive the definitions of $f|_A$ from f and A while theirs is to provide efficient evaluation mechanisms of pattern-matching. Second, we specialize patterns and function calls but do nothing on pattern-matching, while they specialize pattern-matching mechanisms using compilations of patterns but not function calls. Third, in our specialization, for a recursively-defined function, a function call of the function may be specialized with respect to different types like $\mathit{reverse}$ in Section 2, while, in their specialization, the input type of a function is given beforehand. Note that patterns in their languages are more expressive than that in our target language.

[4] `http://www.w3.org/TR/xslt`

2. Idea of Specialization

This section roughly explains the idea behind our specialization and problems with its naive execution.

Our specialization approach, which will be discussed more in Section 4, targets a class of first-order functional programming languages. The target language is similar to ordinary functional programming languages such as Haskell [2] except that the target language contains variable pattern $x :: T$ where T is given by regular hedge grammar [6] and concatenation patterns/expressions "$\centerdot$", like the function $splitAB$[5] defined by

$$splitAB(x :: \texttt{<a>}^* \centerdot y :: \texttt{<b>}^*) \mathrel{\hat{=}} \texttt{<as>}(x) \centerdot \texttt{<bs>}(y).$$

Some syntactic restrictions have also been added to the language. One of the most important syntactic restrictions in our target language is that no output of a function can be examined by a pattern. The formal definition of the language will be given in Section 3.

2.1 Idea

The basic idea behind our specialization is quite simple: we produce new definition rules for function $f|_S$ where S is type of its argument. That is, for type S, function f defined by

$$f(\ldots x :: T \ldots) \mathrel{\hat{=}} \ldots g(x) \ldots$$

is specialized to the following $f|_S$.

$$f|_S(\ldots x :: T' \ldots) \mathrel{\hat{=}} \ldots g|_{T'}(x) \ldots$$

Here, T' is a calculated type from T so that the semantics of function f for the inputs in S is preserved (Section 4). For example, the specialization of the following *reverse*

$$\begin{array}{ll} reverse(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\ reverse(a :: Elem \centerdot r :: Elem^*) & \mathrel{\hat{=}} reverse(r) \centerdot a \end{array}$$

with respect to type $S = (\texttt{<a>} \centerdot \texttt{<b>})^*$ results in

$$\begin{array}{ll} reverse|_S(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\ reverse|_S(a :: \texttt{<a>} \centerdot r :: S') & \mathrel{\hat{=}} reverse|_{S'}(r) \centerdot a \\ reverse|_{S'}(a :: \texttt{<b>} \centerdot r :: S) & \mathrel{\hat{=}} reverse|_S(r) \centerdot a \end{array}$$

where S' is a type defined by $S' = \texttt{<b>} \centerdot S$.

This idea of type-based specialization itself is not new and is nothing but partial evaluation. A similar method is adopted for a similar purpose in [23]. One of the contributions in this paper is to discuss our construction of type-based specialization in which the properties below are guaranteed to hold.

Termination of Specialization

Termination is one of the most important properties of program transformation. However, the proof is not entirely direct for our target language. Recall that, when a specialization of f

$$f(\ldots x :: T \ldots) \mathrel{\hat{=}} \ldots g(x) \ldots$$

with respect to type S generates a rule

$$f|_S(\ldots x :: T' \ldots) \mathrel{\hat{=}} \ldots g|_{T'}(x) \ldots,$$

new type T' and new function call $g|_{T'}(x)$ are produced. Then, the specialization specializes function g with respect to T' to obtain the definition rules of $g|_{T'}$. The specialization process will finish when the rules of every called functions in generated rules have been generated. However, it is not clear whether the specialization terminates or infinitely produces new types and new definition rules of new functions. A pattern like $x :: U \centerdot y :: V$ makes the problem more difficult. Without this kind of pattern, we can easily give a proof using the technique of product-construction [6] of automata, similarly to [23].

[5] Here, <a> is shorthand for <a>(ε).

Determinism of Specialized Programs

Another important issue is that specialization generates deterministic programs. Nondeterminism of a program may make a program difficult to analyze. Specialization may produce more than one rule from a rule. For example, the specialization of g defined by

$$idAB(x :: (\texttt{<a>}|\texttt{<b>}) \centerdot y :: (\texttt{<a>}|\texttt{<b>})) \mathrel{\hat{=}} x \centerdot y$$

with respect to type $T = (\texttt{<a>} \centerdot \texttt{<b>}) \mid (\texttt{<b>} \centerdot \texttt{<a>})$ results in $g|_T$ defined as follows.

$$\begin{array}{l} idAB|_T(x :: \texttt{<a>} \centerdot y :: \texttt{<b>}) \mathrel{\hat{=}} x \centerdot y \\ idAB|_T(x :: \texttt{<b>} \centerdot y :: \texttt{<a>}) \mathrel{\hat{=}} x \centerdot y \end{array}$$

This production of more than one rule from a rule may result in a nondeterministic program.

3. Target Language

This section describes our target language to describe XML transformations.

3.1 Hedge Values

In our language, XML elements, XML documents, and strings are represented by *hedge values*, i.e., sequences of (unranked) trees. For example, an XML fragment as

```
<name>kztk</name><email>...</email>
```

is internally represented by

$$\texttt{<name>}(\texttt{k}() \centerdot \texttt{z}() \centerdot \texttt{t}() \centerdot \texttt{k}()) \centerdot \texttt{<email>}(\ldots)$$

where <name>, <email>, and k, z, t are *labels* to construct the hedge value. Formally, *hedges* are defined inductively from a set of labels Σ:

- Empty hedge ε is a hedge.
- For hedges h_1 and h_2, a concatenation $h_1 \centerdot h_2$ is a hedge.
- For label $\sigma \in \Sigma$ and hedge h, $\sigma(h)$ is a hedge.

We assume that all the tree-labels and characters are encoded to labels Σ. Note that hedge concatenation $\centerdot$ is associative and ε is the unit of hedge concatenation; $h_1 \centerdot (h_2 \centerdot h_3) = (h_1 \centerdot h_2) \centerdot h_3$ and $h \centerdot \varepsilon = \varepsilon \centerdot h = h$ hold. We sometimes write a tree $\sigma(\varepsilon)$ as $\sigma()$ or σ. For convenience, we sometimes omit $\centerdot$ and write $t_1 t_2$ instead of $t_1 \centerdot t_2$ if no confusion would arise. *Context* $\mathcal{C}$ is a hedge containing special hole variable $\square$; we write the hedge that is obtained from $\mathcal{C}$ by replacing hole $\square$ with hedge h as $\mathcal{C}[h]$.

XML Attributes or IDREFs are not treated in this paper.

3.2 Types

Set-theoretic types [11] defined by *regular hedge grammars* [6] are used in our language. As an example of regular hedge grammars, a set of hedges described by regular expression $A = \texttt{<a>}^* \mid \texttt{<b>}$ is represented by regular hedge grammar $A \to \texttt{<a>}A', A \to \texttt{<b>}E, A \to \varepsilon, A' \to \varepsilon, A' \to \texttt{<a>}A', E \to \varepsilon$. As examples given in the Introduction and Section 2, the types are used in patterns.

Definition 1 (Regular Hedge Grammar)**.** A *regular hedge grammar* (RHG for short) is a triple (Σ, N, R) where Σ is a finite set of labels (or terminals), N is a finite set of nonterminals, and R is a finite set of production rules with the form $T \to \sigma(T_1)T_2$ or $T \to \varepsilon$.

The definition for RHGs is the same as that for regular tree grammars [6] on the binary tree encoding of hedges [16] except that the above definition does not include the start nonterminals. Note that the definition looks different from the definition for RHGs given in [27] but the two definitions are the same in terms of expressive power.

Syntax:

$$\begin{array}{lll} prog & ::= prod_1 \dots prod_n \; rule_1 \dots rule_m & \text{(Program)} \\ prod & ::= \mathbf{data}\, T \doteq t & \text{(Production Rule)} \\ t & ::= \varepsilon \mid t_1 \,\texttt{.}\, t_2 \mid \sigma(t) \mid T & \text{(Type Expression)} \\ rule & ::= f(p) \doteq e & \text{(Definition Rule)} \\ e & ::= \varepsilon \mid e_1 \,\texttt{.}\, e_2 \mid \sigma(e) \mid x \mid f(x) & \text{(Expression)} \\ p & ::= \varepsilon \mid p_1 \,\texttt{.}\, p_2 \mid \sigma(p) \mid x :: T & \text{(Pattern)} \end{array}$$

(x is a variable, T is a type name, f is a function name, and σ is a label)

Semantics:

$$\frac{}{\varepsilon \Downarrow \varepsilon}\textsc{Eps} \qquad \frac{e \Downarrow v}{\sigma(e) \Downarrow \sigma(v)}\textsc{Con} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \,\texttt{.}\, e_2 \Downarrow v_1 \,\texttt{.}\, v_2}\textsc{Cat}$$

$$\frac{\exists\theta, \exists f(p) \doteq e.\; p\theta = v, \quad e\theta \Downarrow u}{f(v) \Downarrow u}\textsc{Fun}$$

Figure 1. The language describing forward transformations

We write $A \xrightarrow{*} h$ if nonterminal A generates hedge h. The semantics, or the language, of nonterminal A in G is defined by $[\![A]\!]_G = \{h \mid A \xrightarrow{*} h, h \text{ does not contain any nonterminal}\}$. For simplicity, we sometimes write $[\![A]\!]$ instead of $[\![A]\!]_G$ if G is clear from the context. A language of an RHG is called a *regular hedge language*.

We define the following relation for the discussions that follow.

Definition 2 (Horizontal Parallel)**.** Let R and S be sets of hedges. We define $R \parallel S$ if and only if there exist no hedges $h_1, h_3 \in R$, $h_2, h_4 \in S$ such that $h_1h_2 = h_3h_4$ but $h_1 \neq h_3$ and $h_2 \neq h_4$.

3.3 Syntax and Semantics

The syntax of the language is summarized in Figure 1. Formally, program $\mathcal{P}$ is a pair (G, R): RHG G defining types and set R defining rules of functions.

A type declaration

$$\mathbf{data}\, T \doteq t$$

describes production rule $T \rightarrow t$ in the RHG G. For convenience, we sometimes use regular expressions and production rules beyond RHG when we present examples of programs written in the language, if they actually define regular hedge language.

A rule of a function takes the form

$$f(p) \doteq e$$

where p is a pattern and e is a *treeless* expression [32]. In treeless expressions, all the arguments of a function call must be variables, i.e., there is no nested function application. Note that we treat hedge concatenation "$\texttt{.}$" as a "freeze"d data constructor rather than a function. To simplify our presentation, we add some restrictions to the language: every variable on the left-hand side of a rule must be used at most once on the corresponding right-hand side and the number of parameters of a function is always one. These restrictions are not essential and even the treeless restriction can be relaxed (Section 6). Note that, in the language, no output of a function can be decomposed by a pattern. For convenience, we use type expressions in variable patterns in addition to type names, as $x :: \texttt{<a>}^*$. Some example programs in this language have been given in the Introduction and Section 2.

We write a type environment obtained by gathering variable patterns in pattern p as Γ_p. For example, for $p = (x :: T, \texttt{<t>}(y :: T'))$, we have $\Gamma_p = \{x \mapsto T, y \mapsto T'\}$. Substitution θ is a function that maps a variable to a hedge or to the same variable. We write a pattern/expression obtained from t by replacing each variable x in t with hedge $\theta(x)$ as $t\theta$. Especially for a pattern, we assume that $p\theta$ implies $\theta(x) \in [\![\Gamma_p(x)]\!]_G$ for each variable x in p. The set of all hedges matching pattern p in a program $\mathcal{P}$, $[\![p]\!]_{\mathcal{P}}$, is defined by $[\![p]\!]_{\mathcal{P}} = \{h \mid \exists\theta.\; h = p\theta\}$. Note that variable pattern $x :: T$ only accepts the values in $[\![T]\!]_G$, e.g., $[\![x :: \texttt{<a>}]\!]_{\mathcal{P}} = \{\texttt{<a>}\}$. For simplicity, we write $[\![p]\!]$ instead of $[\![p]\!]_{\mathcal{P}}$ if $\mathcal{P}$ is clear from the context.

In this paper, we assume that transformation programs are *deterministic* in the sense that pattern matching is unique. Formally, for any concatenation pattern $p_1 \,\texttt{.}\, p_2$, $[\![p_1]\!]_{\mathcal{P}} \parallel [\![p_2]\!]_{\mathcal{P}}$ holds, and for any two rules $f(p) = e$ and $f(p') \doteq e'$, $[\![p]\!]_{\mathcal{P}} \cap [\![p']\!]_{\mathcal{P}} = \emptyset$ holds. For example, functions f and g below are not deterministic.

$$\begin{array}{ll} f(x :: \texttt{<a>}^* \,\texttt{.}\, y :: \texttt{<a>}^*) & \doteq \dots \\ g(x :: \texttt{<a>}) & \doteq \dots \\ g(x :: \texttt{<a>}^* \,\texttt{.}\, y :: \texttt{<a>}) & \doteq \dots \end{array}$$

The semantics of the language is defined by the call-by-value semantics shown in Figure 1.

4. Type-Based Specialization

This section introduces a program transformation method that specializes functions with respect to the types of their arguments. In a specialized program, for any function call $f(x)$ with $x :: A$ and rules $f(p_1) \doteq e_1, \dots, f(p_n) \doteq e_n$ of f, the apparent domain of f, $\bigcup_{1 \le i \le n} [\![p_i]\!]$, is equal to $[\![A]\!]$. The specialization, roughly speaking, with respect to type S, converts function f

$$f(\dots x :: T \dots) \doteq \dots g(x) \dots$$

to function $f|_S$

$$f|_S(\dots x :: T' \dots) \doteq \dots g|_{T'}(x) \dots$$

where each variable pattern $x :: T$ in a pattern is replaced with $x :: T'$ by *pattern specialization* of the pattern with respect to S. Then, the specialization attempts to generate rules of function g with respect to T'. Let us roughly explain the specialization with concrete examples of *reverse* and *idAB* below.

Consider the following function, *reverse*.

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{data}\, T \doteq \texttt{<a>} \mid \texttt{<b>}} \\ reverse(\varepsilon) & \doteq \varepsilon \\ reverse(a :: T \,\texttt{.}\, r :: T^*) & \doteq reverse(r) \,\texttt{.}\, a \end{array}$$

We specialize function *reverse* with respect to type $S = (\texttt{<a>} \,\texttt{.}\, \texttt{<b>})^*$. Here, the semantics of S, T, and T^* is defined by the following RHG.

$$\begin{array}{lll} S \rightarrow \texttt{<a>}S' & S \rightarrow \varepsilon & S' \rightarrow \texttt{<b>}S \\ T \rightarrow \texttt{<a>}T' & T \rightarrow \texttt{<b>}T' & T' \rightarrow \varepsilon \\ T^* \rightarrow \texttt{<a>}T^* & T^* \rightarrow \texttt{<b>}T^* & T^* \rightarrow \varepsilon \end{array}$$

First, the specialization tries to specialize the rule

$$reverse(\varepsilon) \doteq \varepsilon$$

with respect to type S. To achieve this specialization, our specialization tries to specialize pattern ε with respect to type ε. Since a set of inputs S contains ε, i.e., $\varepsilon \in [\![S]\!]$, pattern ε is specialized to ε, and the following rule is produced by the specialization.

$$reverse|_S(\varepsilon) \doteq \varepsilon$$

Then, the specialization tries to specialize the rule

$$reverse(a :: T \,\texttt{.}\, r :: T^*) \doteq reverse(r) \,\texttt{.}\, a$$

with respect to S. To achieve this, we try to specialize pattern $a :: T \,\texttt{.}\, r :: T^*$ to pattern $a :: H_1 \,\texttt{.}\, r :: H_2$ with respect to S. To find sets of hedges H_1 and H_2, we consider hedges $h_1 \in [\![T]\!]$, $h_2 \in [\![T^*]\!]$ satisfying $h_1 \,\texttt{.}\, h_2 \in [\![S]\!]$. For this purpose, we try to find nonterminal N satisfying $S \xrightarrow{*} h_1N$, $N \xrightarrow{*} h_2$, $[\![N]\!] \cap [\![T^*]\!] \neq \emptyset$, and $\{h \mid S \xrightarrow{*} hN\} \cap [\![T]\!] \neq \emptyset$. Since we have $\{h \mid S \xrightarrow{*} hS\} \cap [\![T]\!] = \emptyset$ but $[\![S']\!] \cap [\![T^*]\!] = [\![\texttt{<b>} \,\texttt{.}\, T^*]\!]$ and

$\{h \mid S \xrightarrow{*} hS'\} \cap [\![T]\!] = \{$<a>$\}$, a rule and types

$$\begin{array}{l} \mathbf{data}\ H_1 \mathrel{\hat{=}} \texttt{<a>} \\ \mathbf{data}\ H_2 \mathrel{\hat{=}} \texttt{<b>} \centerdot S \\ \mathit{reverse}|_S(a :: H_1 \centerdot r :: H_2) \mathrel{\hat{=}} \mathit{reverse}|_{H_2}(r) \centerdot a \end{array}$$

are produced. Since $\mathit{reverse}|_{H_2}(r)$ appears on the right-hand side of newly produced rules and the rules of function $\mathit{reverse}|_{H_2}(r)$ have not been produced, the specialization tries to specialize function $\mathit{reverse}$ with respect to type H_2. First, the specialization tries to specialize the rule

$$\mathit{reverse}(\varepsilon) \mathrel{\hat{=}} \varepsilon$$

with respect to H_2. To achieve this specialization, we specialize pattern ε with respect to H_2. Since H_2 does not contain ε, i.e., the rule is never used for an input in H_2, no rule is produced. Then, the specialization tries to specialize the rule

$$\mathit{reverse}(a :: T \centerdot r :: T^*) \mathrel{\hat{=}} \mathit{reverse}(r) \centerdot a$$

with respect to H_2. By a similar procedure to the above, the following rule is produced by the specialization.

$$\mathit{reverse}|_{H_2}(a :: \texttt{<b>} \centerdot r :: S) \mathrel{\hat{=}} \mathit{reverse}|_S(r) \centerdot a$$

Since the rules of $\mathit{reverse}|_S$ have already been produced, the specialization is complete. If we simplify H_1 by $H_1 = $ <a>, specialized function $\mathit{reverse}|_S$ is as follows.

$$\begin{array}{ll} \mathbf{data}\ S \mathrel{\hat{=}} (\texttt{<a>} \centerdot \texttt{<b>})^* & \\ \mathbf{data}\ T \mathrel{\hat{=}} \texttt{<a>} \mid \texttt{<b>} & \\ \mathbf{data}\ H_2 \mathrel{\hat{=}} \texttt{<b>} \centerdot S & \\ \mathit{reverse}|_S(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\ \mathit{reverse}|_S(a :: \texttt{<a>} \centerdot r :: H_2) & \mathrel{\hat{=}} \mathit{reverse}|_{H_2}(r) \centerdot a \\ \mathit{reverse}|_{H_2}(a :: \texttt{<b>} \centerdot r :: S) & \mathrel{\hat{=}} \mathit{reverse}|_S(r) \centerdot a \end{array}$$

Sometimes, the specialization produces more than one rule from a rule. Consider function idAB defined by

$$\mathit{idAB}(x :: (\texttt{<a>}|\texttt{<b>}) \centerdot y :: (\texttt{<a>}|\texttt{<b>})) \mathrel{\hat{=}} x \centerdot y$$

and the specialization of function idAB with respect to type $T = (\texttt{<a>} \centerdot \texttt{<b>}) \mid (\texttt{<b>} \centerdot \texttt{<a>})$. Here, T is represented by the following RHG.

$$T \to \texttt{<a>}U \quad T \to \texttt{<b>}V \quad U \to \texttt{<b>}W \quad V \to \texttt{<a>}W \quad W \to \varepsilon$$

Then, we try to specialize pattern $x :: (\texttt{<a>}|\texttt{<b>}) \centerdot y :: (\texttt{<a>}|\texttt{<b>})$ with respect to T. Similar to the above procedure, we try to find h_1, h_2 such that $h_1, h_2 \in [\![\texttt{<a>}|\texttt{<b>}]\!]$ and $h_1 \centerdot h_2 \in [\![T]\!]$. Unlike $\mathit{reverse}$, the choice of h_2 affects the choice of h_1; if we choose $h_2 = $ <b> then we must choose $h_1 = $ <a>, while if we choose $h_2 = $ <a> then we must choose $h_1 = $ <b>. In other words, sets $\{h \mid T \xrightarrow{*} hU\}$ and $\{h \mid T \xrightarrow{*} hV\}$ are different, where U and V are only nonterminals such that $[\![\texttt{<a>}|\texttt{<b>}]\!] \cap [\![U]\!] \neq \emptyset$ and $[\![\texttt{<a>}|\texttt{<b>}]\!] \cap [\![V]\!] \neq \emptyset$ hold. Hence, two rules

$$\begin{array}{l} \mathit{idAB}|_T(x :: \texttt{<a>} \centerdot y :: \texttt{<b>}) \mathrel{\hat{=}} x \centerdot y \\ \mathit{idAB}|_T(x :: \texttt{<b>} \centerdot y :: \texttt{<a>}) \mathrel{\hat{=}} x \centerdot y \end{array}$$

are produced for one rule of idAB.

To use the specialization in other analyses or in another automated frameworks, we should clarify two points.

- Whether the specialization terminates.
- Whether the specialization generates deterministic programs.

The second point requires careful treatment of cases where the specialization generates more than one rule from a rule.

4.1 Pattern Specialization

In the examples above, we consider set $\{h \mid A \xrightarrow{*} hB\}$ for nonterminals A, B in an RHG. This kind of set is always regular hedge language, i.e., there exists an RHG that describes the set. However, if we naively produce a new RHG when we encounter a concatenation pattern, a discussion on whether or not the specialization terminates becomes difficult. To make the discussion easier, we introduce a notion of *chopped RHG*, by which a set like $\{h \mid A \xrightarrow{*} hB\}$, which is obtained by "chopping" the language of the original RHG at nonterminal B, can easily be represented.

Definition 3 (Chopped RHG)**.** A *chopped RHG* $G\varepsilon E$ consists of an RHG G and set E of nonterminals in G, where the semantics of nonterminal A in the chopped RHG $G\varepsilon E$ is defined by $[\![A]\!]_{G\varepsilon E} = \{h \mid A \xrightarrow{*} hB, B \in E\}$.

Note that every RHG G can be converted to a chopped RHG $G\varepsilon F$ where $F = \{A \mid A \to \varepsilon \in R\}$ with the R of $G = (\_, \_, R)$[6]. It is not difficult to show that every chopped RHG $G\varepsilon E$ can be converted to an RHG G' where, for any nonterminal A in G, there exists a corresponding nonterminal A' in G' such that $[\![A]\!]_G = [\![A']\!]_{G'}$. We write an RHG obtained from an RHG G and an RHG G' by product-construction [6] as $G \times G'$, under which nonterminal $(A, A') \in G \times G'$ has language $[\![(A, A')]\!]_{G \times G'} = [\![A]\!]_G \cap [\![A']\!]_{G'}$. Similarly, we write RHG $(G \times G')\varepsilon(F \times E)$ as $G \times G'\varepsilon E'$ where $F = \{A \mid A \to \varepsilon \in R\}$ with the R of $G = (\_, \_, R)$. For example, types H_1, H_2 in the example of $\mathit{reverse}$ satisfy

$$\begin{array}{l} [\![H_1]\!] = [\![(T, S)]\!]_{G \times G\varepsilon S'} \\ [\![H_2]\!] = [\![(T^*, S)]\!]_{G \times G}. \end{array}$$

To simplify our explanation in this section, we represent type A by A_G ($A_{G\varepsilon E}$) where G ($G\varepsilon E$) is an RHG (a chopped RHG) used to define A. For example, $\mathit{reverse}$ can be rewritten as follows.

$$\begin{array}{ll} \mathit{reverse}(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\ \mathit{reverse}(a :: T_G \centerdot r :: T_G^*) & \mathrel{\hat{=}} \mathit{reverse}(r) \centerdot a \end{array}$$

Here, G is the RHG used to define types S, T, and T^*.

Pattern specialization is formally described by pattern specialization procedure $\mathsf{psp}(p; A_{G'\varepsilon E})$ in Figure 2 that calculates a set of specialized patterns from pattern p and a type described by A in $G'\varepsilon E$. Roughly, the behavior of psp for each line of the definition is as follows.

1. For pattern ε, the first line of psp returns pattern ε if $[\![A]\!]_{G'\varepsilon E}$ contains ε.
2. For pattern $x :: T_G$, the second line of psp returns pattern $x :: (T, A)_{G \times G'\varepsilon E}$ if the newly generated pattern accepts at least one hedge.
3. For pattern $\sigma(p)$, the third line of psp tries to find patterns p' such that $\bigcup \sigma(p') = [\![A]\!]_{G'\varepsilon E}$. To find such patterns p', the third line calls $\mathsf{psp}(p; B_{G'})$ for rule $A \to \sigma(B)C$ in G' with $B \in E$. Here, we use G' as $\mathsf{psp}(p; B_{G'})$ instead of $\mathsf{psp}(p; B_{G'\varepsilon E})$ because E is a set of *horizontally* chopping nonterminals. Pattern p in $\sigma(p)$ is not located on the same horizontal level as $\sigma(p)$.
4. For pattern $p_1 \centerdot p_2$, the fourth line of psp tries to find patterns p_1', p_2' such that $[\![p_1']\!] \subseteq [\![p_1]\!]$, $[\![p_2']\!] \subseteq [\![p_2]\!]$ and $\bigcup [\![p_1' \centerdot p_2']\!] = [\![A]\!]_{G'\varepsilon E}$. To find such patterns p_1', p_2', the fourth line "chop"s $A_{G'\varepsilon E}$ to $A_{G'\varepsilon\{B\}}$ and $B_{G'\varepsilon E}$, and calls $\mathsf{psp}(p_1; A_{G'\varepsilon\{B\}})$ and $\mathsf{psp}(p_2; B_{G'\varepsilon E})$ for any nonterminal B in G'.
5. The fifth line implies that the input pattern of psp cannot be used for input in $[\![A]\!]_{G'\varepsilon E}$.

Examples 1 and 2, which will be explained later, include the examples of psp.

[6] The notation "_" means that we do not care what "_" is, as "_" in Haskell.

$$\begin{array}{lll}
\mathsf{psp}(\varepsilon; \quad A_{G'\varepsilon E}) & \hat{=} & \{\varepsilon\} \quad \text{if } A \in E \\
\mathsf{psp}(x :: T_G;\ A_{G'\varepsilon E}) & \hat{=} & \{x :: (T, A)_{G\times G'\varepsilon E}\} \quad \text{if } [\![(T,A)]\!]_{G\times G'\varepsilon E} \neq \emptyset \\
\mathsf{psp}(\sigma(p); \quad A_{G'\varepsilon E}) & \hat{=} & \{\sigma(p') \mid p' \in \mathsf{psp}(p; B_{G'}),\ A \to \sigma(B)C \in \mathit{Rules}_{G'},\ C \in E\} \\
\mathsf{psp}(p_1 \centerdot p_2;\ A_{G'\varepsilon E}) & \hat{=} & \{p_1' \centerdot p_2' \mid p_1' \in \mathsf{psp}(p_1; A_{G'\varepsilon\{B\}}),\ p_2' \in \mathsf{psp}(p_2; B_{G'\varepsilon E}),\ B \in \mathit{NonTerms}_{G'}\} \\
\mathsf{psp}(\_; \quad A_{G'\varepsilon E}) & \hat{=} & \emptyset
\end{array}$$

where $\mathit{Rules}_{G'}$ is a set of all the production rules in G', and $\mathit{NonTerms}_{G'}$ is a set all the nonterminal in G'.

Figure 2. The definition of pattern specialization procedure psp

Theorem 1. Function psp correctly specializes pattern p with respect to type $A_{G'\varepsilon E}$, i.e.,

$$[\![A]\!]_{G'\varepsilon E} \cap [\![p]\!] = \bigcup\{[\![p']\!] \mid p' \in \mathsf{psp}(p; A_{G'\varepsilon E})\}.$$

Proof Sketch. Theorem 1 is easily proved by using the induction on the structure of p. To clarify the behavior of psp, we prove the induction step where $p = p_1 \centerdot p_2$ because the discussion on other cases is rather easy.

First, we show the following statement.

$$[\![A]\!]_{G'\varepsilon E} \cap [\![p_1 \centerdot p_2]\!] \subseteq \bigcup\{[\![p_1' \centerdot p_2']\!] \mid p_1' \centerdot p_2' \in \mathsf{psp}(p_1 \centerdot p_2; A_{G'\varepsilon E})\}$$

Let x and y be hedges such that $x \centerdot y \in [\![A]\!]_{G'\varepsilon E}$, $x \in [\![p_1]\!]$ and $y \in [\![p_2]\!]$. Since we have $x \centerdot y \in [\![A]\!]_{G'\varepsilon E}$, there exists at least one nonterminal B in G such that $A \xrightarrow{*} xB$ and $B \xrightarrow{*} yC$ for some C. That is, we have $x \in [\![A]\!]_{G'\varepsilon\{B\}}$ and $y \in [\![B]\!]_{G'\varepsilon E}$. From the induction hypothesis, we have

$$[\![A]\!]_{G'\varepsilon\{B\}} \cap [\![p_1]\!] \subseteq \bigcup\{[\![p_1']\!] \mid p_1' \in \mathsf{psp}(p_1; A_{G'\varepsilon\{B\}})\}$$

and

$$[\![B]\!]_{G'\varepsilon E} \cap [\![p_2]\!] \subseteq \bigcup\{[\![p_2']\!] \mid p_2' \in \mathsf{psp}(p_2; A_{G'\varepsilon E})\}.$$

Hence, there exist patterns $p_1' \in \mathsf{psp}(p_1; A_{G'\varepsilon\{B\}})$ and $p_2' \in \mathsf{psp}(p_2; A_{G'\varepsilon E})$ such that $x \in [\![p_1']\!], y \in [\![p_2']\!]$. By using the definition of psp, we have

$$p_1' \centerdot p_2' \in \mathsf{psp}(p_1 \centerdot p_2; A_{G'\varepsilon E}),$$

which implies the statement.

Then, we show the following statement.

$$[\![A]\!]_{G'\varepsilon E} \cap [\![p_1 \centerdot p_2]\!] \supseteq \bigcup\{[\![p_1' \centerdot p_2']\!] \mid p_1' \centerdot p_2' \in \mathsf{psp}(p_1 \centerdot p_2; A_{G'\varepsilon E})\}$$

Let x and y be hedges such that $x \in [\![p_1']\!]$ and $y \in [\![p_2']\!]$ for some patterns p_1', p_2' such that $p_1' \centerdot p_2' \in \mathsf{psp}(p_1 \centerdot p_2; A_{G'\varepsilon E})$. By using the definition of psp, we have $p_1' \in \mathsf{psp}(p_1; A_{G'\varepsilon\{B\}})$ and $p_2' \in \mathsf{psp}(p_2; B_{G'\varepsilon E})$. From the induction hypothesis, we have

$$[\![A]\!]_{G'\varepsilon\{B\}} \cap [\![p_1]\!] \supseteq \bigcup\{[\![p_1']\!] \mid p_1' \in \mathsf{psp}(p_1; A_{G'\varepsilon\{B\}})\}$$

and

$$[\![B]\!]_{G'\varepsilon E} \cap [\![p_2]\!] \supseteq \bigcup\{[\![p_2']\!] \mid p_2' \in \mathsf{psp}(p_2; A_{G'\varepsilon E})\}.$$

Hence, we have $x \in [\![A]\!]_{G'\varepsilon\{B\}} \cap [\![p_1]\!]$ and $y \in [\![B]\!]_{G'\varepsilon E} \cap [\![p_2]\!]$. By using the definition of $\centerdot$, we have $x \centerdot y \in [\![p_1 \centerdot p_2]\!]$. By using the definition of chopped RHG, we have $x \centerdot y \in [\![A]\!]_{G'\varepsilon E}$. Hence, we have $x \centerdot y \in [\![A]\!]_{G'\varepsilon E} \cap [\![p_1 \centerdot p_2]\!]$, which implies the statement. □

4.2 Specialization Algorithm

We use the following notion of unambiguous RHGs in the specialization algorithm.

Definition 4 (Unambiguous RHG)**.** An RHG $G = (\Sigma, N, R)$ is unambiguous if

$$\forall A, B \in N.\ A \neq B \Rightarrow [\![A_G]\!] \cap [\![B_G]\!] = \emptyset \qquad \text{(UnAmb)}$$

and

$$\forall A \in N.\ [\![A_G]\!] \neq \emptyset$$

hold.

It is known that using conversion from NFTA to DFTA [6] every RHG G can be converted to unambiguous RHG G' in which, for any nonterminal A in G, there exist nonterminals $A_1', \ldots, A_n'$ in G' satisfying $[\![A]\!]_G = \bigcup_{i\in\{1,\ldots,n\}} [\![A_i']\!]_{G'}$. Similarly, every chopped RHG $G\varepsilon E$ has a corresponding unambiguous RHG G'.

Now, we are ready to define our specialization procedure.

Algorithm (Type-Based Specialization)**.**
Input: A program.
Output: A specialized program.
Procedure:

1. For each function call $f(x)$ in each rule $h(q) \mathrel{\hat{=}} \mathcal{C}[f(x)]$ in a program with $\Gamma_q(x) = A_{G'\varepsilon E}$, and then repeat Steps 2–5.
2. Construct an unambiguous RHG G'' corresponding to $G'\varepsilon E$, in which the nonterminals $A_1'', \ldots, A_n''$ satisfy $\bigcup_{i\in\{1\ldots n\}} [\![A_i'']\!] = [\![A]\!]_{G\varepsilon E'}$.
3. For each rule $f(p) \mathrel{\hat{=}} e$ of f in the original program, repeat Steps 3–5.
4. For each specialized pattern $p' \in \bigcup_{i\in\{1\ldots n\}} \mathsf{psp}(p; {A_i''}_{G''})$, generate a rule $f|_{A_{G'\varepsilon E}}(p') \mathrel{\hat{=}} e'$ where e' is obtained from e by replacing function calls $g(y)$ by $g|_T(y)$ where $T = \Gamma_{p'}(y)$.
5. Recursively apply this algorithm to all the function calls occurring in the newly-produced rules until no new rules are generated in Step 4. □

The construction of an unambiguous RHG in Step 3 guarantees that the specialized programs are deterministic.

Let us explain the behavior of the specialization algorithm step-by-step using the examples of *reverse* and *idAB*, which were presented earlier in this section.

Example 1 (*reverse*)**.** Let us consider a program as follows.

$$\mathit{main}(x :: S) \mathrel{\hat{=}} \mathit{reverse}(x)$$

Here, type S is defined by $S = (\texttt{<a>} \centerdot \texttt{<b>})^*$. Let G be the RHG used to define S, T and T^* in *reverse*. First, by using Step 1, we target *reverse* and type S_G. Second, by using Step 2, from G, we construct an unambiguous RHG G' that defines the semantics of S. Here, this is easily done by collecting production rules that are relevant to S. Third, by using Step 3, for two rules of *reverse*, we specialize the rule $\mathit{reverse}(\varepsilon) \mathrel{\hat{=}} \varepsilon$ with respect to $S_{G'}$. Fourth, by using the Step 4, since we have $\mathsf{psp}(\varepsilon; S_{G'}) = \{\varepsilon\}$ by definition, a rule

$$\mathit{reverse}|_{S_{G'}}(\varepsilon) \mathrel{\hat{=}} \varepsilon$$

is generated. Fifth, by using Step 5, since there is no function on the right-hand side of the generated rule, we go back to Step 3 to deal with the rest of the rules of *reverse*. Sixth, by using Step 3, we specialize the rule $\mathit{reverse}(a :: T_G \centerdot r :: T_G^*) \mathrel{\hat{=}} \mathit{reverse}(r) \centerdot a$ with respect to $S_{G'}$. Seventh, by using Step 4, since we have

$$\begin{array}{l}\mathsf{psp}(a :: T_G \centerdot r :: T_G^*; S_{G'}) \\ \qquad = \{a :: (T, S)_{G\times G'\varepsilon S'} \centerdot r :: (T^*, S')_{G\times G'}\},\end{array}$$

a rule

$$\begin{array}{l} \mathit{reverse}|_{S_{G'}}(a :: (T,S)_{G\times G'\varepsilon S'} \mathbin{\centerdot} r :: (T^*,S')_{G\times G'}) \\ \quad \mathrel{\hat{=}} \mathit{reverse}|_{(T^*,S')_{G\times G'}}(r) \mathbin{\centerdot} a \end{array}$$

is generated. Eighth, by using Step 5, since the generated rule contains function call $\mathit{reverse}|_{(T^*,S')_{G\times G'}}$, we try to specialize $\mathit{reverse}$ with respect to type $(T^*,S')_{G\times G'}$. Ninth, by using Step 2, we obtain unambiguous RHG G'' defined by the rules

$$U \to \texttt{<b>}() \mathbin{\centerdot} V \quad V \to \texttt{<a>}() \mathbin{\centerdot} U \quad V \to \varepsilon$$

where $[\![U]\!]_{G''} = [\![(T^*,S')]\!]_{G\times G'}$. Tenth, by using Step 3, we specialize rule $\mathit{reverse}(\varepsilon) \mathrel{\hat{=}} \varepsilon$ with respect to type $U_{G''}$. Eleventh, by using Step 4, since $\mathsf{psp}(\varepsilon; U_{G''}) = \emptyset$, we go back to Step 3 without producing any rules. Twelfth, by using Step 3, we specialize rule $\mathit{reverse}(a :: T_G \mathbin{\centerdot} r :: T_G^*) \mathrel{\hat{=}} \mathit{reverse}(r) \mathbin{\centerdot} a$ with respect to type $U_{G''}$. Thirteenth, by using Step 4 since we have

$$\begin{array}{l} \mathsf{psp}(a :: T_G \mathbin{\centerdot} r :: T_G^*; U_{G''}) \\ \quad = \{a :: (T,U)_{G\times G''\varepsilon V} \mathbin{\centerdot} r :: (T^*,V)_{G\times G''}\} \end{array}$$

a rule

$$\begin{array}{l} \mathit{reverse}|_{(T^*,S')_{G\times G'}}(a :: (T,U)_{G\times G''\varepsilon V} \mathbin{\centerdot} r :: (T^*,V)_{G\times G''}) \\ \quad \mathrel{\hat{=}} \mathit{reverse}|_{(T^*,V)_{G\times G''}}(r) \mathbin{\centerdot} a \end{array}$$

is generated. Note that here we have the following equations on regular hedge languages.

$$\begin{array}{ll} [\![(T,S)_{G\times G'\varepsilon S'}]\!] & = \{\texttt{<a>}\} \\ [\![(T^*,S')_{G\times G'}]\!] & = \{\texttt{<b>} \mathbin{\centerdot} h \mid h \in [\![S]\!]_G\} \\ [\![(T,U)_{G\times G''\varepsilon V}]\!] & = \{\texttt{<b>}\} \\ [\![(T^*,V)_{G\times G''}]\!] & = [\![S]\!]_G \end{array}$$

Fourteenth, at Step 5, we do nothing and the specialization is complete while $\mathit{reverse}|_{(T^*,V)_{G\times G''}}$ appears on the right-hand side of the generated rule, because $\mathit{reverse}|_{(T^*,V)_{G\times G''}}$ is nothing but $\mathit{reverse}|_{S_{G'}}$ whose rules have already been generated. Collecting the generated rules, we obtain the following program.

$$\begin{array}{ll} \textbf{data}\ S \mathrel{\hat{=}} (\texttt{<a>} \mathbin{\centerdot} \texttt{<b>})^* & \\ \textbf{data}\ U \mathrel{\hat{=}} \texttt{<b>} \mathbin{\centerdot} V & \\ \textbf{data}\ V \mathrel{\hat{=}} \texttt{<a>} \mathbin{\centerdot} U \mid \varepsilon & \\ \mathit{reverse}|_S(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\ \mathit{reverse}|_S(a :: \texttt{<a>} \mathbin{\centerdot} r :: U) & \mathrel{\hat{=}} \mathit{reverse}|_U(r) \mathbin{\centerdot} a \\ \mathit{reverse}|_U(a :: \texttt{<b>} \mathbin{\centerdot} r :: S) & \mathrel{\hat{=}} \mathit{reverse}|_S(r) \mathbin{\centerdot} a \end{array}$$

Ensure that $[\![U]\!] = [\![\texttt{<b>} \mathbin{\centerdot} S]\!]$.

Example 2 (idAB)**.** Let us consider a program as follows.

$$\mathit{main}(x :: T_G) \mathrel{\hat{=}} \mathit{idAB}(x)$$

where idAB is defined by

$$\mathit{idAB}(x :: S_G \mathbin{\centerdot} y :: S_G) \mathrel{\hat{=}} x \mathbin{\centerdot} y$$

and G is a grammar defined as follows.

$$\begin{array}{lllll} S \to \texttt{<a>}S' & S \to \texttt{<b>}S' & S' \to \varepsilon & & \\ T \to \texttt{<a>}U & T \to \texttt{<b>}V & U \to \texttt{<b>}W & V \to \texttt{<a>}W & W \to \varepsilon \end{array}$$

First, by using Step 1, we target idAB and type T_G. Second, by using Step 2, from G, we construct an unambiguous RHG G' that defines the semantics of T. In this case, this is easily done by collecting production rules that are relevant to T. Third, by using Step 3, since idAB only has one rule, we specialize rule $\mathit{idAB}(x :: S_G \mathbin{\centerdot} y :: S_G) \mathrel{\hat{=}} x \mathbin{\centerdot} y$ with respect to $T_{G'}$. Fourth, by using Step 4, since we have

$$\begin{array}{l} \mathsf{psp}(x :: S_G \mathbin{\centerdot} y :: S_G; T_{G'}) \\ \quad = \left\{\begin{array}{l} x :: (S,T)_{G\times G'\varepsilon V} \mathbin{\centerdot} y :: (S,V)_{G\times G'}, \\ x :: (S,T)_{G\times G'\varepsilon U} \mathbin{\centerdot} y :: (S,U)_{G\times G'} \end{array}\right\}, \end{array}$$

the rules

$$\begin{array}{l} \mathit{idAB}|_T(x :: (S,T)_{G\times G'\varepsilon V} \mathbin{\centerdot} y :: (S,V)_{G\times G'}) \mathrel{\hat{=}} x \mathbin{\centerdot} y \\ \mathit{idAB}|_T(x :: (S,T)_{G\times G'\varepsilon U} \mathbin{\centerdot} y :: (S,U)_{G\times G'}) \mathrel{\hat{=}} x \mathbin{\centerdot} y \end{array}$$

are generated. Fifth, at Step 5, we do nothing and the specialization is complete because the generated rules do not contain any function call on the right-hand sides. If we simplify the patterns, the following program is obtained by the specialization.

$$\begin{array}{l} \mathit{idAB}|_T(x :: \texttt{<b>} \mathbin{\centerdot} y :: \texttt{<a>}) \mathrel{\hat{=}} x \mathbin{\centerdot} y \\ \mathit{idAB}|_T(x :: \texttt{<a>} \mathbin{\centerdot} y :: \texttt{<b>}) \mathrel{\hat{=}} x \mathbin{\centerdot} y \end{array}$$

4.3 Properties of Specialization

We then prove that the specialization algorithm terminates. Thanks to chopped RHGs, we can avoid introducing a new type described by a *new* RHG in pattern specialization; psp only generates a type expressed by the composition of *existing* (chopped) RHGs, which serves as a key to our proof. In advance to the proof of termination, we prove the lemmas below. In the proof of the lemmas for an unambiguous RHG G, we use the notations $\mathcal{D}_\mathsf{G}(x), \mathcal{D}'_\mathsf{G}(x), \mathcal{D}''_\mathsf{G}(x), \mathcal{D}'''_\mathsf{G}(x)$ to represent some disjunctive normal forms (DNFs) on the predicates of form $\mathsf{p}_{A,B}(x)$ for $A, B \in \mathsf{G}$ where predicate $\mathsf{p}_{A,B}(x)$ means $\exists y.\ (x \mathbin{\centerdot} y) \in [\![A]\!]_\mathsf{G} \wedge y \in [\![B]\!]_\mathsf{G}$. The number of all DNFs on the predicates of form $\mathsf{p}_{A,B}(x)$ is finite because the number of the predicates is finite. Note that, for an unambiguous G, we have

$$\forall x.\ \left(\exists y.\ (x \mathbin{\centerdot} y) \in [\![A]\!]_G \wedge y \in [\![B]\!]_G \Leftrightarrow x \in [\![A]\!]_{G\varepsilon\{B\}}\right).$$

Roughly, Lemma 1 says that, when psp introduces type $A_{G''\varepsilon M}$, if the semantics of all the nonterminals in an RHG associated with the input of psp can be expressed by the composition of existing types, then the semantics of all the nonterminals in G'' also can be expressed by the composition of existing types. Lemma 2 states that all the nonterminals in a chopped RHG $G''\varepsilon E''$ where E'' is a set of nonterminals in G'' also can be expressed by the composition of existing types if G'' is unambiguous. Lemma 3 and 4 state that there exist methods transforming RHGs to suitable RHGs preserving the property where the semantics of all the nonterminals in the RHG can be expressed by the composition of existing types.

Lemma 1. Let $\mathcal{P} = (G, \_)$ be program where $G = (\_, N, \_)$. Let q be a pattern in the program, $\mathsf{psp}(q; A'_{G'})$ with $G' = (\_, N', \_)$ satisfies the following statement.

$$\begin{array}{l} \forall T \in N, \exists \mathcal{D}_\mathsf{G}.\ (\forall x.\ x \in [\![T']\!]_G \equiv \mathcal{D}_\mathsf{G}(x)) \\ \wedge\, \forall T' \in N', \exists \mathcal{D}'_\mathsf{G}.\ (\forall x.\ x \in [\![T']\!]_{G'} \equiv \mathcal{D}'_\mathsf{G}(x)) \\ \Rightarrow \begin{pmatrix} \forall q' \in \mathsf{psp}(q; A'_{G'}), \\ \forall x \in \mathsf{vars}(q'). \\ \text{let } (\_, G''\varepsilon E'') = \Gamma_{q'}(x) \text{ with } G'' = (\_, N'', \_), \\ \forall T'' \in N'', \exists \mathcal{D}''_\mathsf{G}.\ (\forall x.\ x \in [\![T'']\!]_{G''} \equiv \mathcal{D}''_\mathsf{G}(x)) \end{pmatrix} \end{array}$$

Proof. By using the definition of psp, when $B_{G''\varepsilon E''} = \Gamma_{q'}(x)$ for $x \in \mathsf{vars}(q')$ with $q' \in \mathsf{psp}(q; A'_{G'})$, G'' and E'' take the form

$$G''\varepsilon E'' = G \times G'\varepsilon E' = (G \times G')\varepsilon(F \times E')$$

for some $E' \subseteq N'$ where $F = \{A \mid A \to \varepsilon \in R\}$ with R of $G = (\_, \_, R)$. Therefore, $(T, T') \in N''$ implies $[\![(T,T')]\!]_{G''} = [\![T]\!]_G \cap [\![T']\!]_{G'}$. By using the premise of the lemma, we have $x \in [\![T]\!]_G \equiv \mathcal{D}_\mathsf{G}(x)$ and $x \in [\![T']\!]_{G'} \equiv \mathcal{D}'_\mathsf{G}(x)$. Then, there exists DNF $\mathcal{D}''_\mathsf{G}(x)$ on the atomic predicates with form $\mathsf{p}_{A,B}(x)$ such that

$$\begin{array}{rcl} x \in [\![(T,T')]\!]_{G''} & \equiv & x \in ([\![T]\!]_G \cap [\![T']\!]_{G'}) \\ & \equiv & \mathcal{D}_\mathsf{G}(x) \wedge \mathcal{D}'_\mathsf{G}(x) \\ & \equiv & \mathcal{D}''_\mathsf{G}(x) \end{array}$$

Thus, the proof is done. □

Lemma 2. For unambiguous $G = (\Sigma, N, R)$, if any $T \in N$ satisfies $x \in T \equiv \mathcal{D}_{\mathsf{G}}(x)$ for some $\mathcal{D}_{\mathsf{G}}(x)$, then any $T \in G\varepsilon E$ for any $E \subseteq N$ satisfies $x \in T \equiv \mathcal{D}'_{\mathsf{G}}(x)$ for some $\mathcal{D}'{}_{\mathsf{G}}(x)$.

Proof. The language of nonterminal T in $G\varepsilon E$ can be expressed in the following form because G is unambiguous.

$$\begin{aligned}\llbracket T \rrbracket_{G\varepsilon E} &= \left\{ x \;\middle|\; \bigvee_{S \in E} T \xrightarrow{*} xS \wedge S \xrightarrow{*} y \right\} \\ &= \left\{ x \;\middle|\; \bigvee_{S \in E} \exists y.\ (x \centerdot y) \in \llbracket T \rrbracket_G, y \in \llbracket S \rrbracket_G \right\}\end{aligned}$$

When we write a corresponding DNF to a nonterminal A in G as $\mathcal{D}^A_{\mathsf{G}}(x)$, the above formula can be rewritten as follows.

$$\llbracket T \rrbracket_{G\varepsilon E} = \left\{ x \;\middle|\; \bigvee_{S \in E} \exists y.\ \mathcal{D}^T_{\mathsf{G}}(x \centerdot y) \wedge \mathcal{D}^S_{\mathsf{G}}(y) \right\}$$

Using the fact that $\mathsf{p}_{A,B}(x \centerdot y) = \bigvee_{C \in \mathsf{N}} \mathsf{p}_{A,C}(x) \wedge \mathsf{p}_{C,B}(y)$ where $\mathsf{G} = (\_, \mathsf{N}, \_)$, let $\mathcal{F}_i(z), \mathcal{F}'_i(z)$ be logical formulae containing $\wedge$, $\neg$ on atomic predicates with form $\mathsf{p}_{A,B}(z)$ for $i \in \{1, \ldots, n\}$ for some n; rearranging DNF, we can rewrite the above formula as follows

$$\llbracket T \rrbracket_{G\varepsilon E} = \left\{ x \;\middle|\; \bigvee_{i} \exists y.\ \mathcal{F}_i(x) \wedge \mathcal{F}'_i(y) \right\},$$

where we use $\exists$-distributivity ($\exists x.\ p(x) \vee p'(x) \equiv (\exists x.p(x)) \vee (\exists x.p'(x))$). Since each $\mathcal{F}_i(x)$ is irrelevant to quantifier $\exists y$, we have

$$\llbracket T \rrbracket_{G\varepsilon E} = \left\{ x \;\middle|\; \bigvee_{i} \mathcal{F}_i(x) \wedge \exists y.\ \mathcal{F}'_i(y) \right\}.$$

Since each $\exists y.\ \mathcal{F}'_i(y)$ is true or false, i.e., removable, we obtain

$$\llbracket T \rrbracket_{G\varepsilon E} \equiv \mathcal{D}'_{\mathsf{G}}(x)$$

for some $\mathcal{D}'_{\mathsf{G}}(x)$, which implies the statement of the lemma. □

Lemma 3. There exists a method of constructing RHG $G' = (\_, N', \_)$ form chopped RHG $G\varepsilon E$ with $G = (\_, N, \_)$ satisfying

$$\forall T \in N, \exists T' \in N'.\ \llbracket T \rrbracket_{G\varepsilon E} = \llbracket T' \rrbracket_{G'}$$

and

$$\begin{aligned}&\forall T \in N, \exists \mathcal{D}_{\mathsf{G}}.\ x \in \llbracket T \rrbracket_{G\varepsilon E} \equiv \mathcal{D}_{\mathsf{G}}(x) \\ &\quad\Rightarrow \forall T' \in N', \exists \mathcal{D}'_{\mathsf{G}}.\ x \in \llbracket T' \rrbracket_{G'} \equiv \mathcal{D}_{\mathsf{G}}(x).\end{aligned}$$

Proof Sketch. For each nonterminal T in N, two nonterminals are introduced in G': T' for $\llbracket T \rrbracket_{G\varepsilon E}$ and T for $\llbracket T \rrbracket_G$. Productions rules of these nonterminals are obtained straightforwardly. Note that XDuce uses a similar conversion method [15]. □

Lemma 4. There exists a method of constructing unambiguous RHG $G' = (\_, N', \_)$ from RHG $G = (\_, N, \_)$ satisfying

$$\begin{aligned}&\forall T \in N, \exists \mathcal{D}_{\mathsf{G}}.\ x \in \llbracket T \rrbracket_G \equiv \mathcal{D}_{\mathsf{G}}(x) \\ &\quad\Rightarrow \forall T' \in N', \exists \mathcal{D}'_{\mathsf{G}}.\ x \in \llbracket T' \rrbracket_{G'} \equiv \mathcal{D}'_{\mathsf{G}}(x).\end{aligned}$$

Proof Sketch. The subset construction that converts NFTA to DFTA [6] satisfies the above condition. The minimization of RHGs [6] after disambiguation can also be applied preserving the condition. □

Theorem 2. The specialization terminates.

Proof Sketch. We prove the theorem by showing that the number of regular hedge *languages* appearing in the algorithm is finite.

Let $\mathcal{P} = (G, \_)$ be program with $G = (\_, \_, R)$. We define G as an unambiguous RHG corresponding to G. Then, any nonterminal T in G satisfies

$$x \in \llbracket T \rrbracket_G \equiv \bigvee_{i \in \{1,\ldots,n\}, C \in F} \mathsf{p}_{A_{T,i},C}(x) = \mathcal{D}'_{\mathsf{G}}(x)$$

for some $\mathcal{D}_{\mathsf{G}}$ where $\llbracket A_{T,1}, \ldots, A_{T,n_T} \rrbracket_{\mathsf{G}} = \llbracket T \rrbracket_G$ and $F = \{X \mid X \to \varepsilon \in \mathsf{R}\}$ with $\mathsf{G} = (\_, \_, \mathsf{R})$. Using Lemmas 1, 2, 3 and 4, by using the induction of the number of times psp is applied, every type T appearing in the program satisfies $\forall x \in \llbracket T \rrbracket \equiv \mathcal{D}_{\mathsf{G}}(x)$ for some $\mathcal{D}_{\mathsf{G}}$. Recall that the number of the DNFs is finite.

In the algorithm, we also use the fact that the equivalence of two regular hedge languages is decidable [6]. □

The proof of the termination also gives an upper bound for program growth. Since the number of DNFs on m atomic predicates is $\mathrm{O}(2^{2^m})$, an upper bound for the number of rules in a specialized program is $\mathrm{O}(2^{2^{n^2}})$ where n is the number of nonterminals in G of program $\mathcal{P} = (G, \_)$.

One may think that there is a simpler and more concise proof for the above theorem. The reason we proved the theorem as above is that we wanted to introduce flexibility to choose an internal representation of a type, i.e., an RHG. For example, we can apply minimization [6] to the unambiguous RHGs obtained in Step 2, and we can merge some patterns returned by psp as long as the merges preserve the semantics of the program. These techniques have improved the efficiency of our prototype implementation. Note that the specialization algorithm contains equivalence checks between types and both techniques benefit by reducing the number of equivalence checks. Equivalence checks on types described by RHGs are decidable but are EXPTIME-complete problems [6].

To guarantee that the programs after the specialization are deterministic, we must ensure that the following condition holds for pattern p and nonterminals $A''_1 \ldots, A''_n$ in unambiguous G'.

$$\big(\exists i.\ p', p'' \in \mathsf{psp}(p; A''_i{}_{G''})\big) \Rightarrow \llbracket p' \rrbracket \cap \llbracket p'' \rrbracket = \emptyset \qquad \text{(D-PSP)}$$

Thanks to the unambiguity (UnAmb) of G', we have the following lemma for determinism.

Lemma 5. For any nonterminal A in G, and any pattern p, the condition (D-PSP) holds.

Hence, we have the following theorem.

Theorem 3. The specialization returns deterministic programs.

Note that the method of specialization does not infer the output types of functions. In the specialization, we have assumed that any output of a function cannot be decomposed by a pattern of another function. In other words, an input to be decomposed by the pattern of a function is always part of an input of a function that calls the function. Also note that the domain of a function in our target language may be beyond the regular hedge language, concretely, context-free. This context-freeness results from the recursion structure of functions. Note that the specialization does not change the recursion structure; if the domain of the original function is context-free then the domain of the specialized function is also context-free. For example, function cf defined by

$$\begin{aligned}&\mathbf{data}\ T \mathrel{\hat{=}} (\texttt{<a>}|\texttt{<b>})^* \\ &cf(x :: T) \mathrel{\hat{=}} g(x) \\ &g(\varepsilon) \qquad\quad\ \mathrel{\hat{=}} \varepsilon \\ &g(\texttt{<a>} \centerdot r \centerdot \texttt{<b>}) \mathrel{\hat{=}} \texttt{<c>} \centerdot g(x) \centerdot \texttt{<d>}\end{aligned}$$

is specialized to the following function.

$$\begin{aligned}&cf(x :: T) \mathrel{\hat{=}} g|_T(x) \\ &g|_T(\varepsilon) \qquad\qquad\quad \mathrel{\hat{=}} \varepsilon \\ &g|_T(\texttt{<a>} \centerdot r :: T \centerdot \texttt{<b>}) \mathrel{\hat{=}} \texttt{<c>} \centerdot g|_T(x) \centerdot \texttt{<d>}\end{aligned}$$

The specialization eliminates the use of subtyping by replacing a type in a variable pattern and generating new rules of specialized functions so that everything is clear from the recursion structures in a specialized program.

Theorem 4 (Fully-Specializedness)**.** After specialization, for any function call $f(x)$ occurring in rule $g(p) \mathrel{\hat{=}} \mathcal{C}[f(x)]$, for any substitution θ, $\exists v.\ f(x\theta) \Downarrow v$ implies $\theta(x) \in [\![\Gamma_p(x)]\!]$.

Theorem 5 (Correctness)**.** For $f|_T$ obtained from function f and type T by the specialization, $v \in T, f(v) \Downarrow u$ if and only if $f|_T(v) \Downarrow u$.

Theorem 4 is the main result of the specialization. The theorem states that, after specialization, we do not need to consider types of variables in estimating the type (range) of an expression except a variable expression. For example, in the type inference of expression $f(x)$, we do not need to consider the type of x while, in the type inference of expression x, we must consider the type of x.

5. Applications

This section explains the effectiveness of our proposed specialization by presenting applications in which the specialization plays an important role.

5.1 Type Inference

The specialization enables us to do simple and exact type inference/checking. Type inference and type checking are important features of languages for for XML transformations and much research has been done in this area [11, 16, 22, 23].

Precise type inference for our target program without specialization is not straightforward because the output type of function f may differ from the type of function call $f(x)$. For example, the type of function call $f(x)$ in g is actually singleton set $\{\texttt{<a>}\}$ while the range of function f is a set described by $\texttt{<a>}^*$.

$$g(x :: \texttt{<a>}) \mathrel{\hat{=}} f(x)$$
$$f(x :: \texttt{<a>}^*) \mathrel{\hat{=}} x$$

However, in specialized programs the output type of function f and the type of function call $f(x)$ coincide. Thanks to this property, we can apply an algorithm similar to the type inference in [23], by which the types of expressions are exactly calculated by using context-free grammars.

For example, consider the following specialized program.

$$\begin{array}{l}
\textbf{data}\ C \mathrel{\hat{=}} (\texttt{<chapter>}(\texttt{<title>}(\textit{string}) \,.\, P \,.\, S))^* \\
\textbf{data}\ S \mathrel{\hat{=}} (\texttt{<section>}(\texttt{<title>}(\textit{String}) \,.\, P))^* \\
\textbf{data}\ P \mathrel{\hat{=}} (\texttt{<p>}(\textit{String}))^* \\
\mathit{c2x}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
\mathit{c2x}(\texttt{<chapter>}(\texttt{<title>}(t :: \textit{String}) \,.\, p :: P \,.\, s :: S) \,.\, r :: C) \\
\quad \mathrel{\hat{=}} \texttt{<h1>}(t) \,.\, p \,.\, \mathit{s2x}(s) \,.\, \mathit{c2x}(r) \\
\mathit{s2x}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
\mathit{s2x}(\texttt{<section>}(\texttt{<title>}(t :: \textit{String}) \,.\, p :: P) \,.\, r :: S) \\
\quad \mathrel{\hat{=}} \texttt{<h2>}(t) \,.\, p \,.\, \mathit{s2x}(r)
\end{array}$$

This function transforms a hedge with a paper-like structure as

$$\begin{array}{l}
\texttt{<chapter>}(\texttt{<title>}(t_1) \,.\, \texttt{<p>}(p_1) \\
\qquad\qquad \,.\, \texttt{<section>}(\texttt{<title>}(t_1) \,.\, \texttt{<p>}(p_2))) \\
\texttt{<chapter>}(\texttt{<title>}(t_3))
\end{array}$$

to an XHTML fragment as

$$\texttt{<h1>}(t_1) \,.\, \texttt{<p>}(p_1) \,.\, \texttt{<h2>}(t_2) \,.\, \texttt{<p>}(p_2) \,.\, \texttt{<h1>}(t_3).$$

Inference of the output types of a function is sufficient because the output types of functions and the types of function-call expressions coincide and the types of other expressions are calculated easily from the types. The basic idea underlying Maneth et al.'s algorithm is to forget the parameters of functions to generate a grammar that describes the range of functions. For example, from the above program, we obtain a grammar

$$\begin{array}{l}
T_{c2x} \rightarrow \varepsilon \\
T_{c2x} \rightarrow \texttt{<h1>}(\textit{String}) \,.\, P \,.\, T_{s2x} \,.\, T_{c2x} \\
T_{s2x} \rightarrow \varepsilon \\
T_{s2x} \rightarrow \texttt{<h2>}(\textit{String}) \,.\, P \,.\, T_{s2x}
\end{array}$$

that exactly calculates the ranges of functions $\mathit{c2x}$ and $\mathit{s2x}$. Sometimes, the range of a function is not regular as function f defined by

$$\begin{array}{ll}
f(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\
f(\texttt{<c>}(x)) & \mathrel{\hat{=}} \texttt{<a>} \,.\, f(x) \,.\, \texttt{<b>}
\end{array}$$

and the corresponding context-free grammar

$$\begin{array}{l}
T_f \rightarrow \varepsilon \\
T_f \rightarrow \texttt{<a>} \,.\, T_f \,.\, \texttt{<b>}.
\end{array}$$

This context-freeness of the range is not problematic for type checking because for type R described by an RHG and type C obtained by Maneth et al.'s algorithm, $C \subseteq R$ is known to be decidable while $R \subseteq C$ is not [23].

Note that a nondeterministic program is sufficient for type inference. Without a guarantee that the specialized programs are deterministic, we can provide another version of specialization that runs faster than that with a guarantee. The other version is obtained by removing Step 3 in the specialization algorithm, which constructs unambiguous RHGs. The other version also terminates because only the operations on types in the other version of specialization are taking products and switching chopping nonterminals. We have not given any formal proof of termination of the other version because it is beyond the scope of this paper. An upper bound for the program growth by the other version of specialization is $\mathrm{O}(2^{n^2})$, where n is the number of nonterminals in G of a program, $\mathcal{P} = (G, \_)$. The difference in complexities between the other version of our specialization and the pre-processing used in [23] results from the difference in patterns: pattern $x :: T$ introduces a power of 2^{n^2} and pattern $p_1 \,.\, p_2$ introduces a square of n^2.

5.2 Injectivity Analysis

In addition to the determinism of a program, the exact types of expressions for a specialized program can be obtained as previously noted. We can then adopt the injectivity analysis used in XSugar [3, 4] with slight modifications.

The nondeterminism of a program may make injectivity analysis more difficult. If two rules of function f in a deterministic specialized program, have range-overlapping expressions e_1, e_2

$$\begin{array}{l}
f(p_1) \mathrel{\hat{=}} e_1 \\
f(p_2) \mathrel{\hat{=}} e_2
\end{array}$$

then f is non-injective because (1) the deterministic property ensures that a set $[\![p_1]\!]$ of values matching p_1 and a set $[\![p_2]\!]$ of values matching p_2 are disjoint, and (2) Theorem 4 ensures that $\exists v.\ e\theta \Downarrow v$ implies $p\theta \in [\![p]\!]$ for any $f(p) \mathrel{\hat{=}} e$. However, in a nondeterministic program there is an injective function in which the range of right-hand side expressions of two rules overlap. For example, the following injective nondeterministic function has range-overlapping right-hand side expressions.

$$\begin{array}{l}
f(\texttt{<true>}) \mathrel{\hat{=}} \texttt{<true>} \\
f(\texttt{<true>}) \mathrel{\hat{=}} \texttt{<true>}
\end{array}$$

Consequently, it is more difficult to precisely analyze injectivity for nondeterministic programs than for those that are deterministic.

Also, recall that there is a case where, while generic function $f :: A \rightarrow B$ is not injective, specific function $f|_{A'}$ with $A' \subseteq A$ is injective, as was the *unifyAddress* in the Introduction. Since we have obtained a program for $f|_{A'}$ after specialization, we can analysis injectivity more precisely.

There are only three types of non-injective programs in our specialized program, which is simply proved using induction. The first type of program is one that does not use a variable with a type whose cardinality is more than one on some right-hand sides, as $f(x :: \texttt{<a>}^* \,.\, y :: \texttt{<b>}^*) \mathrel{\hat{=}} x$. The second type of program is one that concatenates two expressions whose exact types T_1, T_2 satisfy $[\![T_1]\!] \not\parallel [\![T_2]\!]$ as $f(x :: \texttt{<a>}^* \,.\, y :: \texttt{<b>}^* \,.\, z :: \texttt{<a>}^*) \mathrel{\hat{=}} x \,.\, z \,.\, y$. The third type of program is one that contains two rules of a function in which right-hand-side expression ranges overlap as $f(\texttt{<true>}) \mathrel{\hat{=}} \texttt{<true>}; f(\texttt{<false>}) \mathrel{\hat{=}} \texttt{<true>}$. In addition, we must also examine whether or not such a non-injective function above will be called from the function whose injectivity one wants to check. The specialization also simplifies this process; after specialization, a syntactically-called function, i.e., a function that appears on the right-hand side, is semantically called with some input, i.e., a function is used in evaluation for the input.

Hence, injectivity analysis can be achieved by examining existence of the three places above. However, it is known that, with exact types of expressions, checking whether two ranges overlap or not is undecidable [3]. We must approximate the exactly-inferred types as in XSugar [3, 4], where Mohri and Nederhof's regular approximation algorithm of context-free grammars [26] is used. To use the algorithm for programs in our target language, we must modify it slightly because the original algorithm is for strings but not for hedges. Using a hedge version of their regular approximation algorithm, we can analyze the injectivity of a function written in our target language. To do this, we approximate the types of function calls and then determine the types of other expressions according to approximated type for the later inversion step. This ensures that the approximated type of expression and the pattern generated from the expression in the later naive inversion will coincide. Note that context-freeness is only caused by the function-call structure of a program.

Note that there is no exact algorithm for analyzing injectivity for our target language, which can be shown by reducing the problem to Post's Correspondence Problem [31].

5.3 Inversion

With specialization, we can perform correct inversion by naively swapping left-hand sides with right-hand sides. In contrast, without specialization, this naive inversion produces incorrect results. For example, for function g defined by

$$g(x :: A') \mathrel{\hat{=}} f(x)$$

with $f :: A \to B$ and $A' \subset A$, even when f is injective, naive inversion would produce the following inverse.

$$g^{-1}(v) \mathrel{\hat{=}} f^{-1}(v)$$

Function g^{-1} is problematic because f^{-1} may return a value that does not belong to A', i.e., the domain of g. We can avoid the problem with specialization, because the exact range of function $f|_{A'}$ is known.

Approximated types, i.e., estimated ranges of functions, also work properly as long as the injectivity analysis discussed above determines a program is injective. For such a program, naively swapping left-hand sides with right-hand sides produces a correct inverse program in our target language because variable pattern $x :: T$ where T is defined by an RHG is permitted in our target program. For example, the role of the types of function calls is clear in the following inverse of the function, $c2x$, presented earlier in this section.

$$\begin{array}{l}
\mathbf{data}\, T_1 \mathrel{\hat{=}} (\texttt{<h2>}(\mathit{String}) \,.\, P)^* \\
\mathbf{data}\, T_2 \mathrel{\hat{=}} (\texttt{<h1>}(\mathit{String}) \,.\, P \,.\, (\texttt{<h2>}(\mathit{String}) \,.\, P)^*)^* \\
\mathbf{data}\, T_3 \mathrel{\hat{=}} T_1 \\
c2x^{-1}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
c2x^{-1}(\texttt{<h1>}(t :: \mathit{String}) \,.\, p :: P \,.\, v_1 :: T_1 \,.\, v_2 :: T_2) \\
\quad \mathrel{\hat{=}} \texttt{<chapter>}(\texttt{<title>}(t) \,.\, p \,.\, s2x^{-1}(v_1)) \,.\, c2x^{-1}(v_2) \\
s2x^{-1}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
s2x^{-1}(\texttt{<h2>}(t :: \mathit{String}) \,.\, p :: P \,.\, v_3 :: T_3) \\
\quad \mathrel{\hat{=}} \texttt{<section>}(\texttt{<title>} \,.\, p) \,.\, s2x^{-1}(v_3)
\end{array}$$

Here, T_1, T_2, and T_3 are the types of function calls $s2x(s)$ and $c2x(r)$ in the second rule of $c2x$, and $s2x(x)$ in the second rule of $s2x$, respectively. For example, the following inverse of cf discussed in Section 4 demonstrates that naive inversion works properly for functions of which the domain and the range are beyond regular hedge language.

$$\begin{array}{ll}
cf^{-1}(x :: S) \mathrel{\hat{=}} g|_T^{-1}(x) & \\
g|_T^{-1}(\varepsilon) & \mathrel{\hat{=}} \varepsilon \\
g|_T^{-1}(\texttt{<c>} \,.\, v :: S \,.\, \texttt{<d>}) & \mathrel{\hat{=}} \texttt{<a>} \,.\, g|_T^{-1}(v) \,.\, \texttt{<b>}
\end{array}$$

Here, $S = (\texttt{<c>}|\texttt{<d>})^*$ is an approximated type of the output type of $g|_T$.

Injectivity analysis on approximated types guarantees that the obtained inverses are deterministic, and in the same class as the original programs. It is common to use the results of injectivity analysis to obtain deterministic inverses [9, 12, 13, 14].

6. Extensions

To simplify our presentation, we have restricted each variable occurring on the left-hand side to occur at most once on the corresponding right-hand side and the number of parameters for every function to one. Relaxing this restriction is straightforward. Essentially, the properties of the specialization algorithm (Theorems 2, 4 and 5) only depend on the characteristics of a program where no output of a function can be examined by a pattern. For example, accumulation parameters as in macro forest transducers [30] can be introduced without violating the properties. Then, Theorem 4 changes slightly; variables passed to the accumulation parameters of a function must be treated in the same way as variables in variable expressions, i.e., the types of such variables must be considered in type inference. Multi-return [17] can also be permitted, for which some program analyses might be easier.

Attributes in XML can be permitted because they can be encoded into RHGs. However, naively-encoding XML attributes into RHGs causes an explosion in the number of nonterminals in the resulting RHG; an RHG generating n-different XML attributes contains more than 2^n nonterminals because it must distinguish which attributes have appeared. Note that the XML attributes of an element appear in any order but they all appear exactly once. A discussion of record types [5] might make the specialization that supports XML attributes more efficient.

7. Conclusion

We proposed a method of program transformation that generates specialized function definitions from the use of a function, i.e., a function-call expression. The method of specialization always terminates and generates deterministic programs. In the proofs of properties of the specialization, we used the characteristics of our target program where no output of a function cloud be examined by a pattern. We demonstrated the effectiveness of the new specialization using many applications: type inference/check, injectivity analysis, and inversion.

We intend to explore all other applications of the proposed method of specialization. We especially think we will use the new method in bidirectionalization [24] where precise injectivity analysis plays an important role. Although the new technique of specialization does not improve efficiency at all, we believe that type-based optimization techniques [10, 19] would work effectively for specialized programs because the specialization would enable us to infer the precise input and output types for all functions.

Acknowledgements

This work is partially supported by the Japan Society for the Promotion of Science, Grant-in-Aid for JSPS Fellows 20-9584, 2008 and Grant-in-Aid for Scientific Research (A) 19200002, the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from National Institute of Informatics, and a grant from National Natural Science Foundation of China under Grant No. 60528006.

We wish to thank Keisuke Nakano who gave us the impetus for this research, and Akimasa Morihata who gave us some valuable comments on an earlier version of the paper. We are also grateful to the anonymous referee who pointed out mistakes in the old proofs of our main results.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03: Proceedings of the 2003 ACM SIGPLAN international conference on Functional programming*, pages 51–63, 2003.

[2] R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, 1998.

[3] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. In *Proceedings of 12th International Conference on Implementation and Application of Automata, CIAA '07*, volume 4783 of *LNCS*. Springer-Verlag, July 2007.

[4] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4), June 2008. Earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer-Verlag LNCS vol. 3774.

[5] P. Bunneman and B. C. Pierce. Union types for seminstrcutred data. Technical Report MS-CIS-99-09, University of Pennsylvania Department of Computer and Information Science, 1999.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1st 2002.

[7] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[8] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9):613–698, 2003.

[9] D. Eppstein. A heuristic approach to program inversion. In *International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 219–221, 1985.

[10] A. Frisch. Regular tree language recognition with static information. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004)*, pages 661–674, 2004.

[11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146, Washington, DC, USA, 2002. IEEE Computer Society.

[12] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In *FLOPS '04: The Seventh International Symposium on Functional and Logic Programming*, pages 291–306, 2004.

[13] R. Glück and M. Kawabe. Revisiting an automatic program inverter for lisp. *SIGPLAN Notices*, 40(5):8–17, 2005.

[14] D. Gries. *The Science of Programming*, Chapter 21 Inverting Programs. Springer, 1981.

[15] H. Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[16] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

[17] K. Inaba, H. Hosoya, and S. Maneth. Multi-return macro tree transducers. In *Proceedings of 13th International Conference on Implementation and Application of Automata, CIAA '08*, pages 102–111, 2008.

[18] M. Y. Levin. *Run, Xtatic, Run: Efficient Implementation of an Object-Oriented Language with Regular Pattern Matching*. PhD thesis, University of Pennsylvania, 2005.

[19] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *DBPL '05: Proceedings of 10th International Symposium on Database Programming Languages*, pages 184–198, 2005.

[20] S. Maneth. The macro tree transducer hierarchy collapses for functions of linear size increase. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, pages 326–337, 2003.

[21] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 283–294, 2005.

[22] S. Maneth and K. Nakano. XML type checking for macro tree transducers with holes. In *PLAN-X '08: Programming Languages Technologies for XML*, 2008.

[23] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT '07: Proceedings of the 11th International Conference of Database Theory*, volume 4353 of *LNCS*, pages 254–268, 2007.

[24] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 47–58, New York, NY, USA, 2007. ACM.

[25] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[26] M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord, editors, *Robusteness in Language and Speech Technology*. Kluwer Academic Publishers, The Netherlands, 2001.

[27] Murata. Hedge automata: a formal model for XML schemata, 1999. Available on: `http://www.xml.gr.jp/relax/hedge_nice.html`.

[28] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005.

[29] N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse term rewriting systems for pure treeless functions. In Y. Toyama, editor, *Proceedings of the International Workshop on Rewriting in Proof and Computation (RPC'01)*, pages 188–198, 10 2001.

[30] T. Perst and H. Seidl. Macro forest transducers. *Information Processsing Letters*, 89(3):141–149, 2004.

[31] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[32] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.