MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Type Specialization for Effective Bidirectionalization

Kazutaka MATSUDA, Zhenjiang HU,
and Masato TAKEICHI

METR 2008–29 July 2008

Type Specialization for Effective Bidirectionalization

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

A *bidirectional transformation* is a pair of transformations, a *forward transformation* and a *backward transformation*, where a forward transformation maps one data structure called source to another called view, and a corresponding backward transformation reflects changes on the view to the source. Its practical applications include replicated data synchronization, presentation-oriented editor development, tracing software development, and document format conversion.

It is, however, difficult to develop bidirectional transformations, because the forward and backward mappings must satisfy the bidirectional properties for consistency. It is even more difficult if we want to obtain "better" bidirectional transformations with, for example, clearer consistent semantics between sources and views and more updates on views. To resolve this problem, a program transformation named *bidirectionalization* is proposed, in which a useful backward transformation can be derived automatically from a given forward transformation based on derivation of a complementary function. However, the language there for describing forward transformations is still too restrictive to write many practical transformations.

In this paper, we relax the restrictions on the previous language by supporting forest concatenation and look-ahead mechanism specified by regular expression types, which allows us to write practical transformations. In the language, a program transformation named *type specialization* not only enables us to obtain "better" backward transformations but also provides exact type checking of transformations. Our new approach has been implemented, and the experimental results show our approach is promising.

1 Introduction

There are many situations in which one data structure, called *source*, is transformed to another, called *view*, in such a way that changes on the view can be reflected to the source. This is called *bidirectional transformation* (Foster et al. 2005), and practical examples include synchronization of replicated data in different formats (Foster et al. 2005; Bohannon et al. 2008), presentation-oriented structured document development (Hu et al. 2004), interactive user interface design (Meertens 1998), coupled software transformation (Lämmel 2004), and the well-known *view updating* mechanism which has been intensively studied in the database community (Bancilhon and Spyratos 1981; Gottlob et al. 1988; Lechtenbörger and Vossen 2003).

As a simple example, consider that we have a member list consisting of students and professors.

```
<mems>
  <std>Metsuda</std>
  <prf>Hu</prf>
  <prf>Takeichi</prf>
</mems>
```

On one hand, we want to extract all the students out. This transformation can be realized by the following forward transformation *students*[1]:

$$\begin{array}{ll}
\mathit{students}(\texttt{<mems>}(x)) & \hat{=}\ \texttt{<mems>}(f(x)) \\
\quad f(\varepsilon) & \hat{=}\ \varepsilon \\
\quad f(\texttt{<std>}(x)\,\centerdot\,r) & \hat{=}\ \texttt{<std>}(x)\,\centerdot\,f(r) \\
\quad f(\texttt{<prf>}(x)\,\centerdot\,r) & \hat{=}\ f(r)
\end{array}$$

where function f keeps students and ignores professors. Applying the forward transformation *student* yields the following view.

`<mems><std>Metsuda</std></mems>`

On the other hand, we want to change the view and wish to reflect the change to the source. For example, we find that the student name `Metsuda` is wrong and want to change the name to `Matsuda` on the view and propagate this correction to the source. This can be realized by the following backward transformation of *students*:

$$\begin{array}{ll}
\mathit{student}_{\mathrm{B}}(\texttt{<mems>}(s), \texttt{<mems>}(v)) & \hat{=}\ \texttt{<mems>}(f_{\mathrm{B}}(s,v)) \\
\quad f_{\mathrm{B}}(\varepsilon,\varepsilon) & \hat{=}\ \varepsilon \\
\quad f_{\mathrm{B}}(\texttt{<std>}(x)\,\centerdot\,r, \texttt{<std>}(x')\,\centerdot\,r') & \hat{=}\ \texttt{<std>}(x')\,\centerdot\,f_{\mathrm{B}}(r,r') \\
\quad f_{\mathrm{B}}(\texttt{<prf>}(x)\,\centerdot\,r, r') & \hat{=}\ \texttt{<prf>}(x)\,\centerdot\,f_{\mathrm{B}}(r,r')
\end{array}$$

where f_{B} accepts the old source and the revised view as input, carefully compares them, and produces a new source as the result.

It is, however, hard in practice to manually write both forward and backward transformations while guaranteeing that the two transformations really form a bidirectional transformation and satisfy consistency properties (Bancilhon and Spyratos 1981). This has led to a large amount of work on *bidirectionalization*, automatic derivation of backward transformations from forward transformations. There are basically two approaches to bidirectionalization. The first approach is to design a set of general *combinators* (Foster et al. 2005; Bohannon et al. 2008; Hu et al. 2004; Meertens 1998) for constructing bigger bidirectional transformations by *composing* smaller ones. A set of primitive bidirectional transformations, each being defined by a pair of forward and backward transformation, is prepared, and a new bidirectional transformation is defined by assembling the primitive transformations with a fixed set of general combinators. This approach has proved to be practically useful for domain-specific applications, because primitive bidirectional transformations for a specific application are easily determined, designed, and implemented. However, for an involved application or in a more general setting, many primitive bidirectional transformations may need to be prepared, and it is still hard to verify whether a pair consisting of a view function and a backward transformation forms a (primitive) bidirectional transformation.

The second approach, aiming to solve the above problem, is to design a language so that for any forward transformation specified in this language a backward transformation can be automatically derived. One such language has been proposed by Matsuda et al. (2007), where backward transformations can be fully automatically derived from forward transformation description based on complement function derivation (Bancilhon and Spyratos 1981). The derived backward transformations are "good" in the sense that they not only satisfy the nice bidirectional properties (Section 3), but also reflect many view changes by the proposed optimizations of the derived backward transformations and provide information about what kind of changes on the view are allowed.

Though being ideal and attractive, the second approach has two major limitations. The first limitation is that it can only work well with a first-order functional language with two syntactic restrictions: *affine* (no variable is used more than once) and *treeless* (arguments of a function call cannot be a function call again). This restriction of "treeless" would prevent it from being used to describe many practical transformations as seen below.

[1]Here, "$\centerdot$" represents forest concatenation, ε denotes the empty forest, and `<tag>`(x) is a shorthand for `<tag>`x`</tag>`.

Example 1 (Chapters to XHTML)**.** Consider the transformation[2] from a sequence of chapters:

```
<chp><t>chapterTitle_1</t>
     <p>para_1</p><p>para_2</p>
     <sec><t>sectionTitle_2</t>
          <p>para_3</p></sec></chp>
<chp><t>chapterTitle_3</t><p>para_4</p></chp>
```

of the type

$$\begin{array}{ll}\textbf{data}\ \mathit{Cs} & \hat{=}\ (\texttt{<chp>}(\texttt{<t>}(\mathit{String})\ \textbf{.}\ P^*\ \textbf{.}\ S^*))^* \\ \textbf{data}\ P & \hat{=}\ \texttt{<p>}(\mathit{String}) \\ \textbf{data}\ S & \hat{=}\ \texttt{<sec>}(\texttt{<t>}(\mathit{String})\ \textbf{.}\ P^*)\end{array}$$

to the following XHTML fragment.

```
<h1>chapterTitle_1</h1>
    <p>para_1</p><p>para_2</p>
    <h2>sectionTitle_2</h2>
        <p>para_3</p>
<h1>chapterTitle_3</h1><p>para_4</p>
```

The main difficulty in writing this transformation in the proposed language is that we have to concatenate (glue) pieces of transformed data to form the final result, which is not allowed due to the treeless restriction, because the function concatenation is not permitted to be applied to the results produced by other transformations. This concatenation, a simple gluing of the transformed results, has known to be very important and carefully treated in XML uni-transformation (Hosoya and Pierce 2003). □

The second limitation is the assumption that only one backward transformation is derived for a forward transformation, no matter how many times and wherever the forward transformation is called in a transformation program. This means that derivation of backward transformations does not consider the use context of the forward transformation. In fact, a suitable context description and a mechanism of context propagation would lead to a better backward transformation so that more view changes can be reflected in the source, which is clear from the following example.

Example 2 (Table of Contents)**.** Consider obtaining a table-of-contents from the sequence of chapters in Example 1 with all paragraphs being removed.

```
<h1>chapterTitle_1</h1>
    <h2>sectionTitle_2</h2>
<h1>chapterTitle_3</h1>
```

This transformation is not injective in general; a modification on the view may correspond to many ways of modification on the source. So, not arbitrary modification on the view is allowed because of the "constant" property of complement function in the derivation of backward transformation (Section 3). However, if we know that this transformation is used in the place where the source will contain no paragraph, then the transformation there becomes injective and arbitrary modification on the view produced is allowed. □

This paper is about two important extensions of the work (Matsuda et al. 2007), which are aimed to relax the above two limitations. We extend the language with the forest concatenation operation, and introduce regular types for "look-ahead" in transformation and for context propagation. These extensions, as will be seen later, allow users to specify more useful forward transformations with more precise information for later

[2]A simplified version of a transformation in Appendix D of XSLT specification: `http://www.w3.org/TR/xslt.html#section-Examples`

bidirectionalization. For instance, the (forward) transformations for Examples 1 and 2, which cannot be specified before, can be specified respectively by

$$\begin{array}{l} \mathit{c2x}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\ \mathit{c2x}(\texttt{<chp>}(\texttt{<t>}(t) \,.\, x :: P^* \,.\, y :: S^*) \,.\, r :: \mathit{Cs}) \\ \quad \mathrel{\hat{=}} \texttt{<h1>}(t) \,.\, x \,.\, \mathit{s2x}(y) \,.\, \mathit{c2x}(r) \\ \mathit{s2x}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\ \mathit{s2x}(\texttt{<sec>}(\texttt{<t>}(t) \,.\, x :: P^*) \,.\, r :: S^*) \mathrel{\hat{=}} \texttt{<h2>}(t) \,.\, x \,.\, \mathit{s2x}(r) \end{array}$$

and

$$\begin{array}{l} \mathit{toc}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\ \mathit{toc}(\texttt{<chp>}(\texttt{<t>}(t) \,.\, x :: P^* \,.\, y :: S^*) \,.\, r :: \mathit{Cs}) \\ \quad \mathrel{\hat{=}} \texttt{<h1>}(t) \,.\, \mathit{sc}(y) \,.\, \mathit{toc}(r) \\ \mathit{sc}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\ \mathit{sc}(\texttt{<sec>}(\texttt{<t>}(t) \,.\, x :: P^*) \,.\, r :: S^*) \mathrel{\hat{=}} \texttt{<h2>}(t) \,.\, \mathit{sc}(r) \end{array}$$

where $x :: P^*$ denotes a variable x with a regular type P^* for "look-ahead" and context propagation, and "." represents forest concatenation.

It should be noted that although forest concatenation and look-ahead by regular expression types are common and not surprising for a uni-directional transformation language (Hosoya and Pierce 2003; Benzaken et al. 2003), they offer a challenge and introduce many new problems for automatic bidirectionalization based on derivation of complement functions (Matsuda et al. 2007).

The first problem is the difficulty in estimating ranges (range inference). The estimation of the range is significant in bidirectionalization because it plays an important role in the detection of injective functions and construction of effective complement functions. The range of function call expressions may differ depending on their called contexts. For example, the ranges of the following two calls of *id* differ, where $\mathit{id}(x) \mathrel{\hat{=}} x$.

$$f(x :: \texttt{<a>}, y :: \texttt{<a>}^*) \mathrel{\hat{=}} \mathit{id}(x) \,.\, \mathit{id}(y).$$

Besides, as seen in the above transformation examples, we allow non-tail variables in patterns and can bind a variable to a forest instead of a tree, which makes the exact range inference difficult (Hosoya 2003). The second problem is the bidirectionalization steps become more complicated. For example, it is not clear how to derive a backward transformation for the following program.

$$f(x, y) \mathrel{\hat{=}} h(x) \,.\, y;\ \ h(x) \mathrel{\hat{=}} \ldots$$

What is a small complement function of f so that it can be combined with f to form an injective transformation? One solution is to store the length of $h(x)$ or y and give a constant complement function as follows.

$$f^{\mathrm{c}}(x, y) \mathrel{\hat{=}} \ldots \textsc{len}(h(x)) \ldots;\ \ h(x) \mathrel{\hat{=}} \ldots$$

However, this introduces function composition in complement functions, which makes the inverse calculation step difficult. The third problem, which is important in practice, is how to represent and derive a useful view update checker in the context where the regular types are used, so that users can understand what kind of change on the view can be reflected in the source.

In this paper, we show that all these new problems can be successfully resolved by extending the previous work (Matsuda et al. 2007) with more involved range analysis. In particular, we show that program transformation called *type specialization* plays an important role in obtaining more effective bidirectionalization of a wide class of forward transformations. Our main contributions can be summarized as follows.

- *More expressive forward transformations can be bidirectionalized.* Thanks to forest concatenation and look-ahead with regular expression type, our new language can be used to describe various kinds of useful transformations, including useful apply-to-all (map) and filter-like functions, the core part of XSugar (Brabrand et al. 2008) transformations, and many transformations written in biXid (Kawanaka and Hosoya 2006) except transformations having horizontally unzipping operations. It will be shown that all the transformations in this language can be fully automatically bidirectionalized and the derived backward transformations are always deterministic.

- *Type specialization realizes exact rang inference.* We recognize that exact types of function calls, i.e., ranges of functions, can be obtained by taking into account of the context where the function calls are, which are described by regular expression types (from look-ahead). We propose an *exact* range inference based on a program transformation named *type specialization*, and indicate that the type specialization terminates, and leads to effective bidirectionalization.
- *View update checker is automatically generated.* We show that an exact view update checker can be automatically derived with type specialization, so that users can know whether or not a view change is reflectable without actual execution of the backward transformation. This is practically useful, because a backward transformation usually cannot reflect all the view changes to the source because of the consistency between sources and views. Moreover, we demonstrate that the set of all the reflectable view changes can be described by regular expression types with a little reasonable syntactic restriction.

All the algorithms have been implemented, and a bidirectionalization system is available at the following URL.

`http://www.ipl.t.u-tokyo.ac.jp/~kztk/b18nf/`

Most of the proofs of the theorems in this paper are shown in Appendix.

The rest of this paper is organized as follows. In Section 2, we show some examples of our bidirectionalization to give some flavor of what our system can do. In Section 3, we review our previous work and explain the problem of the previous work. In Section 4, we define the target language, an extended language for specifying forward transformations. In Section 5, we discuss type specialization, range inference and type checking, which play an important role in our bidirectionalization system discussed in Section 6. We discuss the related work in Section 7, and conclude the paper in Section 8.

2 Examples of Bidirectionalization

Before explaining our language extension with the forest concatenation and the regular expression types for look-ahead, and the details of bidirectionalization for derivation of backward transformations, we give a short demonstration of what our system can do for the two examples in Introduction. More examples are available at the system web site (given in Introduction).

2.1 Example: Chapters to XHTML

Recall Example 1 in Introduction, which is to transform a sequence of chapters to an XHTML fragment. From the forward transformation specified by $c2x$, our system can automatically determine that this forward transformation is injective, and return the following backward transformation $c2x_{\mathrm{B}}$ that accepts an old source s and a revised view v and returns a new source.

$$\begin{array}{l}
\textbf{data}\ Q \mathrel{\hat{=}} \texttt{<h1>}(\mathit{String}) \centerdot P^{*} \centerdot (\texttt{<h2>}(\mathit{String}) \centerdot P^{*})^{*} \\
c2x_{\mathrm{B}}(s, v) \mathrel{\hat{=}} c2x^{-1}(v) \\
c2x^{-1}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
c2x^{-1}(\texttt{<h1>}(t) \centerdot x :: P^{*} \centerdot y :: (\texttt{<h2>}(\mathit{String}) \centerdot P^{*})^{*} \centerdot r :: Q^{*}) \\
\quad \mathrel{\hat{=}} \texttt{<chp>}(\texttt{<t>}(t) \centerdot x \centerdot s2x^{-1}(y)) \centerdot c2x^{-1}(r) \\
s2x^{-1}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
s2x^{-1}(\texttt{<h2>}(t) \centerdot x :: P^{*} \centerdot r :: S^{*}) \\
\quad \mathrel{\hat{=}} \texttt{<sec>}(\texttt{<t>}(t) \centerdot x) \centerdot s2x^{-1}(r)
\end{array}$$

The inverse of a forward transformation $c2x$ is the best backward transformation because this backward transformation can reflect any view change to the source, the reflection is independent of the reflection history, and any reflected source changes can be canceled by some view change. In addition to the above backward transformation, our system also returns a view update checker showing what kind of change on the views can be reflected to the source.

Now, consider a simple variation of the forward transformation, where we transformation each section title to an `<h1>` element instead of an `<h2>` element. The forward transformation becomes

$$\begin{aligned}
&\mathit{c2y}(\varepsilon) \mathrel{\hat{=}} \varepsilon\\
&\mathit{c2y}(\texttt{<chp>}(\texttt{<t>}(t) \centerdot x :: P^* \centerdot y :: S^*) \centerdot r :: \mathit{Cs})\\
&\qquad \mathrel{\hat{=}} \texttt{<h1>}(t) \centerdot x \centerdot \mathit{s2x}(y) \centerdot \mathit{c2y}(r)\\
&\mathit{s2y}(\varepsilon) \mathrel{\hat{=}} \varepsilon\\
&\mathit{s2y}(\texttt{<sec>}(\texttt{<t>}(t) \centerdot x :: P^*) \centerdot r :: S^*)\\
&\qquad \mathrel{\hat{=}} \underline{\texttt{<h1>}(t)} \centerdot x \centerdot \mathit{s2y}(r)
\end{aligned}$$

where only underlined part is changed. By injectivity checking, our system knows that this transformation is not injective, and derives a complement function for $\mathit{c2y}^{\mathrm{c}}$ so that tupling of $\mathit{c2y}$ and $\mathit{c2y}^{\mathrm{c}}$ forms an injective transformation[3].

$$\begin{aligned}
&\mathit{c2y}^{\mathrm{c}}(\varepsilon) \mathrel{\hat{=}} \mathsf{R}_1\\
&\mathit{c2y}^{\mathrm{c}}(\texttt{<chp>}(\texttt{<t>}(t) \centerdot x :: P^* \centerdot y :: S^*) \centerdot r :: \mathit{Cs})\\
&\qquad \mathrel{\hat{=}} \mathsf{R}_2\langle \textsc{len}(\mathit{s2y}(y)), \mathit{c2y}^{\mathrm{c}}(r)\rangle
\end{aligned}$$

Here, R_1 and R_2 are two new data constructors, R_2 has two arguments inside $\langle\rangle$, and LEN is to compute the length of a forest. The function $\mathit{c2y}^{\mathrm{c}}$ is basically to remember the length of all transformed sections in each chapter. Finally, our system automatically derives the following backward transformation $\mathit{c2y}_{\mathrm{B}}$.

$$\begin{aligned}
&\textbf{data } H \mathrel{\hat{=}} \texttt{<h1>}(\mathit{String}) \centerdot P^*\\
&\mathit{c2y}_{\mathrm{B}}(s, v) \mathrel{\hat{=}} \langle \mathit{c2y}, \mathit{c2y}^{\mathrm{c}}\rangle^{-1}(v, \mathit{c2y}^{\mathrm{c}}(s))\\
&\langle \mathit{c2y}, \mathit{c2y}^{\mathrm{c}}\rangle^{-1}(\varepsilon, \mathsf{R}_1) \mathrel{\hat{=}} \varepsilon\\
&\langle \mathit{c2y}, \mathit{c2y}^{\mathrm{c}}\rangle^{-1}(v_1, \mathsf{R}_2\langle l_1, w_4\rangle) \mathrel{\hat{=}} \texttt{<chp>}(\texttt{<t>}(t) \centerdot x \centerdot y) \centerdot r\\
&\quad \textbf{where}\\
&\qquad (\texttt{<h1>}(t) \centerdot x :: P^* \centerdot w_3 :: H^*, w_2) \mathrel{\hat{=}} \textsc{spl}(l_1, v_1)\\
&\qquad r :: \mathit{Cs} \mathrel{\hat{=}} \langle \mathit{c2y}, \mathit{c2y}^{\mathrm{c}}\rangle^{-1}(w_2, w_4)\\
&\qquad y :: S^* \mathrel{\hat{=}} \mathit{s2y}^{-1}(w_3)\\
&\mathit{s2y}^{-1}(\varepsilon) \mathrel{\hat{=}} \varepsilon\\
&\mathit{s2y}^{-1}(\texttt{<h1>}(t) \centerdot x :: P^* \centerdot r :: H^*)\\
&\qquad \mathrel{\hat{=}} \texttt{<sec>}(\texttt{<t>}(t) \centerdot x) \centerdot \mathit{s2y}^{-1}(r)
\end{aligned}$$

Here, the function $\textsc{spl}(f, l)$ splits a forest f to (f_1, f_2) such that $f = f_1 \centerdot f_2$ where the length of f_1 is l. Note that this backward transformation may be a bit difficult to read because of complicated function names and many subscripts and superscripts. We prefer this because it can show the correspondence of functions among derivation steps. If we rename $\langle \mathit{c2y}, \mathit{c2y}^{\mathrm{c}}\rangle^{-1}$ to $\mathit{y2c}$, then the above program is as follows.

$$\begin{aligned}
&\mathit{c2y}_{\mathrm{B}}(s, v) \mathrel{\hat{=}} \mathit{y2c}(v, \mathit{c2y}^{\mathrm{c}}(s))\\
&\mathit{y2c}(\varepsilon, \mathsf{R}_1) \mathrel{\hat{=}} \varepsilon\\
&\mathit{y2c}(v_1, \mathsf{R}_2\langle l_1, w_4\rangle) \mathrel{\hat{=}} \texttt{<chp>}(\texttt{<t>}(t) \centerdot x \centerdot y :: S^*) \centerdot r\\
&\quad \textbf{where}\\
&\qquad (\texttt{<h1>}(t) \centerdot x :: P^* \centerdot w_2 :: H^*, w_3) \mathrel{\hat{=}} \textsc{spl}(l_1, v_1)\\
&\qquad r :: \mathit{Cs} \mathrel{\hat{=}} \mathit{y2c}(w_3, w_4)\\
&\qquad y :: S^* \mathrel{\hat{=}} \mathit{s2y}^{-1}(w_2)
\end{aligned}$$

Note that by this backward transformation any change on the view is allowed unless it does not change the length of a resulting forest between an `<h1>` corresponding to one chapter and another `<h1>` corresponding to another chapter.

[3]Note that $\mathit{s2y}$ is injective, so we need not compute its complement function.

2.2 Example: Table of Contents

This example is to show the use of type specialization. Recall Example 2 that computes a table of contents. Since *toc* is not injective by our injectivity checking, our system generates the following complement functions.

$$\begin{array}{l}
toc^{c}(\varepsilon) \mathrel{\hat{=}} \mathsf{R}_1 \\
toc^{c}(\texttt{<chp>}(\texttt{<t>}(t)\,\centerdot\, x :: P^*\,\centerdot\, y :: S^*)\,\centerdot\, r :: Cs) \\
\quad \mathrel{\hat{=}} \mathsf{R}_2\langle toc^{c}(r), sc^{c}(y), x\rangle \\
sc^{c}(\varepsilon) \mathrel{\hat{=}} \mathsf{R}_3 \\
sc^{c}(\texttt{<sec>}(\texttt{<t>}(t)\,\centerdot\, x :: P^*)\,\centerdot\, r :: S^*) \mathrel{\hat{=}} \mathsf{R}_4\langle sc^{c}(r), x\rangle
\end{array}$$

Then, the system generates the following backward transformation.

$$\begin{array}{l}
\mathbf{data}\ T \mathrel{\hat{=}} \texttt{<h1>}(String)\,\centerdot\, U^* \\
\mathbf{data}\ U \mathrel{\hat{=}} \texttt{<h2>}(String) \\
toc_{\mathrm{B}}(s, v) \mathrel{\hat{=}} \langle toc, toc^{c}\rangle^{-1}(v, toc^{c}(s)) \\
\langle toc, toc^{c}\rangle^{-1}(\varepsilon, \mathsf{R}_1) \mathrel{\hat{=}} \varepsilon \\
\langle toc, toc^{c}\rangle^{-1}(\texttt{<h1>}(t)\,\centerdot\, w_1 :: U^*\,\centerdot\, w_2 :: T^*, \mathsf{R}_2\langle w_3, w_4, x\rangle) \\
\quad \mathrel{\hat{=}} \texttt{<chp>}(\texttt{<t>}(t)\,\centerdot\, x\,\centerdot\, y)\,\centerdot\, r \\
\quad \mathbf{where}\ \ y :: S^* \mathrel{\hat{=}} \langle sc, sc^{c}\rangle^{-1}(w_1, w_3) \\
\qquad\qquad\ r :: Cs \mathrel{\hat{=}} \langle toc, toc^{c}\rangle^{-1}(w_2, w_4) \\
\langle sc, sc^{c}\rangle^{-1}(\varepsilon, \mathsf{R}_3) \mathrel{\hat{=}} \varepsilon \\
\langle sc, sc^{c}\rangle^{-1}(\texttt{<h2>}(t)\,\centerdot\, w_5, \mathsf{R}_4\langle w_6, x\rangle) \mathrel{\hat{=}} \texttt{<sec>}(\texttt{<t>}(t)\,\centerdot\, x)\,\centerdot\, r \\
\quad \mathbf{where}\ r \mathrel{\hat{=}} \langle sc, sc^{c}\rangle^{-1}(w_5, w_6)
\end{array}$$

This *toc* may be used in a different context, and our system can utilize the context information for better backward transformations allowing more view changes. Consider the following use of *toc*.

$$\begin{array}{l}
\mathbf{data}\ V \mathrel{\hat{=}} \texttt{<chp>}(\texttt{<t>}(String)\,\centerdot\, W^*) \\
\mathbf{data}\ W \mathrel{\hat{=}} \texttt{<sec>}(\texttt{<t>}(String)) \\
ctxt(x :: V^*) \mathrel{\hat{=}} toc(x)
\end{array}$$

In this case, our system will produce the type-specialized versions of *toc* and *sc* as follows.

$$\begin{array}{l}
ctxt(x :: V^*) \mathrel{\hat{=}} toc_{V^*}(x) \\
toc_{V^*}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
toc_{V^*}(\texttt{<chp>}(\texttt{<t>}(t)\,\centerdot\, y :: W^*)\,\centerdot\, r) \\
\quad \mathrel{\hat{=}} \texttt{<h1>}(t)\,\centerdot\, s_{W^*}(y)\,\centerdot\, toc_{V^*}(r) \\
sc_{W^*}(\varepsilon) \mathrel{\hat{=}} \varepsilon \\
sc_{W^*}(\texttt{<sec>}(\texttt{<t>}(t))\,\centerdot\, r) \mathrel{\hat{=}} \texttt{<h2>}(t)\,\centerdot\, sc_{W^*}(r)
\end{array}$$

Now *ctxt* and toc_{T^*} are injective and can be automatically bidirectionalized. The derived backward transformation is much better than that with direct use of toc_{B}.

3 Preliminaries: Bidirectionalization based on Derivation of Complement Functions

In this section, we briefly review the notations and basic concepts of the constant complement approach to bidirectionalization (Bancilhon and Spyratos 1981), and the previous work on automatic bidirectionalization based on automatic derivation of complement functions (Matsuda et al. 2007).

3.1 Notations

Our notations, if not explained, follow Haskell[4], a functional programming language. For a partial function f, we write $f(x)\downarrow$ if $f(x)$ is defined, and write $f(x) = \bot$ otherwise. For a function $f : X \to Y$ and a function $g : X \to Z$, we define a tupled function $\langle f, g\rangle : X \to (Y \times Z)$ by $\langle f, g\rangle(x) = (f(x), g(x))$. For a partial function $f : X \to Y$ and a partial function $g : X \to Y$, we write $f \sqsubseteq g$ to denote $\forall x \in X, f(x)\downarrow \Rightarrow f(x) = g(x)$. Intuitively, $f \sqsubseteq g$ means that g is more widely defined than f.

3.2 Bidirectional Transformation

A bidirectional transformation is a pair of two transformations: a forward transformation that maps one data structure *source* to another called *view*, and a corresponding backward transformation that reflects changes on the view to the source.

Definition 1 (Bidirectional Transformation)**.** A *forward transformation* $f : S \to V$ is a function. A *backward transformation* ρ of a forward transformation f is a function satisfying

$$\forall s \in S, \forall v \in V.\ \rho(s, v)\downarrow \Rightarrow f(\rho(s, v)) = v.$$

A backward transformation ρ of f translates an update $f(s) \rightarrowtail v$ to an update $s \rightarrowtail \rho(s, v)$, and the view of updated-reflected source, $f(\rho(s, v))$, is equal to the updated view, v. In other words, for a source $s \in S$ and its view $f(s) \in V$, let u be an update $f(s) \rightarrowtail v$ and u' a translated update $s \rightarrowtail \rho(s, v)$, then the following diagram commutes.

$$\begin{tikzcd} V \arrow[r, "u"] & V \\ S \arrow[u, "f"] \arrow[r, "u'"'] & S \arrow[u, "f"'] \end{tikzcd}$$

3.3 Constant Complement Bidirectionalization

The constant complement bidirectionalization (Bancilhon and Spyratos 1981) derives a backward transformation by generating a complementary function. The basic idea of their bidirectionalization is to produce an injective function because a backward transformation of an injective function can be defined its inverse. A complement function is introduced by Bancilhon and Spyratos (1981) to make injective functions. Formally, a *complement function* g of f is a function that makes $\langle f, g\rangle$ injective.

By a complement function g of f, a backward transformation is obtained by

$$f_{\mathrm{B}}(s, v) = \langle f, g\rangle^{-1}(v, g(s)).$$

For example, for a function *add* defined by $add(x, y) = x + y$ and a complement function $fst(x, y) = x$ of *add*, the obtained backward transformation is as follows.

$$add_{\mathrm{B}}((x, \_), v) = (x, v - x)$$

Since, as the above example, the return value of a complement function remains constant through a backward transformation step, the bidirectionalization is called *constant complement.* It is known that the obtained backward transformations satisfy the following good *bidirectional properties.*

Acceptability:
$\rho(s, f(s)) = s,$

Undoability:
$\rho(s, v)\downarrow \Rightarrow \rho(\rho(s, v), f(s)) = s,$

Composability:
$\rho(s, v)\downarrow \wedge \rho(\rho(s, v), v')\downarrow \Rightarrow \rho(\rho(s, v), v') = \rho(s, v').$

[4]Haskell 98 Report: `http://www.haskell.org/onlinereport/`

Acceptability means that if there is no change on views there should be no change on sources. Undoability means that all reflected updates can be canceled by updates on views. Composability means that reflected updates do not depend on the reflection history. What is also interesting is the backward transformation satisfies the above bidirectional properties has some complement function that yields the same backward transformation (Bancilhon and Spyratos 1981).

Generally, there are many complement functions for a function. For example, functions *fst*, $sub(x, y) = x - y$ and $id_{\mathrm{P}}(x, y) = (x, y)$ are all complement functions of *add*, and they yield different backward transformations.

$$\begin{array}{ll} add_{\mathrm{B}}^{fst}((s_1, \_), v) & = (s_1, v - s_1) \\ add_{\mathrm{B}}^{sub}((s_1, s_2), v) & = \left(\frac{v+(s_1-s_2)}{2}, \frac{v-(s_1-s_2)}{2}\right) \\ add_{\mathrm{B}}^{id_{\mathrm{P}}}((s_1, s_2), v) & = (s_1, s_2) \text{ if } v = add(s_1, s_2) \end{array}$$

Clearly, the above backward transformations have different semantics. While the backward transformations add_{B}^{fst} and add_{B}^{sub} are pairwise incomparable with respect to the order $\sqsubseteq$, the third backward transformation $add_{\mathrm{B}}^{id_{\mathrm{P}}}$ is the worst of the three with respect to $\sqsubseteq$. Bancilhon and Spyratos (1981) introduced the following preorder, under which smaller complement functions give more updatability.

Definition 2 (Collapsing Order)**.** Let $f : X \to Y$ and $g : X \to Z$ be functions. The *collapsing order*, $\precsim$, is a preorder defined by

$$f \precsim g \Leftrightarrow \forall x_1, x_2 \in X.\ g(x) = g(y) \Rightarrow f(x) = f(y)$$

Theorem 1 (Bancilhon and Spyratos (1981))**.** Let $f : S \to V$ be a forward transformation, and $g_1 : S \to V'$ and $g_2 : S \to V''$ be complement functions of f. If and only if $g_1 \precsim g_2$, then $f_{\mathrm{B}}^{g_2} \sqsubseteq f_{\mathrm{B}}^{g_1}$ holds.

Order $f \precsim g$ means that, with respect to the results of mappings, f collapses input more than g. Hence, all elements in the input collapse into one in the result by the minimal functions, i.e., constant functions, and nothing collapses by the maximal functions, i.e., the injective functions. For the above examples, id_{P} is greater than the others because it keeps the values of the input. The functions *fst* and *sub* are pairwise incomparable.

3.4 Automatic Bidirectionalization

Our previous bidirectionalization (Matsuda et al. 2007) targets first-order functional language with two syntactic restrictions: *affine* and *treeless* (Wadler 1990). Affine means no variable is used more than once and treeless means arguments of function calls must be variables. For example, the function *add* defined by

$$add(\mathsf{Z}, y) \mathrel{\hat{=}} y;\ \ add(\mathsf{S}(x), y) \mathrel{\hat{=}} \mathsf{S}(add(x, y))$$

is affine and treeless but the following functions are not.

$$copy(x) \mathrel{\hat{=}} (x, x) \quad twice(x) \mathrel{\hat{=}} g(g(x))$$

The bidirectionalization process (Matsuda et al. 2007) mainly consists of the following three steps.

1. Derivation of a complement function f^{c} for a given forward transformation f.
2. Calculation of $\langle f, f^{\mathrm{c}} \rangle^{-1}$.
3. Generation of a backward transformation by $f_{\mathrm{B}}(s, v) = \langle f, f^{\mathrm{c}} \rangle^{-1}(v, f^{\mathrm{c}}(s))$.

Since these steps will be addressed and generalized in Section 6, we omit the details here.

4 The Target Language

In this section, we describe the language to describe forward transformations from forests to forests. It is an extension of the language in Matsuda et al. (2007) with the concatenation and regular expression types for look-ahead.

4.1 Forest Values

A forest, or a hedge, is a finite concatenation of unranked-trees like XML fragments such as

`<std>Matsuda</std><prf>Hu</prf>`.

In our language, this forest is internally represented by

$$\texttt{<std>(M.a.t.s.u.d.a).<prf>(H.u)}.$$

Formally, a set $\mathcal{T}_{\Sigma,X}$ of *trees* generated by a set of labels Σ and a set of variables X is defined inductively as follows: $X \subseteq \mathcal{T}_{\Sigma,X}$, and if $t_1,\ldots,t_n \in \mathcal{T}_{\Sigma,X}$ then $\sigma(t_1 \mathbin{.} \ldots \mathbin{.} t_n) \in \mathcal{T}_{\Sigma,X}$ for any $\sigma \in \Sigma$. A set $\mathcal{T}_{\Sigma,X}{}^*$ of *forests* is defined by $\{t_1 \mathbin{.} \ldots \mathbin{.} t_n \mid n \in \mathbb{N}, t_i \in \mathcal{T}_{\Sigma,X}\}$. We assume that all the tree-labels and characters are encoded to the labels Σ. The empty forest is written by ε; we have $\varepsilon \mathbin{.} f = f \mathbin{.} \varepsilon = f$ for any forest f. We sometimes write a tree $\sigma(\varepsilon)$ by $\sigma()$ or by σ. Note that the forest concatenation $.$ is associative; $t_1 \mathbin{.} (t_2 \mathbin{.} t_3) = (t_1 \mathbin{.} t_2) \mathbin{.} t_3$ holds. For convenience, we sometimes omit $.$ and write t_1t_2 instead of $t_1 \mathbin{.} t_2$ if no confusion would arise. The set $\mathcal{T}_{\Sigma,\emptyset}$ is written as $\mathcal{T}_{\Sigma}$. A *context* $\mathcal{C}$ is a forest containing special hole variables $\square_1,\ldots,\square_n$; we write by $\mathcal{C}[f_1,\ldots,f_n]$ a forest that is obtained from $\mathcal{C}$ by replacing each variable $\square_i$ with f_i.

4.2 Types

In this language, we consider set-theoretic types (Frisch et al. 2002), defined by *forest grammars*. Two types of forest grammars are used in this paper: the *regular forest grammar* is used as "look-ahead" and for describing input and output types externally, and the *context-free forest grammar* is used for describing expression types internally.

As an example of regular forest grammars, a set of forest described by the regular expression $A = \texttt{<a>}^* \mid \texttt{<b>}$ is represented by the regular forest grammar $A \to A', A \to \texttt{<b>}, A' \to \varepsilon, A' \to \texttt{<a>}A'$. As in Section 2, they are used as look-ahead, and is called *regular expression types* in Hosoya and Pierce (2003). The context-free forest grammar is more expressive and can represent, for instance, a set $\{\texttt{<a>}^n\texttt{<b>}^n \mid n \in \mathbb{Z}\}$ by $T \to \varepsilon, T \to \texttt{<a>}T\texttt{<b>}$, which can be used to define the range of the following function f.

$$f(\varepsilon) \mathrel{\hat{=}} \varepsilon;\;\; f(\texttt{<c>}(x)) \mathrel{\hat{=}} \texttt{<a>} \mathbin{.} f(x) \mathbin{.} \texttt{<b>}$$

Definition 3 (Context-Free Forest Grammar)**.** A *context-free forest grammar* (CFFG, for short) is a 3-tuple (Σ, N, R) where Σ is a set of labels, N is a set of non-terminals, and R is a set of production rules of the form $T \to f$ where $T \in N$ and $f \in \mathcal{T}_{\Sigma,N}$.

Definition 4 (Regular Forest Grammar)**.** A *regular forest grammar* (RFG, for short) is a CFFG (Σ, N, R) where R only contains the rules of the form $T \to \sigma(T_1)T_2$ or $T \to \varepsilon$.

The definition of RFGs is the same as that of regular tree grammars (Comon et al. 1997) on binary tree encoding of forests (Hosoya and Pierce 2003), and the definition of CFFGs is the same as that of context-free sequence-tree automata (Ohsaki et al. 2003) but less expressive than that of context-free tree grammars (Comon et al. 1997) in its binary tree encoding. Note that the above definitions do not include the start non-terminals.

For a CFFG $G = (\Sigma, N, R)$, we may simply write $T \to f \in G$ or $T \in G$ instead of $T \to f \in R$ or $T \in N$, respectively. Moreover, we sometimes do not distinguish $G = (\Sigma, N, R)$ from R because Σ and N are usually clear from the context of R.

We write $A \xrightarrow{*} f$ if A generates a forest f. Semantics, or language, of a non-terminal A in G is defined by $[\![A]\!]_G = \{t \mid A \xrightarrow{*} t, t \in \mathcal{T}_\Sigma\}$. Languages of an RFG are called *regular forest language*.

Generally, deciding whether a language of CFFG is regular is known to be undecidable. However, there is a class of CFFG of which languages are regular (Mohri and Nederhof 2001).

Definition 5 (Mutually Defined Non-terminals)**.** For a CFFG $G = (\Sigma, N, R)$, non-terminals T and S are called *mutually defined* if $T \xrightarrow{*} \mathcal{C}[S]$ and $S \xrightarrow{*} \mathcal{C}'[T]$ for some contexts $\mathcal{C}$ and $\mathcal{C}'$.

Syntax:

$$
\begin{array}{lcl}
\mathit{prog} & ::= & \mathit{trule}_1 \ldots \mathit{trule}_n \ \mathit{frule}_1 \ldots \mathit{frule}_m \\
\mathit{trule} & ::= & \mathbf{data}\ T \mathrel{\hat{=}} t \\
t & ::= & \varepsilon \mid t_1 \centerdot t_2 \mid \sigma(t) \mid T \\
\mathit{frule} & ::= & f(p_1, \ldots, p_n) \mathrel{\hat{=}} e \\
e & ::= & \varepsilon \mid e_1 \centerdot e_2 \mid \sigma(e) \mid x \mid f(x_1, \ldots, x_n) \\
p & ::= & \varepsilon \mid p_1 \centerdot p_2 \mid \sigma(p) \mid x :: T
\end{array}
$$

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

Semantics:

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

$$
\frac{\exists \theta, \exists f(\overline{p}) \mathrel{\hat{=}} e.\ \overline{p}\theta = \overline{v}, \forall x \in \mathsf{vars}(\overline{p}).\ \theta(x) \in \Gamma_{\overline{p}}(x) \quad e\theta \Downarrow u}{f(\overline{v}) \Downarrow u}\textsc{Fun}
$$

Figure 1: The language describing forward transformations

Definition 6 (Strongly Regular Grammar)**.** A CFFG G is called *strongly regular* if the following condition holds. For each production rule $T \to f \in R$, if $f = \mathcal{C}[S]$ with S such that T and S are mutually defined, then S appears at the rightmost position of forest concatenation.

For example, $T \to \varepsilon, T \to$ `<a>`T`<b>` is not strongly regular because T does not occur at rightmost position of concatenations. On the other hand, $T \to$ `<a>`$, F \to TF$ is strongly regular because T does not generate F, i.e., F and T are not mutually defined.

Theorem 2 (Mohri and Nederhof (2001))**.** A strongly-regular CFFG $G = (\Sigma, N, R)$ can be converted to an RFG $G' = (\Sigma, N', R')$ such that for all $A \subseteq N$, there exists $A' \subseteq N'$ such that $[\![A]\!]_G = [\![A']\!]_{G'}$.

By this fact, we do not distinguish between a strongly-regular CFFG and the RFG corresponding to the CFFG in this paper.

We define the following relation for later discussions.

Definition 7 (Horizontal Overlap)**.** Let R and S be sets of forests. We define $R \parallel S$ if and only if there exist no forests f_1, f_2, f_3 such that $f_2 \neq \varepsilon$, $f_1 f_2 \in R \wedge f_3 \in S$ and $f_1 \in R \wedge f_2 f_3 \in S$.

We call "R and S are horizontally overlapped" if $R \nparallel S$. The notion $R \parallel S$ is called "unambiguously concatenable" in Bohannon et al. (2008) and "horizontally unambiguous" in Brabrand et al. (2007). The following is a well-known fact on forest grammars.

Fact. The following statements hold for forest grammars.

- For RFGs G_1, G_2, $[\![A]\!]_{G_1} \cap [\![B]\!]_{G_2} = \emptyset$ is decidable.
- For RFGs G_1, G_2, $[\![A]\!]_{G_1} \nparallel [\![B]\!]_{G_2}$ is decidable.

4.3 Syntax and Semantics

The syntax of the language is summarized in Figure 1. Formally, a program $\mathcal{P}$ is a pair (G, R): a strongly regular CFFG G defining types for look-ahead and a set of R defining rules of transformation functions.

A type declaration

$$\mathbf{data}\ T \mathrel{\hat{=}} t$$

describes a production rule $T \to t$ in the CFFG G, where we sometimes call T a *non-terminal* of G. For convenience, we sometimes use regular expressions when we show examples of programs written in the language. Since G must be strongly regular, every mutually defined non-terminal must occur at the rightmost position of concatenation.

A rule for transformations has the form of

$$f(p_1, \ldots, p_n) \hat{=} e$$

where each p_i is a pattern and e is a *treeless* expression (Wadler 1990). 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 a forest concatenation "$\centerdot$" as a freezed data constructor rather than a function. In addition, we require that variable occurrences should be *affine*: every variable must not be used more than once, i.e., there is no duplication.

Some example programs in this language have been given in Introduction and Section 2. In this paper, for simplicity, we assume that transformation programs are *deterministic* in the sense that pattern matching is unique. Formally, for any concatenation pattern $p_1 \centerdot p_2$, $[\![p_1]\!]_{\mathcal{P}} \parallel [\![p_2]\!]_{\mathcal{P}}$ holds, and for any two rules $f(p_1, \ldots, p_n) = e$ and $f(p'_1, \ldots, p'_n) \hat{=} e'$, $[\![p_i]\!]_{\mathcal{P}} \cap [\![p'_i]\!]_{\mathcal{P}} = \emptyset$ holds for some i. In addition, we require that every variable pattern $x :: T$ satisfies $|[\![T]\!]| > 1$, and assume that the length of the domain of every transformation function is more than one.

To describe the semantics, we use $\mathsf{vars}(t_1, \ldots, t_n)$ to return the set of all the variables occurring in a sequence of patterns/expressions $t_1, \ldots, t_n$, and $\overline{f}$ to denote a sequence $f_1, \ldots, f_n$, and $|\overline{f}|$ to denote the length of the sequence $\overline{f}$. A substitution $\theta : X \to \mathcal{T}_{\Sigma,X}{}^*$ is a function that maps a variable to a forest, of which $\{x \mid \theta(x) \neq x\}$ is finite. We denote by $t\theta$ a pattern/expression obtained from t by replacing each variable x in t with a forest $\theta(x)$. We write by $\Gamma_{\overline{p}}$ a type environment obtained by gathering variable pattern in patterns $\overline{p}$. For example, for $\overline{p} = (x :: T, \texttt{<t>}(y :: T'))$, we have $\Gamma_{\overline{p}} = \{x \mapsto T, y \mapsto T'\}$. The set of all the forests matching with a pattern p in a program $\mathcal{P}$, $[\![p]\!]_{\mathcal{P}}$, is defined by $[\![p]\!]_{\mathcal{P}} = \{f \mid \exists\theta.\ f = p\theta, \forall x \in \mathsf{vars}(p).\ \theta(x) \in \Gamma_p(x)\}$.

The semantics of the language is defined by the call-by-value semantics shown in Figure 1. We define the semantics of an expression e in a program $\mathcal{P}$ by $[\![e]\!]_{\mathcal{P}} = v$ if $e \Downarrow v$, $[\![e]\!]_{\mathcal{P}} = \perp$ otherwise. For simplicity, unless confusion arises, $[\![e]\!]_{\mathcal{P}}$ is sometimes written by $[\![e]\!]$ or even e. For a type T in a program $\mathcal{P} = (G, R)$, we sometimes write $[\![T]\!]_{\mathcal{P}}$ instead of $[\![T]\!]_G$. We may even use the simple notations like T and $[\![T]\!]$. The type of an expression e under a type environment Γ is defined by the set of all its possible evaluation results, i.e., the range of e under Γ, $\mathsf{ran}_\Gamma(e)$, which is defined by $\{[\![e\theta]\!] \mid \mathsf{dom}(\theta) = \mathsf{vars}(e), \theta(x) \in \Gamma(x)\}$.

For the convenience of analyses and transformations in later sections, we assume that every expression e has a unique ID throughout a program, which is denoted by $\#e$. We use $e_1 \centerdot_{(k)} e_2$ to denote a concatenation expression $e_1 \centerdot e_2$ with ID k,

5 Type Specialization and Range Inference

A type specialization is a program transformation that specializes a forward transformation according to the "look-ahead" type information to that for which exact range inference can be done. The idea of type specialization is not special; a similar approach is adopted in tree transducers (Maneth et al. 2007). We adopt it for our bidirectionalization.

5.1 Type Specialization

We start by considering the following program.

$$\begin{array}{l}
\textbf{data}\ A \hat{=} \texttt{<a>}^* \\
f(x :: A) \hat{=} g(x) \\
g(\varepsilon) \hat{=} \varepsilon;\ \ g(\texttt{<a>} \centerdot x) \hat{=} \texttt{<a>} \centerdot g(x);\ \ g(\texttt{<b>} \centerdot x) \hat{=} \texttt{<b>} \centerdot g(x)
\end{array}$$

Notice that the range of the expression $g(x)$ in the definition body of f differs from the range of the function g (i.e., $(\texttt{<a>} \mid \texttt{<b>})^*$), because of the specific type of x passed from f to g. We want to calculate the exact

$$\begin{array}{lcl}
\mathsf{ip}_{G/N,A}(\varepsilon) & \hat{=} & \{\emptyset\} \quad \text{if } A \in N \\
\mathsf{ip}_{G/N,A}(x :: (G,T)) & \hat{=} & \{x :: (G \times G/N, T \times A)\} \quad \text{if } [\![T \times A]\!]_{G \times G/N} \neq \emptyset \\
\mathsf{ip}_{G/N,A}(\sigma(p)) & \hat{=} & \{\Gamma \mid \Gamma \in \mathsf{ip}_{G,B}(p), A \to \sigma(B)C \in G, C \in N\} \\
\mathsf{ip}_{G/N,A}(p_1 \mathbin{.} p_2) & \hat{=} & \{\Gamma_1 \cup \Gamma_2 \mid \Gamma_1 \in \mathsf{ip}_{G/\{B\},A}(p_2), \Gamma_2 \in \mathsf{ip}_{G/N,B}(p_2), B \in G\} \\
\mathsf{ip}_{G/N,A}(\_) & \hat{=} & \emptyset
\end{array}$$

Figure 2: The definition of type inference procedure ip

range of $g(x)$ for later effective bidirectionalization and precise type checking. To do so, we make use the type information passed from f to generate a type-specialized version of g as follows.

$$f(x :: A) \mathrel{\hat{=}} g_A(x)$$
$$g_A(\varepsilon) \mathrel{\hat{=}} \varepsilon;\ \ g_A(\texttt{<a>} \mathbin{.} x) \mathrel{\hat{=}} \texttt{<a>} \mathbin{.} g_A(x)$$

Unlike g, g_A is an injective function, of which the inverse (a very effective backward transformation) can be automatically generated by our system, which will be addressed in Section 6.

A specialized transformation function may have more rules. For example, specializing the following program

$$\textbf{data}\ T \mathrel{\hat{=}} (\texttt{<a>} \mathbin{.} \texttt{<b>}) \mid (\texttt{<b>} \mathbin{.} \texttt{<a>}); \textbf{data}\ S \mathrel{\hat{=}} \texttt{<a>} \mid \texttt{<b>}$$
$$f(x :: T) \mathrel{\hat{=}} g(x)$$
$$g(x :: S \mathbin{.} y :: S) \mathrel{\hat{=}} x \mathbin{.} y$$

will yield the following g_T with two rules while the original g has only one rule.

$$f(x :: T) \mathrel{\hat{=}} g_T(x)$$
$$g_T(\texttt{<a>} \mathbin{.} \texttt{<b>}) \mathrel{\hat{=}} \texttt{<a>} \mathbin{.} \texttt{<b>};\ \ g_T(\texttt{<b>} \mathbin{.} \texttt{<a>}) \mathrel{\hat{=}} \texttt{<b>} \mathbin{.} \texttt{<a>}$$

We turn to formal definition of type specialization. We shall use $\Gamma(\overline{p})$ to represent the patterns obtained from the patterns $\overline{p}$ by replacing each $x :: T$ in $\overline{p}$ with $x :: \Gamma(x)$. To ease our explanation in this section, we represent the type A in $(x :: A)$ by (G, A) where G is a regular grammar used to define A.

Our type-specialization algorithm uses the type-inference procedure $\mathsf{ip}_{G,A}(p)$ to calculate a set of type environments mapping from variables (in the pattern p) to their types by specializing p under the type (G, A). For the above example, $\mathsf{ip}_{G,T}(x :: (G,S) \mathbin{.} y :: (G,S))$ returns $\{\{x \mapsto \texttt{<a>}, y \mapsto \texttt{<b>}\}, \{x \mapsto \texttt{<b>}, y \mapsto \texttt{<a>}\}\}$, which consists of two type environments corresponding to two rules for the specialized g_T. To define $\mathsf{ip}_{G,A}(p)$, we introduce the following notion.

Definition 8 (Chopped RFG)**.** A chopped regular forest grammar G/N is a pair of an RFG G and non-terminals N of which language $[\![A]\!]_{G/N}$ is defined by $\{t \mid A \xrightarrow{*} tB, B \in N\}$.

Note that every RFG G can be converted to a chopped RFG G/N where $N = \{A \mid A \to \varepsilon \in G\}$. It is not difficult to show that every chopped RFG G/N can be converted to a RFG G' that has the same languages. Now the definition of $\mathsf{ip}_{G/N,A}(p)$ is given in Figure 2.

Theorem 3. The function ip exactly infers the types of variables in pattern p that appears the context specified by (G, A), i.e.,

$$\mathsf{ip}_{G/N,A}(p) = \mathcal{G} \Rightarrow [\![A]\!]_{G/N} \cap [\![p]\!] = \bigcup\{[\![\Gamma(p)]\!] \mid \Gamma \in \mathcal{G}\}.$$

Now our type specialization algorithm is defined as follows.

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

$$\frac{}{\Gamma, \varepsilon \overset{g}{\rightsquigarrow} \varepsilon}\textsc{Eps} \qquad \frac{}{\Gamma, x \overset{g}{\rightsquigarrow} \Gamma(x)}\textsc{Var} \qquad \frac{\Gamma, e \overset{g}{\rightsquigarrow} t}{\Gamma, \sigma(e) \overset{g}{\rightsquigarrow} \sigma(t)}\textsc{Con}$$

$$\frac{\Gamma, e_1 \overset{g}{\rightsquigarrow} t_1 \quad \Gamma, e_2 \overset{g}{\rightsquigarrow} t_2}{\Gamma, e_1 \centerdot e_2 \overset{g}{\rightsquigarrow} t_1 \centerdot t_2}\textsc{Cat} \qquad \frac{}{\Gamma, f(\overline{x}) \overset{g}{\rightsquigarrow} T_f}\textsc{Fun}$$

$$\frac{\Gamma_{\overline{p}}, e \overset{g}{\rightsquigarrow} t}{f(\overline{p}) \mathrel{\hat{=}} e \overset{g}{\dashrightarrow} T_f \to t}\textsc{R}$$

Figure 3: The derivation rules for type inference of functions where $r \overset{g}{\dashrightarrow} r'$ reads a production rule r' is defined from r

1. For each function call $f(\overline{x})$ in $h(\overline{q}) \mathrel{\hat{=}} \mathcal{C}[f(\overline{x})]$ with $\Gamma_{\overline{q}}(x_i) = (G_i, A_i)$ for $i \in \{1, \ldots, |\overline{x}|\}$, repeat the steps 2–5.
2. For each rule $f(\overline{p}) \mathrel{\hat{=}} e$, repeat the steps 3–5.
3. $\mathcal{G} := \{\Gamma_1 \cup \cdots \cup \Gamma_{|\overline{x}|} \mid \Gamma_i \in \mathcal{G}_i\}$ where $\mathcal{G}_i = \mathsf{ip}_{G_i,A_i}(p_i)$ for each $i \in \{1, \ldots, |\overline{x}|\}$.
4. For each $\Gamma \in \mathcal{G}$, generate rules $f_{\overline{(G,N)}}(\Gamma(\overline{p})) \mathrel{\hat{=}} e'$ where e' is obtained by replacing function calls $g(\overline{y})$ by $g_{\overline{\Gamma(y)}}(\overline{y})$.
5. Recursively apply this algorithm to all the function calls occurring in the newly-produced rules until no new rules are generated in the step 4. □

Theorem 4. Type specialization terminates.

Proof Sketch. The number of every regular forest *language* that appears in the execution of type specialization is finite because production of a finite set of automata becomes closed after finite steps. Concretely, for a program $\mathcal{P} = (G, R)$, the regular forest language is an element of a set of regular forest languages $\{[\![B_1 \times \cdots \times B_m]\!]_{G/A_1 \times \cdots G/A_l \times G/N \times \cdots \times G/N} \mid A_i, B_j \in G, l \leq m\}$ where $m = n(n+1)$, n is the number of non-terminals in G, and N is the set of non-terminals defined by $N = \{A \mid A \to \varepsilon \in G\}$. Note that the equivalence of the two regular forest language is known to be decidable (Comon et al. 1997). □

To guarantee that an obtained program after type specialization is deterministic, we must be sure that G is unambiguous in the sense that there are no A, B such that $[\![A]\!]_G \cap [\![B]\!]_G \neq \emptyset$. Converting a program to that satisfying this condition is not difficult, by means of standard conversion from NFTA to DFTA (Comon et al. 1997). Note that this conversion may increase the program size exponentially in the worst case, although this rarely happens in practice. The above proof also implies the upper bound of space complexity is $\mathrm{O}(2^{n^2})$, where n is the number of non-terminals in G of a program $\mathcal{P} = (G, R)$. The conversion from NFTA to DFTA implies $n = 2^m$ in the worst case where m is the number of non-terminals of the original G of a program $\mathcal{P} = (G, R)$.

Theorem 5 (Type Specialized Program)**.** After type specialization, for every $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[g(\overline{x})]$, if $g(\overline{t})\downarrow$, then $t_i \in \Gamma_{\overline{p}}(x_i)$ $(i \in \{1, \ldots, |\overline{x}|\})$ holds.

Theorem 5 says that, after type specialization, the range of a function call expression is never shrunk by the context where the expression occurs. In other words, the later range inference can ignore the contexts where the functions are called.

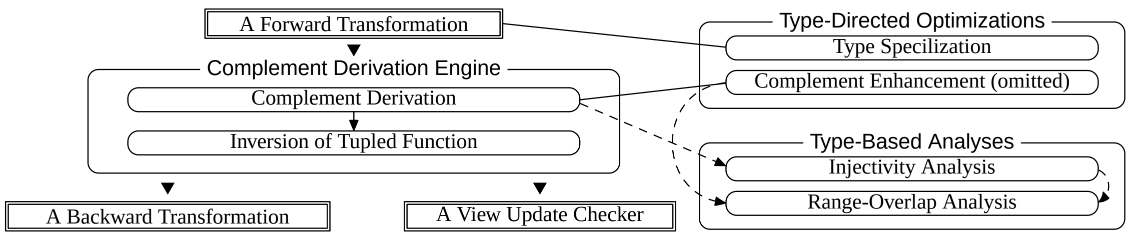

Figure 4: An overview of bidirectionalization system

5.2 Range Inference

Range inference is to infer ranges of functions, i.e., the types of function call expressions. It plays an important role in effective bidirectionalization (Section 6) and type checking. The basic idea for inferring ranges of functions is to forget arguments of functions. For example, for a type-specialized program

$$\begin{array}{l} \textbf{data}\ C \mathrel{\hat{=}} \texttt{<c>}^* \\ f(x :: \texttt{<a>}^*, y :: C) \mathrel{\hat{=}} y \centerdot g(x) \\ g(\varepsilon) \mathrel{\hat{=}} \varepsilon;\ g(\texttt{<a>} \centerdot x :: \texttt{<a>}^*) \mathrel{\hat{=}} \texttt{<b>} \centerdot g(x) \end{array}$$

forgetting arguments of functions yields the following CFFG.

$$T_f \to C T_g \quad T_g \to \varepsilon \quad T_g \to \texttt{<b>} T_g \quad C \to \varepsilon \quad C \to \texttt{<c>} C$$

A formal range inference algorithm based on this idea is defined by the derivation rules in Figure 3. We will denote the obtained CFFG for a program $\mathcal{P}$ by $G_{\mathcal{P}}$.

Theorem 6 (Soundness)**.** For each expression $g(\overline{x})$ in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[g(\overline{x})]$, $\mathsf{ran}_{\Gamma_{\overline{p}}}(g(\overline{x})) \subseteq [\![T_g]\!]$ holds.

Theorem 7 (Exactness)**.** After type specialization, for each expression $g(\overline{x})$ in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[g(\overline{x})]$, $\mathsf{ran}_{\Gamma_{\overline{p}}}(g(\overline{x})) = [\![T_g]\!]$ holds.

5.3 Type Checking

As seen before, the range of a function in our language is captured by a language of a CFFG if its inputs are specified by regular forests languages. Consider to check whether or not the type of a forward transformation h subsumes a type $\overline{A} \to B$. For a function h and its input types $\overline{A}$, we can derive a type-specialized version $h_{\overline{A}}$ of which range $\mathsf{ran}(h_{\overline{A}})$ is described by a CFFG. Then, it is sufficient and necessary to check whether $\mathsf{ran}(h_{\overline{A}}) \cap B^{\mathrm{c}}$ is empty. If the set is empty then there exist forests $\overline{f} \in [\![\overline{A}]\!]$ such that $h(\overline{f}) \notin [\![B]\!]$, otherwise there are no such forests. It is known that the emptiness of intersection of a language of a CFFG and a language of an RFG is decidable (Guessarian 1983), and the complement of a regular forest language is a regular forest language (Comon et al. 1997).

Moreover, we can easily guarantee that the result of $h_{\overline{A}\mathrm{B}}$, the derived backward transformation of $h_{\overline{A}}$ by the bidirectionalization discussed later, always falls in $[\![\overline{A}]\!]$.

6 Bidirectionalization

An overview of the present bidirectionalization is shown is Figure 4. The core part of the bidirectionalization is the same as that in (Matsuda et al. 2007), which consists of three major steps: (1) Derivation of a complement function f^{c} from a given forward transformation f (Section 6.3), (2) Calculation of the definition of $\langle f, f^{\mathrm{c}}\rangle^{-1}$ (Section 6.4), (3) Generation of the backward transformation corresponding to the derived complement function by $f_{\mathrm{B}}(s, v) = \langle f, f^{\mathrm{c}}\rangle^{-1}(v, f^{\mathrm{c}}(s))$. Besides, our new bidirectionalization system uses type-directed optimizations to derive much smaller complement functions than before. We adopt two optimizations here: one is the type specialization in Section 5, and the other is the complement enhancement technique discussed in the previous work (Matsuda et al. 2007). In addition, by the injectivity checking based on range inference of expressions, for a detected injective function, our bidirectionalization system returns the inverse, which is the "best" backward transformation, as a backward transformation of the function. Together with a backward transformation, our bidirectionalization system provides a view update checker (Section 6.6) that tells users what kinds of update on views can be reflected to the source.

Types play important roles in our bidirectionalization. All the optimizations for effective backward transformations are based on inference of ranges of expressions or types of expressions. The importance of types has been seen in type specialization discussed in Section 5, which enables us to infer more precise types of expressions. As will be seen later, thanks to the type specialization, our bidirectionalization system can derive *exact* view update checkers.

In this section, we focus on the new parts of our bidirectionalization compared with the previous work (Matsuda et al. 2007). Since the present language contains forest concatenation and look-ahead, the present bidirectionalization must treat the forest concatenation and the look-ahead well.

6.1 Type Approximation

Consider the following two functions:

$$\begin{array}{l}\mathbf{data}\ A \mathrel{\hat{=}} \texttt{<a>}^*; \mathbf{data}\ B \mathrel{\hat{=}} \texttt{<b>}^* \\ f(x :: A, y :: B) \mathrel{\hat{=}} x \centerdot y;\ \ g(x :: A, y :: A) \mathrel{\hat{=}} x \centerdot y\end{array}$$

where f is injective whereas g is not; the non-injectivity of g comes from non-injectivity of forest concatenation. The injectivity of forest concatenation depends on the types of two operands; in the above example, we have $A \not\parallel A$ but $A \parallel B$.

Although types of expressions are easily calculated by an extension of the range inference, types expressed by CFFGs are not suitable for injectivity checking because, in the injectivity checking, the following must be computable for types A and B.

- $A \cap B = \emptyset$?
- $A \not\parallel B$?

It is known that the first is undecidable for types described by general CFFGs, while both are decidable for types described by RFGs (Brabrand et al. 2007).

Our idea is to approximate types described by CFFGs by types described by RFGs using the regular approximation of context-free grammars by Mohri and Nederhof (2001)[5]. For instance, for the following context-free word grammar that cannot be converted to a regular word grammar because the grammar is not strongly regular,

$$A \to \mathtt{a}A\mathtt{b} \quad A \to \varepsilon$$

the algorithm splits the problematic non-terminal A to two non-terminals as follows: A to produce the "left" part of the original A and A' to produce the "right" part of the original A, to obtain the following approximated regular word grammar.

$$A \to \mathtt{a}A \quad A \to A' \quad A' \to \varepsilon \quad A' \to \mathtt{b}A'$$

As another example, for the following CFFG

$$A \to \texttt{<a>}(\texttt{<b>}A\texttt{<c>}) \centerdot \texttt{<d>} \quad A \to \varepsilon$$

[5]A forest-version of their algorithm is shown in Appendix B.

$$\frac{}{\Gamma, \varepsilon \overset{\mathrm{ga}}{\leadsto} \{T_{\#\varepsilon} \to \varepsilon\}}\,\textsc{Eps} \qquad \frac{}{\Gamma, x \overset{\mathrm{ga}}{\leadsto} \{T_{\#x} \to \Gamma(x)\}}\,\textsc{Var}$$

$$\frac{\Gamma, e \overset{\mathrm{ga}}{\leadsto} \Delta}{\Gamma, \sigma(e) \overset{\mathrm{ga}}{\leadsto} \{T_{\#\sigma(e)} \to \sigma(T_{\#e})\} \cup \Delta}\,\textsc{Con}$$

$$\frac{\Gamma, e_1 \overset{\mathrm{ga}}{\leadsto} \Delta \quad \Gamma, e_2 \overset{\mathrm{ga}}{\leadsto} \Delta'}{\Gamma, e_1 \mathbin{\centerdot} e_2 \overset{\mathrm{ga}}{\leadsto} \{T_{\#e_1 \centerdot e_2} \to T_{\#e_1} \mathbin{\centerdot} T_{\#e_2}\} \cup \Delta \cup \Delta'}\,\textsc{Cat}$$

$$\frac{}{\Gamma, f(\overline{x}) \overset{\mathrm{ga}}{\leadsto} \{T_{\#f(\overline{x})} \to T_f\}}\,\textsc{Fun}$$

Figure 5: The derivation rules for a CFFG for type inference where $e \overset{\mathrm{ga}}{\leadsto} \Delta$ reads production rules Δ are obtained from e

our forest version of the approximation algorithm splits the problematic non-terminal A to two non-terminals as follows: A for the "top-left" part of the original A and A' for the "right" part of the original A, and produces the following RFG.

$$A \to \texttt{<a>}(\texttt{<b>}A) \mathbin{\centerdot} \texttt{<d>} \mathbin{\centerdot} A' \quad A \to A' \quad A' \to \varepsilon \quad A' \to \texttt{<c>}$$

It should be noted that naive application of the approximation algorithm in Mohri and Nederhof (2001) would reduce the precision of the type inference. For example, for a CFFG,

$$A \to BA \quad A \to \varepsilon \quad B \to \texttt{<a>}(A)$$

the approximation returns the following CFFG.

$$A \to B \;\; A \to A' \;\; A' \to \varepsilon \;\; B \to \texttt{<a>}(A)B' \;\; B' \to A \;\; B' \to \varepsilon$$

The language of A in the obtained grammar is wider than A while the language of the original A is regular. To solve this problem, we notice that such general context-free grammars only come from recursion structures of functions in their range inference, therefore we approximate the grammar for types of functions rather than arbitrary expressions.

Figure 5 summarizes our derivation rules for calculating an approximated type $T_{\#e}$ for an expression e. Gathering all the derived production rules with the regular approximated version of $G_{\mathcal{P}}$, we get the grammar $\widetilde{G_{\mathcal{P}}}$ that describes approximated types of expressions. The obtained grammar $\widetilde{G_{\mathcal{P}}}$ may not be strongly regular, but all the languages of non-terminals are regular.

Definition 9. For a expression e in $f(\overline{p}) \doteq \mathcal{C}[e]$, we define approximated type $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e)$ of e by

$$\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) = [\![T_{\#e}]\!]_{\widetilde{G_{\mathcal{P}}}}.$$

The following condition guarantees that $\mathsf{ran}_{\Gamma_{\overline{p}}}(e) = \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e)$ for any e in $f(\overline{p}) \doteq \mathcal{C}[e]$.

Condition 1. For any function call of $g(\overline{x})$ in a rule $f(\overline{p}) = \mathcal{C}[g(\overline{x})]$ where f and g are mutually defined, g occurs at the rightmost positions.

By "f and g are mutually defined", we mean that g is called in the evaluation of f and f is called in the evaluation of g.

6.2 Injectivity Checking

It is important to check whether a transformation function is injective, because an injective function will lead to the "best" backward transformation denoted by its inverse so that any view modifications can be

$$\frac{\exists f(\overline{p}) \hat{=} e \in R.\ (\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e) \neq \emptyset}{f \in \mathit{NINJ}}\text{R1}$$

$$\frac{\exists f(\overline{p}) \hat{=} e, f(\overline{p}') \hat{=} e' \in R.\ \overline{p} \neq \overline{p}', \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) \cap \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}'}}(e') \neq \emptyset}{f \in \mathit{NINJ}}\text{R2}$$

$$\frac{\exists f(\overline{p}) \hat{=} \mathcal{C}[g(\overline{x})] \in R.\ g \in \mathit{NINJ}}{f \in \mathit{NINJ}}\text{R3}$$

$$\frac{\exists f(\overline{p}) \hat{=} \mathcal{C}[e_1 \centerdot e_2] \in R.\ \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) \nparallel \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e)}{f \in \mathit{NINJ}}\text{R4}$$

Figure 6: Logical implication rules to calculate *NINJ*

reflected to the source. Notice that there are only four places where the injectivity of a function is lost in our language (unused variables, right-hand-side-overlapped rules, calls of non-injective functions, and horizontally-overlapped concatenation), so the injectivity checking can be done by checking the existence of the four places. A set of injective functions *INJ* is defined by the complement of a set of possibly-non-injective functions *NINJ* defined by the least fixed point of the logical implication rules shown in Figure 6.

Theorem 8 (Soundness)**.** Any function f in *INJ* is injective.

Theorem 9 (Completeness)**.** After type specialization, if Condition 1 holds, *INJ* contains all the injective functions in a program.

6.3 Derivation of Complement Functions

We have seen some examples of complement functions in Section 3. As another example related to the forest concatenation, recall the following program.

$$\begin{array}{l}\textbf{data}\ A \hat{=} \texttt{<a>}^*; \textbf{data}\ B \hat{=} \texttt{<b>}^* \\ f(x :: A, y :: B) \hat{=} x \centerdot y;\ g(x :: A, y :: A) \hat{=} x \centerdot_{(1)} y\end{array}$$

By the injective checking discussed before, our bidirectionalization detects the injectivity of f and the non-injectivity of g caused by the non-injective forest concatenation. In the case, our system generates the following complement function to split the result of the non-injective forest concatenation of g.

$$g^{\mathrm{c}}(x :: A, y :: A) \hat{=} \textsc{len}_1(x)$$

The function $\textsc{len}_k$, which calculates the length of a forest, is introduced to split the result of a non-injective forest concatenation expression with ID k. This ID information is used in later inverse calculation process.

Formally, our complement derivation is defined by the complement derivation rules in Figure 7. A complement derivation relation $r \overset{\mathrm{c}}{\dashrightarrow} r^{\mathrm{c}}$ reads that a rule of complement function r^{c} is derived from r. Gathering all r^{c} for all rules r for all functions, we get a program for complement functions.

Theorem 10 (Correctness of Complement Derivation)**.** Let $\mathcal{P} = (G, R)$ and $\mathcal{P}^{\mathrm{c}} = (G, \{r^{\mathrm{c}} \mid r \in R, r \overset{\mathrm{c}}{\dashrightarrow} r^{\mathrm{c}}\})$ programs. For any function f in $\mathcal{P}$, f^{c} in $\mathcal{P}^{\mathrm{c}}$ is a complement function of f.

Note that this complement derivation can return different results for $(e_1 \centerdot e_2) \centerdot e_3$ and $e_1 \centerdot (e_2 \centerdot e_3)$. For example, for the functions f and g defined by

$$\begin{array}{l}\textbf{data}\ A \hat{=} \texttt{<a>}^*; \textbf{data}\ B \hat{=} \texttt{<b>}^* \\ f(x :: A, y :: B, z :: B) \hat{=} (x \centerdot y) \centerdot_{(1)} z \\ g(x :: A, y :: B, z :: B) \hat{=} x \centerdot (y \centerdot_{(2)} z)\end{array}$$

$$\frac{}{\Gamma, \varepsilon \overset{c}{\rightsquigarrow} \epsilon}\textsc{Eps} \quad \frac{}{\Gamma, x \overset{c}{\rightsquigarrow} \epsilon}\textsc{Var} \quad \frac{\Gamma, e \overset{c}{\rightsquigarrow} \overline{e^c}}{\Gamma, \sigma(e) \overset{c}{\rightsquigarrow} \overline{e^c}}\textsc{Con}$$

$$\frac{f \in \mathit{INJ}}{\Gamma, f(\overline{x}) \overset{c}{\rightsquigarrow} \epsilon}\textsc{IFun} \quad \frac{f \notin \mathit{INJ}}{\Gamma, f(\overline{x}) \overset{c}{\rightsquigarrow} f^c(\overline{x})}\textsc{Fun}$$

$$\frac{\widetilde{\mathsf{ran}}_\Gamma(e_1) \nparallel \widetilde{\mathsf{ran}}_\Gamma(e_2) \quad \Gamma, e_1 \overset{c}{\rightsquigarrow} \overline{e_1{}^c} \quad \Gamma, e_2 \overset{c}{\rightsquigarrow} \overline{e_2{}^c}}{\Gamma, e_1 \centerdot_{(i)} e_2 \overset{c}{\rightsquigarrow} \textsc{len}_i(e), \overline{e_1{}^c}, \overline{e_2{}^c}}\textsc{ICat}$$

$$\frac{\widetilde{\mathsf{ran}}_\Gamma(e_1) \parallel \widetilde{\mathsf{ran}}_\Gamma(e_2) \quad \Gamma, e_1 \overset{c}{\rightsquigarrow} \overline{e_1{}^c} \quad \Gamma, e_2 \overset{c}{\rightsquigarrow} \overline{e_2{}^c}}{\Gamma, e_1 \centerdot e_2 \overset{c}{\rightsquigarrow} \overline{e_1{}^c}, \overline{e_2{}^c}}\textsc{Cat}$$

$$\frac{\Gamma_{\overline{p}}, e \overset{c}{\rightsquigarrow} \overline{e^c} \quad V = \mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e)}{f(\overline{p}) \hat{=} e \overset{c}{\dashrightarrow} f^c(\overline{p}) \hat{=} \mathsf{R}_{f(\overline{p}) \hat{=} e}\langle \overline{e^c}, V \rangle}\textsc{R}$$

Figure 7: The complement derivation rules where $r \overset{c}{\dashrightarrow} r^c$ reads that a rule r^c of complement function is derived from a rule r, and ϵ represents the empty sequence

the complement derivation returns the following functions.

$$f^c(x :: A, y :: B, z :: B) \hat{=} \mathsf{R}_1\langle \textsc{len}_1(x \centerdot y)\rangle$$
$$g^c(x :: A, y :: B, z :: B) \hat{=} \mathsf{R}_2\langle \textsc{len}_2(x)\rangle$$

In this paper, we treat the forest concatenation as a right-associative operator, i.e., $e_1 \centerdot e_2 \centerdot e_3 = e_1 \centerdot (e_2 \centerdot e_3)$ but $e_1 \centerdot e_2 \centerdot e_3 \neq (e_1 \centerdot e_2) \centerdot e_3$.

6.4 Inverse Calculation

After a complement function f^c of an original forward transformation f is derived, our bidirectionalization calculates the inverse of the tupled function of f and f^c, i.e., $\langle f, f^c\rangle^{-1}$. In contrast to our previous work (Matsuda et al. 2007), the inverse calculation by simply swapping left-hand sides with right-hand sides does not work for the present target language because of forest concatenations. There are two reasons. First, this simply swapping would produce invalid patterns because the horizontally-overlapped patterns are prohibited in our target language. Second, since the complement derivation algorithm introduces the LEN functions, the complement function could include nested functions calls.

To solve this problem, we introduce the function SPL to cancel the effect of LEN in inversion calculation. So for the following transformation program

$$\begin{array}{l} \textbf{data}\ A \hat{=} \texttt{<a>}^* \\ f(x :: A, y :: A, z :: A) \hat{=} g(x) \centerdot_{(1)} g(y) \centerdot_{(2)} z \\ g(x :: A) \hat{=} x \end{array}$$

we first drive its complement

$$f^c(x, y, z) \hat{=} \mathsf{R}_1\langle \textsc{len}_1(g(x)), \textsc{len}_2(g(y))\rangle$$

and then obtain the following program for $\langle f, f^c\rangle^{-1}$.

$$\begin{array}{lll} \multicolumn{3}{l}{\langle f, f^c\rangle^{-1}(v_1, \mathsf{R}_1\langle l_1, l_2\rangle) \hat{=} (x, y, z)} \\ \quad\textbf{where} & (w_{\#g(x)}, v_2) & \hat{=} \textsc{spl}(l_1, v_1) \\ & (w_{\#g(y)}, z :: A) & \hat{=} \textsc{spl}(l_2, v_2) \\ & x :: A & \hat{=} g^{-1}(w_{\#g(x)}) \\ & y :: A & \hat{=} g^{-1}(w_{\#g(y)}) \end{array}$$

Here, a variable v_k is for a forest concatenation expression with ID k, a variable l_k is for the length of the left operand of a forest concatenation expression with ID k, and a variable w_i is for a function call expression

$$\frac{}{\Gamma, \varepsilon, L \overset{\mathrm{w}}{\rightsquigarrow} \emptyset}\textsc{Eps} \qquad \frac{}{\Gamma, x, L \overset{\mathrm{w}}{\rightsquigarrow} \emptyset}\textsc{Var} \qquad \frac{\Gamma, e, L \overset{\mathrm{w}}{\rightsquigarrow} W}{\Gamma, \sigma(e), L \overset{\mathrm{w}}{\rightsquigarrow} W}\textsc{Con}$$

$$\frac{f \in \mathit{INJ}}{\Gamma, f(\overline{x}), L \overset{\mathrm{w}}{\rightsquigarrow} (\overline{x :: \Gamma(x)}) \mathrel{\hat{=}} {f_i}^{-1}(w_{f_i})}\textsc{IFun}$$

$$\frac{f \notin \mathit{INJ}}{\Gamma, f(\overline{x}), L \overset{\mathrm{w}}{\rightsquigarrow} (\overline{x :: \Gamma(x)}) \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(w_{\#f(\overline{x})}, w_{\#f^{\mathrm{c}}(\overline{x})})}\textsc{Fun}$$

$$\frac{\begin{array}{c}\widetilde{\mathsf{ran}}_\Gamma(e_1) \nparallel \widetilde{\mathsf{ran}}_\Gamma(e_2) \quad l_k \in L \\ \Gamma, e_1, L \overset{\mathrm{w}}{\rightsquigarrow} W \quad \Gamma, e_2, L \overset{\mathrm{w}}{\rightsquigarrow} W' \quad \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \quad \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2\end{array}}{\Gamma, e_1 \bullet_{(k)} e_2, L \overset{\mathrm{w}}{\rightsquigarrow} W \cup W' \cup (p_1, p_2) \mathrel{\hat{=}} \textsc{spl}(l_k, v_k)}\textsc{ICat}$$

$$\frac{\widetilde{\mathsf{ran}}_\Gamma(e_1) \parallel \widetilde{\mathsf{ran}}_\Gamma(e_2) \quad \Gamma, e_1, L \overset{\mathrm{w}}{\rightsquigarrow} W_1 \quad \Gamma, e_2, L \overset{\mathrm{w}}{\rightsquigarrow} W_2}{\Gamma, e_1 \bullet e_2, L \overset{\mathrm{w}}{\rightsquigarrow} W_1 \cup W_2}\textsc{Cat}$$

$$\frac{}{\Gamma, \varepsilon \overset{\mathrm{p}}{\rightsquigarrow} \varepsilon}\textsc{Eps} \qquad \frac{}{\Gamma, x \overset{\mathrm{p}}{\rightsquigarrow} x :: \Gamma(x)}\textsc{Var}$$

$$\frac{\Gamma, e_i \overset{\mathrm{p}}{\rightsquigarrow} p_i \ (i \in \{1, \dots, |\overline{e}|\})}{\Gamma, \mathsf{R}_k\langle\overline{e}\rangle \overset{\mathrm{p}}{\rightsquigarrow} \mathsf{R}_k\langle\overline{p}\rangle}\textsc{FCon} \qquad \frac{\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p}{\Gamma, \sigma(e) \overset{\mathrm{p}}{\rightsquigarrow} \sigma(p)}\textsc{Con}$$

$$\frac{}{\Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_{\#f(\overline{x})} :: T_{\#f(\overline{x})}}\textsc{Fun} \qquad \frac{}{\Gamma, \textsc{len}_k(x) \overset{\mathrm{p}}{\rightsquigarrow} l_k}\textsc{Len}$$

$$\frac{\widetilde{\mathsf{ran}}_\Gamma(e_1) \parallel \widetilde{\mathsf{ran}}_\Gamma(e_2) \quad \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \quad \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2}{\Gamma, e_1 \bullet e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \bullet p_2}\textsc{ICat}$$

$$\frac{\widetilde{\mathsf{ran}}_\Gamma(e_1) \nparallel \widetilde{\mathsf{ran}}_\Gamma(e_2)}{\Gamma, e_1 \bullet_{(k)} e_2 \overset{\mathrm{p}}{\rightsquigarrow} v_k :: T_k}\textsc{Cat}$$

Figure 8: Pattern- and **where**-clause-generation rules

with ID i. The function $\textsc{spl}(l, v)$ is a function that has similar behavior to the `splitAt` function in Haskell except that $\textsc{spl}(l, v)$ is undefined if the length of v is smaller than l. Note that $\textsc{spl}$ satisfies the law

$$\textsc{spl}(\textsc{len}(f_1), f_1 \bullet f_2) = (f_1, f_2)$$

for any forests f_1 and f_2.

In the following, we will show how to construct an algorithm to derive the above $\langle f, f^{\mathrm{c}}\rangle^{-1}$ for f. As shown above, **where**-clauses are introduced to represent the inverse of the tupled functions. Note that patterns are allowed in left-hand sides of rules in **where**.

6.4.1 Generating where

As seen above, inverses that we want to calculate contain **where**-clauses and patterns in the left-hand sides of **where**. We describe an algorithm in Figure 8 for generating **where**-clauses and patterns by (1) the **where**-clause-generation relation $\Gamma, e, L \overset{\mathrm{w}}{\rightsquigarrow} W$ reading that under a type environment Γ **where**-clauses W for the inverse function is derived from an expression e and a set of L for the results of $\textsc{len}$, and (2) the pattern-generation relation $\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p$ reading that a pattern p is derived from e to decompose its result.

6.4.2 Inverse Calculation Algorithm

We derive a function definition rule r' for the inverse of the tupled function by the following derivation rule, where $r \overset{\mathrm{i}}{\dashrightarrow} r'$ reads that r' for the inverse of tupled function is derived from a rule r of the original forward transformation.

$$\frac{f \in \mathit{INJ} \quad \Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q \quad \Gamma_{\overline{p}}, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W}{f(\overline{p}) \mathrel{\hat{=}} e \overset{\mathrm{i}}{\dashrightarrow} f^{-1}(q) \mathrel{\hat{=}} \overline{p} \;\textbf{where}\; W}\,\mathrm{IR}$$

$$\frac{\begin{array}{c} f \notin \mathit{INJ} \quad f(\overline{p}) \mathrel{\hat{=}} e \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{p}) \mathrel{\hat{=}} e^{\mathrm{c}} \\ \Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q \quad \Gamma_{\overline{p}}, e^{\mathrm{c}} \overset{\mathrm{p}}{\rightsquigarrow} q' \quad \Gamma_{\overline{p}}, e, \mathsf{vars}(q') \overset{\mathrm{w}}{\rightsquigarrow} W \end{array}}{f(\overline{p}) \mathrel{\hat{=}} e \overset{\mathrm{i}}{\dashrightarrow} \langle f, f^{\mathrm{c}}\rangle^{-1}(q, q') \mathrel{\hat{=}} \overline{p} \;\textbf{where}\; W}\,\mathrm{R}$$

Let $\mathcal{P} = (G, R)$ be a program, and $\mathcal{P}' = (\widetilde{G_{\mathcal{P}}}, \{r' \mid r \overset{\mathrm{i}}{\dashrightarrow} r', r \in R\})$ be a program for the inverses of the tupled functions. We have the following theorems.

Theorem 11 (Deterministic Inverses)**.** $\mathcal{P}'$ is deterministic.

Theorem 12 (Correctness of Inverse Calculation)**.** For any $f \in \mathit{INJ}$ in $\mathcal{P}$, f^{-1} in $\mathcal{P}'$ is the inverse of f, and for any $f \notin \mathit{INJ}$ in $\mathcal{P}$, $\langle f, f^{\mathrm{c}}\rangle^{-1}$ in $\mathcal{P}'$ is the inverse of $\langle f, f^{\mathrm{c}}\rangle$.

Note that even after complement enhancement (removing tags, unifying tags) in Matsuda et al. (2007), functions in $\mathcal{P}'$ are deterministic because of look-ahead.

6.5 Generation of Backward Transformation

Backward transformations are obtained by simple functional composition. Concretely, our system derives a backward transformation f_{B} of f by $f_{\mathrm{B}}(\_, v) \mathrel{\hat{=}} f^{-1}(v)$ for $f \in \mathit{INJ}$, and $f_{\mathrm{B}}(s, v) \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(v, f^{\mathrm{c}}(s))$ for $f \notin \mathit{INJ}$. All the derived backward transformations satisfy the bidirectional properties shown in Section 3.

6.6 View Update Checker

Since a backward transformation derived from a non-injective forward transformation usually disallows some updates on views, it is convenient for users to know whether a view update is reflectable before the reflection of the view update is performed; it would be irritating if a view updated is rejected after long-time execution of a backward transformation. We shall provide a *exact* view update checker thanks to the type specialization.

For a function f in *INJ*, the range inference is enough because the derived backward transformation can reflect all the views in the range of f. However, for a function not in *INJ*, we need more discussion. Since the backward transformations are obtained under constant complement bidirectionalization, we have

$$\{v \mid f_{\mathrm{B}}(s, v)\downarrow\} = \{v' \mid f_{\mathrm{B}}(f_{\mathrm{B}}(s, v), v'), f_{\mathrm{B}}(s, v)\downarrow\}.$$

This means that reflectable view updates are constant once an initial source is given. Since we have $f_{\mathrm{B}}(s, v) = \langle f, f^{\mathrm{c}}\rangle^{-1}(v, f^{\mathrm{c}}(s))$ for $f \notin \mathit{INJ}$, we realize a view update checker by inference of the domain of $\langle f, f^{\mathrm{c}}\rangle^{-1}$ when the second argument, the return value of the complement function, is fixed. We use alternating CFFGs for view update checkers, in which $\wedge$ is allowed in the right-hand sides.

For example, for a function f

$$\begin{array}{l} \textbf{data}\; A \mathrel{\hat{=}} \texttt{<a>}^{*} \\ f(x :: A, y :: A, z :: A) \mathrel{\hat{=}} g(x) \,\centerdot_{(1)}\, g(y) \,\centerdot_{(2)}\, z \\ g(x :: A) \mathrel{\hat{=}} x \end{array}$$

and $\langle f, f^{\mathrm{c}}\rangle^{-1}$ in Section 6.4 defined by

$$\begin{array}{llll} \multicolumn{4}{l}{\langle f, f^{\mathrm{c}}\rangle^{-1}(v_1, \mathsf{R}_1\langle l_1, l_2\rangle) \mathrel{\hat{=}} (x, y, z)} \\ & \textbf{where} & (w_{\#g(x)}, v_2) & \mathrel{\hat{=}} \textsc{spl}(l_1, v_1) \\ & & (w_{\#g(y)}, z :: A) & \mathrel{\hat{=}} \textsc{spl}(l_2, v_2) \\ & & x :: A & \mathrel{\hat{=}} g^{-1}(w_{\#g(x)}) \\ & & y :: A & \mathrel{\hat{=}} g^{-1}(w_{\#g(y)}) \end{array}$$

$$\frac{f \in \mathit{INJ}}{\theta, (\overline{x :: T}) \mathrel{\hat{=}} f^{-1}(w) \overset{\text{uc}}{\rightsquigarrow}_{r,t} (X^{w}_{r,t} \to S^{\varepsilon}_{f})}\,\text{IFW}$$

$$\frac{f \notin \mathit{INJ} \quad t' = w'\theta}{\theta, (\overline{x :: T}) \mathrel{\hat{=}} \langle f, f^{\text{c}}\rangle^{-1}(w, w') \overset{\text{uc}}{\rightsquigarrow}_{r,t} (X^{w}_{r,t} \to S^{t'}_{f})}\,\text{FW}$$

$$\frac{}{\theta, (p_1, p_2) \mathrel{\hat{=}} \textsc{spl}(l, w) \overset{\text{uc}}{\rightsquigarrow}_{r,t} (X^{w}_{r,t} \to (T_{r,t}[\![p_1]\!] \wedge L_l) \mathbin{\centerdot} T_{r,t}[\![p_2]\!])}\,\text{SW}$$

$$\frac{\exists\theta.\ t = p'\theta \ \ G = \{\delta \mid \theta, w \overset{\text{uc}}{\rightsquigarrow}_{r,t} \delta, w \in W\} \cup G^{p,e,W}_{r,t}}{r = \langle f, f^{\text{c}}\rangle^{-1}(p, p') \mathrel{\hat{=}} (e, t) \textbf{ where } W \overset{\text{uc}}{\dashrightarrow}_{t} G \cup \{S^{t}_{f} \to T_{r,t}[\![p]\!]\}}\,\text{R}$$

$$\frac{G = \{\delta \mid \theta, w \overset{\text{uc}}{\rightsquigarrow}_{r,\varepsilon} \delta, w \in W\} \cup G^{p,e,W}_{r,t}}{r = f^{-1}(p) \mathrel{\hat{=}} e \textbf{ where } W \overset{\text{uc}}{\dashrightarrow}_{t} G \cup \{S^{\varepsilon}_{f} \to T_{r,\varepsilon}[\![p]\!]\}}\,\text{IR}$$

$G^{p,e,W}_{r,t} = \{X^{v}_{r,t} \to \Gamma_{p,W}(v), v \in \mathsf{vars}(e) \cap \mathsf{vars}(p)\}$
where $\Gamma_{p,W}$: the type environment obtained from p and LHSs of W

$$\begin{aligned}
T_{r,t}[\![\sigma(q)]\!] &= \sigma(T_{r,t}[\![q]\!]) \\
T_{r,t}[\![\varepsilon]\!] &= \varepsilon \\
T_{r,t}[\![w]\!] &= X^{w}_{r,t} \\
T_{r,t}[\![p_1 \mathbin{\centerdot} p_2]\!] &= T_{r,t}[\![p_1]\!] \mathbin{\centerdot} T_{r,t}[\![p_2]\!]
\end{aligned}$$

$$L_0 \to \varepsilon \qquad L_k \to \sigma(\_)L_{k_1} \quad \text{(for any label } \sigma)$$

Figure 9: The derivation rules for a view update checker

and an initial source $s = (\texttt{<a>}, \texttt{<a><a>}, \texttt{<a>})$ with the return value of the derived complement function $t = \mathsf{R}_1\langle 1, 2\rangle = f^{\text{c}}(s)$, the following view update checker is derived using partial evaluation of the second argument.

$$\begin{array}{lll}
S^{t}_{f} \to X^{v_1}_{1,t} & X^{w\#g(x)}_{1,t} \to S^{\varepsilon}_{g} & A \to \varepsilon \\
X^{v_1}_{1,t} \to (X^{w\#g(x)}_{1,t} \wedge L_1)X^{v_2}_{1,t} & X^{w\#g(y)}_{1,t} \to S^{\varepsilon}_{g} & A \to \texttt{<a>}A \\
X^{v_2}_{1,t} \to (X^{w\#g(y)}_{1,t} \wedge L_2)X^{z}_{1,t} & S^{\varepsilon}_{g} \to X^{x}_{1,\varepsilon} & \\
X^{z}_{1,t} \to A & X^{x}_{1,\varepsilon} \to A &
\end{array}$$

Here, S^{t}_{f} generates all the forests s such that $\langle f, f^{\text{c}}\rangle^{-1}(s, t)\downarrow$, $X^{w}_{r,t}$ generates all the possible forests u that are bound to variable w in the rule r of $\langle f, f^{\text{c}}\rangle^{-1}$ invoked with $\langle f, f^{\text{c}}\rangle^{-1}(s, t)$ for some input $s = \mathcal{C}[u]$ and $\langle f, f^{\text{c}}\rangle^{-1}(s, t)\downarrow$, and L_k generates all the forests of the length k. Hence, the inferred set of reflectable views in this case is $\{\texttt{<a>}^{n} \mid n \geq 3\}$.

Formally, for a program of inverses of tupled functions $\mathcal{P} = (G, R)$ and a return value of a complement function t, a view update checker $G^{t}_{\text{U}} = \bigcup\{G \mid r, t \overset{\text{uc}}{\dashrightarrow}_{t} G, r \in R\}$ is derived using the derivation relation $\overset{\text{uc}}{\dashrightarrow}_{t}$ defined as Figure 9. Note that this view update checker generation also ignores type information of some variables in left-hand sides of **where**, which correspond to the types for look-ahead in the corresponding forward transformations. The type specialization guarantees that the type information is safely ignored.

An obtained grammar can be converted to an RFG if Condition 1 holds, otherwise it can be converted to a CFFG because, for each $\wedge$, one side of $\wedge$ is always L_l for some l. The concrete description for conversion algorithm is shown in Appendix C.

Theorem 13 (Completeness)**.** For a view update checker G^{t}_{U}, $\langle f, f^{\text{c}}\rangle^{-1}(s, t)\downarrow$ for $f \notin \mathit{INJ}$ implies $S^{t}_{f} \xrightarrow{*} s$, and $f^{-1}(s, t)\downarrow$ fro $f \in \mathit{INJ}$ implies $S^{\varepsilon}_{f} \xrightarrow{*} s$.

Theorem 14 (Soundness)**.** Let $\mathcal{P}$ be a program for the inverse of tupled functions derived from a type-specialized program. For the view update checker G^{t}_{U} obtained from $\mathcal{P}$, $S^{t}_{f} \xrightarrow{*} s$ implies $\langle f, f^{\text{c}}\rangle^{-1}(s, t)\downarrow$ for $f \notin \mathit{INJ}$, and $S^{\varepsilon}_{f} \xrightarrow{*} s$ implies G^{t}_{U} for $f \in \mathit{INJ}$.

7 Related Work

We discuss some other related work, in addition to what has been discussed in Introduction.

The idea of type specialization is not new. The type specialization and type checking/inference in this paper is similar to Maneth et al. (2007), in which type specialization is considered to avoid treating input types. They provide the exact type check in polynomial time for transformation written by linear stay macro tree transducers using context-free tree grammar on binary encoding of forests. The context-free tree grammars on binary encoding of forests are more expressive than CFFGs. The main difference between our type inference and their type inference is that we allow regular look-ahead and non-tail variables in patterns while they do not. XDuce (Hosoya and Pierce 2003) provides the type inference on the language with non-tail variables with look-ahead in patterns. One main difference is that, in XDuce, patterns instead of regular type expressions are used as look-ahead. Another big difference is the range of ip: a *set* of type environment in this paper but one type environment in their algorithm. The other difference is that we do not assume that types of functions are given beforehand because we want to estimate as precise types of expressions as possible and the restriction of our language simplifies the problem.

The injectivity check in this paper follows Brabrand et al. (2008). The difference is that we treat forest-to-forest transformations while they treat tree-to-string transformations. They adopt a more precise approximation method because the ranges of tree-to-string transformations usually fall in context-free word languages.

8 Conclusion

In this paper, we have shown how to relax the treeless restriction of the language in the previous work (Matsuda et al. 2007) by introducing forest concatenation and look-ahead specified by regular expression types, and proposed a new bidirectionalization system, with which a wide class of practical forward transformations can be automatically and effectively bidirectionalized. The key factor of our new system is the novel use of the type specialization in bidirectionalization: the type specialization enables us to infer ranges precisely, derive effective backward transformations, and generate exact view update checkers.

Although the treeless restriction in the previous work has been relaxed in this paper, the affine restriction still remains as an interesting future work. The relaxation of the affine restriction is also important in practice; for example with non-affine transformations, users can obtain two different views of the same data at the same time by pairing two different forward transformations. It would be possible to derive a more effective backward transformation if we could take into account the fact that the information of both transformations is obtained simultaneously.

Acknowledgements

We are gratefull to Keisuke Nakano, who introduced us the related work by Maneth et al. (2007). We also thank Kiminori Matsuzaki for his comments on an earlier version of this paper.

References

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4): 557–575, 1981.

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.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 407–419, 2008.

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*, July 2007.

C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Info. Syst.*, 33(4), June 2008.

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.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.

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, 2002.

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

I. Guessarian. Pushdown tree automata. *Math. Syst. Theor.*, 16(4):237–263, 1983.

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

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

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 178–189, 2004.

S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 201–214, 2006.

R. Lämmel. Coupled software transformations (extended abstract). In *First International Workshop on Software Evolution Transformations*, pages 31–35, 2004.

J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, 2003.

S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *Database Theory — ICDT 2007*, volume 4353 of *LNCS*, pages 254–268, 2007.

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, 2007.

L. Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, 1998.

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

H. Ohsaki, H. Seki, and T. Takai. Recognizing boolean closed A-tree languages with membership conditional rewriting mechanism. In *Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 483–498, 2003.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

A Proofs

A.1 Proof of Theorem 3

We prove by induction on the definition of ip.

For the epsilon case and the variable case, the statement trivially holds.

Inductive Step: Constructor

$(\Rightarrow)$ Let t be a tree such that $t \in [\![\sigma(p)]\!] \cap [\![A]\!]_{G/N}$. It is easy to conclude that $t = \sigma(s)$ for some s and G/N has a production rule $A \to \sigma(B)C$ where $s \in [\![p]\!] \cap [\![B]\!]_G$ and $C \in N$. From the induction hypothesis, there is some $\Gamma \in \mathsf{ip}_{G,B}(p)$ such that $s \in [\![\Gamma(p)]\!]$. By the definition of ip, we have $\Gamma \in \mathsf{ip}_{G/N,B}(p)$ and $t \in [\![\Gamma(\sigma(p))]\!]$.

$(\Leftarrow)$ Let t be a tree such that $t \in [\![\Gamma(\sigma(p))]\!]$ for some $\Gamma \in \mathsf{ip}_{G,B}(\sigma(p))$, which implies $t = \sigma(s)$ for some s. By the definition of ip, we have $\Gamma \in \mathsf{ip}_{G,B}(p)$ and $\sigma(B)C \to A$ with $C \in N$. From the induction hypothesis, we have $s \in [\![p]\!] \cap [\![B]\!]_G$, which implies $t \in [\![\sigma(p)]\!] \cap [\![A]\!]_{G/N}$.

Inductive Step: Concatenation

$(\Rightarrow)$ Let f be a forest such that $f \in [\![p_1 \centerdot p_2]\!] \cap [\![A]\!]_{G/N}$. By the definition of patterns, we have the unique forests f_1, f_2 such that $f_1 \centerdot f_2 = f$, $f_1 \in [\![p_1]\!]$ and $f_2 \in [\![p_2]\!]$, and a non-terminal B such that $A \xrightarrow{*} f_1 B$ and $B \xrightarrow{*} f_2 C$ where $C \in N$, which implies $f_1 \in [\![p_1]\!] \cap [\![A]\!]_{G/\{B\}}$ and $f_2 \in [\![p_2]\!] \cap [\![B]\!]_{G/N}$. From the induction hypothesis, there exist $\Gamma_1 \in \mathsf{ip}_{G/[\![B]\!],A}(p_1)$ and $\Gamma_2 \in \mathsf{ip}_{G/N,B}(p_2)$ such that $f_1 \in [\![\Gamma_1(p_1)]\!]$ and $f_2 \in [\![\Gamma_2(p_2)]\!]$. By the definition of ip, since p_1 and p_2 share no variable, we have some $\Gamma \in \mathsf{ip}_{G/N,A}(p_1 \centerdot p_2)$ such that $\Gamma = \Gamma_1 \cup \Gamma_2$ and $f_1 \centerdot f_2 \in [\![\Gamma(p_1 \centerdot p_2)]\!]$.

$(\Leftarrow)$ Let f be a forest such that $f \in [\![\Gamma(p_1 \centerdot p_2)]\!]$ for some $\Gamma \in \mathsf{ip}_{G/N}(p_1 \centerdot p_2)$, which implies there exists $f = f_1 \centerdot f_2$, $f_1 \in [\![\Gamma_1(p_1)]\!], \in f_2[\![\Gamma_2(p_2)]\!]$ for $\Gamma_1 \in \mathsf{ip}_{G/[\![B]\!]}(p_1)$, $\Gamma_2 \in \mathsf{ip}_{G/N}(p_2)$ and B. By the induction hypothesis, we have $f_1 \in [\![p_1]\!] \cap [\![A]\!]_{G/\{B\}}$ and $f_2 \in [\![p_2]\!] \cap [\![B]\!]_{G/N}$, which implies $f_1 \centerdot f_2 \in [\![p_1 \centerdot p_2]\!] \cap [\![A]\!]_{G/N}$.

A.2 Proof of Theorem 4

Let $\mathcal{P} = (G, R)$ be a program. We prove the statement by showing that the number of all the regular forest languages occurring in the type-specialization algorithm is finite.

First, we discuss the property of ip, the type-inference sub-procedure for patterns. Let G', B be a pair of an RFG and its non-terminal, an input of ip. Then, ip only does the following things.

- Change of the chopping states (N of G'/N) to
  - M such that $M = \{A \mid A \to \varepsilon \in G'\}$
  - $\{A\}$ with $A \in G'$
- Change of the start non-terminal to $B' \in G'$
- Calculation of a product with (G, A) with $A \in G$ at last

The above implies the following property.

Property 1. After $(m-1)$-times of application of ip, the possible pairs of RFG and start non-terminal in the type-specialization algorithm is an element of the set

$$\{[\![B_1 \times \cdots \times B_m]\!]_{G/A_1 \times \cdots G/A_l \times G/N \times \cdots \times G/N} \mid A_i, B_j \in G, l \leq m\}$$

where $N = \{A \mid A \to \varepsilon \in G\}$.

Proof. Straightforward from the above. Note that $G/A \times G/B = (G \times G)/(A \times B)$. $\square$

Let n be the number of the non-terminals in G. After $(n(n+1)-1)$-times of application of ip, the set of the possible pairs of RFG and start non-terminal in the type-specialization algorithm becomes

$$\{[\![B_1 \times \cdots \times B_m]\!]_{G/A_1 \times \cdots G/A_l \times G/N \times \cdots \times G/N} \mid A_i, B_j \in G, l \leq m\}$$

where $m = n(n+1)$. In the set, any combination of A_i and B_j already appears. In other words, we have

$$\begin{aligned}\{[\![B_1 \times \cdots \times B_m]\!]_{G/A_1 \times \cdots G/A_l \times G/N \times \cdots \times G/N} \mid A_i, B_j \in G, l \leq m\} \\ = \{[\![B_1 \times \cdots \times B_{m+1}]\!]_{G/A_1 \times \cdots G/A_l \times G/N \times \cdots \times G/N} \mid A_i, B_j \in G, l \leq m+1\}.\end{aligned}$$

From the above discussion, we conclude that the number of regular forest languages occurring in the execution of the type specialization algorithm is finite, which implies that the type-specialization algorithm terminates. Note that checking whether two regular forest languages are equivalent is decidable (Comon et al. 1997).

A.3 Proof of Theorem 5

Since any call $g(\overline{x})$ in $f(\overline{p}) \hat{=} \mathcal{C}[g(\overline{x})]$ is replaced by $g_{\overline{\Gamma_{\overline{p}}(x)}}(\overline{x})$ in type specialization, by the definition of ip, any rule $g_{\overline{\Gamma_{\overline{p}}(x)}}(\overline{p}) \hat{=} e$ of $g_{\overline{\Gamma_{\overline{p}}(x)}}$ satisfies $[\![\overline{p}]\!] \subseteq [\![\overline{\Gamma_{\overline{p}}(x)}]\!]$, which implies Theorem 5.

From now on, we use the following property of a type-specialized program.

Property 2. For every $f(\overline{p}) \hat{=} \mathcal{C}[g(\overline{x})]$, if $g(\overline{t})\downarrow$, then $t_i \in \Gamma_{\overline{p}}(x_i)$ $(i \in \{1, \ldots, |\overline{x}|\})$ holds.

A.4 Proof of Theorem 6

We prove the following statement by induction on the evaluation of $g(\overline{x})$ in $f(\overline{p}) \hat{=} \mathcal{C}[g(\overline{x})]$.

> For each expression e, if $e\theta \Downarrow v$ for some $\theta : \mathsf{vars}(e) \to \mathcal{T}_\Sigma{}^*$ such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$, then $t \xrightarrow{*} v$ where $\Gamma, e \overset{\mathrm{g}}{\rightsquigarrow} t$.

Clearly the statement implies Theorem 6.

Base Case: Empty Forest

Since we have $\Gamma, \varepsilon \overset{\mathrm{g}}{\rightsquigarrow} \varepsilon$ and $\varepsilon\theta \Downarrow \varepsilon$ for any θ, the statement trivially holds.

Base Case: Variable

Consider the case when $x\theta \Downarrow v$ with $\theta(x) \in \Gamma(x)$. Since $\theta(x)$ does not contain variable, we have $v = \theta(x)$. Since we have $\Gamma, x \overset{\mathrm{g}}{\rightsquigarrow} \Gamma(x)$ and $\theta(x) \in \Gamma(x)$, we also have $\Gamma(x) \xrightarrow{*} v$, which implies the statement.

Inductive Step: Constructor

Consider the case when $\sigma(e\theta) \Downarrow \sigma(v)$ with $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$. We have $\Gamma, \sigma(e) \overset{\mathrm{ga}}{\rightsquigarrow} \sigma(t)$ where $\Gamma, e \overset{\mathrm{ga}}{\rightsquigarrow} t$. From the induction hypothesis, we have $t \xrightarrow{*} v$. Hence, we have $\sigma(t) \xrightarrow{*} \sigma(v)$, which implies the statement.

Inductive Step: Concatenation

Consider the case when $(e_1 \centerdot e_2)\theta \Downarrow v$ with $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1, e_2)$. We have $\Gamma, (e_1 \centerdot e_2) \overset{\mathrm{ga}}{\rightsquigarrow} t_1 \centerdot t_2$ where $\Gamma, e_1 \overset{\mathrm{ga}}{\rightsquigarrow} t_1$ and $\Gamma, e_2 \overset{\mathrm{ga}}{\rightsquigarrow} t_2$. Let v_1, v_2 be forests such that $v = v_1 \centerdot v_2$ and $e_1\theta \Downarrow v_1$ and $e_2\theta \Downarrow v_2$. From the induction hypothesis, we have $t_1 \xrightarrow{*} v_1$ and $t_2 \xrightarrow{*} v_2$. Then, we have $t_1 \centerdot t_2 \xrightarrow{*} v_1 \centerdot v_2$, which implies the statement.

Inductive Step: Functional Call

Consider the case when $f(\overline{x}\theta) \Downarrow v$ with $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(\overline{x})$. By the definition of $\Downarrow$, we have $f(\overline{p}) \mathrel{\hat{=}} e$ such that $\overline{p}\eta = \overline{x}\theta$ for some η with $\eta(x) \in \Gamma_{\overline{p}}(x)$ for any x, and $e\eta \Downarrow v$. From the induction hypothesis, we have $t \xrightarrow{*} v$ where $\Gamma_{\overline{p}}, e \overset{\text{ga}}{\rightsquigarrow} t$. Since we have a production rule $T_f \to t$ by the definition of $\overset{\text{g}}{\dashrightarrow}$ and $\Gamma, f(\overline{x}) \overset{\text{ga}}{\rightsquigarrow} T_f$, we have $T_f \xrightarrow{*} v$, which implies the statement.

A.5 Proof of Theorem 7

We prove the following statement by induction on the production $t \xrightarrow{*} v$.

> For each expression e in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[e]$, if $t \xrightarrow{*} v$ with t such that $\Gamma, e \overset{\text{ga}}{\rightsquigarrow} t$, then there exists θ such that $e\theta \Downarrow v$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$.

Base Case: Empty Forest

In this case, we have $\Gamma, \varepsilon \overset{\text{ga}}{\rightsquigarrow} \varepsilon$ and $\varepsilon \xrightarrow{*} \varepsilon$. Then, taking $\theta = \{\}$, we have $\varepsilon\theta \Downarrow \varepsilon$, which implies the statement.

Base Case: Variable

In this case, we have $\Gamma, x \overset{\text{ga}}{\rightsquigarrow} \Gamma(x)$, and $\Gamma(x) \xrightarrow{*} v$ for a forest v, i.e., $v \in \Gamma(x)$. Then, taking $\theta = \{x \mapsto v\}$, we have $x\theta \Downarrow v$ and $x\theta \in \Gamma(x)$, which implies the statement.

Inductive Step: Constructor

In this case, we have $\Gamma, \sigma(e) \overset{\text{ga}}{\rightsquigarrow} \sigma(t)$ where $\Gamma, e \overset{\text{ga}}{\rightsquigarrow} t$, and have $\sigma(t) \xrightarrow{*} \sigma(v)$ for a forest v. From the induction hypothesis, there exists θ such that $e\theta \Downarrow v$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$. Hence, we have $\sigma(e\theta) \Downarrow \sigma(v)$, which implies the statement.

Inductive Step: Concatenation

In this case, we have $\Gamma, e_1 \centerdot e_2 \overset{\text{ga}}{\rightsquigarrow} t_1 \centerdot t_2$ where $\Gamma, e_1 \overset{\text{ga}}{\rightsquigarrow} t_1$ and $\Gamma, e_2 \overset{\text{ga}}{\rightsquigarrow} t_2$, and have $t_1 \centerdot t_2 \xrightarrow{*} v_1 \centerdot v_2$ for forests v_1 and v_2 such that $t_1 \xrightarrow{*} v_1$ and $t_2 \xrightarrow{*} v_2$. From the induction hypothesis, there exist θ_1 and θ_2 such that $e_1\theta_1 \Downarrow v_1$ and $\theta_1(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1)$, and $e_2\theta_2 \Downarrow v_2$ and $\theta_2(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_2)$. Then, taking $\theta = \theta_1 \cup \theta_2$, we have $(e_1 \centerdot e_2)\theta \Downarrow v_1 \centerdot v_2$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1 \centerdot e_2)$, which implies the statement.

Inductive Step: Function Call

In this case, we have $\Gamma, f(\overline{x}) \overset{\text{ga}}{\rightsquigarrow} T_f$ and $T_f \xrightarrow{*} v$, which implies $t \xrightarrow{*} v$ where $\Gamma_{\overline{p}}, e \overset{\text{ga}}{\rightsquigarrow} t$ for some rule $f(\overline{p}) \mathrel{\hat{=}} e$. From the induction hypothesis, there exists a substitution η such that $e\eta \Downarrow v$ and $\eta(x) \in \Gamma_{\overline{p}}$ for any $x \in \mathsf{vars}(e)$. Then, taking θ by $\overline{x}\theta = \overline{p}\eta$, we have $f(\overline{x})\theta \Downarrow v$ and have $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(\overline{x})$ by the Property 2, which implies the statement.

A.6 Proof of Theorem 8

We prove the following statement, which implies the contraposition of Theorem 8.

> For any function f, if there exist different sequences of forests $\overline{v}$ and $\overline{u}$ such that $[\![f(\overline{u})]\!] = [\![f(\overline{v})]\!]$, then f is in *NINJ*.

There are two cases.

- $\exists f(\overline{p}) \mathrel{\hat{=}} e.\ \overline{u} = \overline{p}\theta \land \overline{v} = \overline{p}\theta'$.

- Otherwise.

The second case is trivial because in the case we have two rules of which the right-hand expressions have a range-overlap. To prove the statement in the first case, we prove the following statement by induction on the evaluation.

> For any expression e in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[e]$, if $[\![e\theta]\!] = [\![e\theta']\!]$ holds for two different substitutions $\theta : \mathsf{vars}(e) \to \mathcal{T}_\Sigma{}^*$ and $\theta' : \mathsf{vars}(e) \to \mathcal{T}_\Sigma{}^*$ such that $\theta(x) \in \Gamma_{[\![p]\!]}(x), \theta'(x) \in \Gamma_{[\![p]\!]}(x)$ for all $x \in \mathsf{vars}(\overline{p})$, then f is in *NINJ*.

Base Case: Empty Forest

Since there are no two different substitutions θ and θ' such that $\varepsilon\theta = \varepsilon\theta'$, the statement trivially holds.

Base Case: Variable

Since there are no two different substitutions θ and θ' such that $x\theta = x\theta'$, the statement trivially holds.

Inductive Step: Constructor

Clear from the induction hypothesis.

Inductive Step: Concatenation

There are three cases for $e_1 \centerdot e_2$.

- $e_1\theta = e_1\theta', e_2\theta \neq e_2\theta$.
- $e_1\theta \neq e_2\theta', e_2\theta = e_2\theta'$.
- Otherwise.

The third case implies there exists a horizontal overlap of the ranges of e_1 and e_2, then algorithm reports $f \in \mathit{NINJ}$. For the first case, it is enough to apply the induction hypothesis for e_1. For the second case, it is enough to apply the induction hypothesis for e_2.

Inductive Step: Function Call

For a function call $g(\overline{x})$, There are two cases.

- $\exists g(\overline{q}) \mathrel{\hat{=}} e.\ \overline{x}\theta = \overline{q}\eta, \overline{x}\theta' = \overline{q}\eta'$.
- Otherwise.

We define ζ and ζ' by $\zeta = \{x \mapsto \eta(x) \mid x \in \mathsf{vars}(e)\}$ and $\zeta' = \{x \mapsto \eta' \mid x \in \mathsf{vars}(e)\}$ respectively. Note that, generally $\eta = \zeta \wedge \eta' = \zeta'$ does not hold because there exists some unused variable in the rule $g(\overline{q}) \mathrel{\hat{=}} e$. Consider the first case. If there exists a variable x such that $\zeta(x) \neq \zeta'(x)$, from the induction hypothesis, we have $g \in \mathit{INJ}$, which implies $f \in \mathit{INJ}$ because f calls g. If there exists no variable x such that $\zeta(x) \neq \zeta'(x)$, i.e., $\zeta = \zeta'$, since there exists a variable x in $\eta(x) \neq \eta'(x)$ because θ and θ' differs, g has unused variables, which implies $g \in \mathit{INJ}$ and $f \in \mathit{INJ}$.

The second case implies the there exists the range overlap of two rules of g, which implies $g \in \mathit{NINJ}, f \in \mathit{NINJ}$.

A.7 Proof of Theorem 9

We prove the following statement by the definition of *NINJ*, which implies the contraposition of Theorem 9.

> If $f \in \mathit{NINJ}$, there exist different $\overline{v}$ and $\overline{u}$ such that $[\![f(\overline{u})]\!] = [\![f(\overline{v})]\!]$.

Note that in the proof we use the following assumption on our target language.

- For any variable pattern $x :: T$, $|[\![T]\!]| > 1$.
- For any function f, $|\mathsf{dom}(f)| > 1$.

Base Case: R1

In this case, there exists a rule $f(\overline{p}) \mathrel{\hat{=}} e$ such that $\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e)$. Let $\overline{u}$ be a sequence of forests such that $\overline{p}\theta = \overline{u}$, $\theta(x) \in \Gamma_{\overline{p}}(x)$ for any x in $\mathsf{vars}(\overline{p})$, and $f(\overline{u})\!\downarrow$. Let z be a variable in $(\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e))$. Since $\Gamma_{\overline{p}}(z)$ contains more than one element by the assumption on our target language, there exists a forest t such that $t \neq \theta(z)$ and $t \in \Gamma_{\overline{p}}(z)$. Taking θ' as

$$\theta'(x) = \begin{cases} \theta(x) & x \neq z \\ t & x = z, \end{cases}$$

we have $[\![f(\overline{u})]\!] = [\![f(\overline{p}\theta')]\!]$ while $\overline{u} \neq \overline{p}\theta'$.

Base Case: R2

In this case, we have two different rules

$$f(\overline{p}) \mathrel{\hat{=}} e \qquad f(\overline{p}') \mathrel{\hat{=}} e'$$

satisfying $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) \cap \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}'}}(e') \neq \emptyset$. Condition 1 and Property 2 say that range approximation is exact, i.e, $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) = \mathsf{ran}_{\Gamma_{\overline{p}}}(e)$ and $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}'}}(e') = \mathsf{ran}_{\Gamma_{\overline{p}'}}(e')$. This implies that there exist two substitutions θ and θ' such that $[\![f(\overline{p}\theta)]\!] = [\![f(\overline{p}'\theta')]\!]$, $\theta(x) \in \Gamma_{\overline{p}}(x)$ for any x in $\mathsf{vars}(\overline{p})$, and $\theta'(x) \in \Gamma_{\overline{p'}}(x)$ for any x in $\mathsf{vars}(\overline{p'})$.

Inductive Step: R3

Consider a rule $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[g(\overline{x})]$ with $g \in \mathit{NINJ}$. In this case, since $g \in \mathit{NINJ}$, from the induction hypothesis, there are two different sequences of forests $\overline{u}$ and $\overline{v}$ such that $[\![g(\overline{u})]\!] = [\![g(\overline{v})]\!]$. Let θ and θ' be substitutions such that $\overline{x}\theta = \overline{u}$, $\overline{x}\theta' = \overline{v}$ and $f(\overline{p}\theta)\!\downarrow$ and $f(\overline{p}\theta')\!\downarrow$. Note that such θ and θ' always exist because of the assumption on our target language and because the language is affine. Property 2 guarantees that $\theta(x) \in \Gamma_{\overline{p}}(x)$ and $\theta'(x) \in \Gamma_{\overline{p}}(x)$ for any x in $\mathsf{vars}(\overline{P})$.

Base Case: R4

Consider a rule $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[e_1 \centerdot e_2]$ satisfying $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) \not\parallel \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}'}}(e')$. Condition 1 and Property 2 say that range approximation is exact, i.e, $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(e) = \mathsf{ran}_{\Gamma_{\overline{p}}}(e)$ and $\widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}'}}(e') = \mathsf{ran}_{\Gamma_{\overline{p}'}}(e')$. This implies that there exist two different substitutions η and η' such that $[\![(e_1 \centerdot e_2)\eta]\!] = [\![(e_2 \centerdot e_2)\eta']\!]$. Since our target language is affine, by the assumption on our target language, there exist two substitutions θ and θ' such that $f(\overline{p}\theta)\!\downarrow$ and $f(\overline{p}\theta')\!\downarrow$ and $\theta(x) = \eta(x) \wedge \theta'(x) = \eta'(x)$ for any x in $\mathsf{vars}(e_1 \centerdot e_2)$. Note that $\overline{p}\theta \neq \overline{p}\theta'$ because η and η' differs.

A.8 Proof of Theorem 10

We prove the following statement.

$$[\![f(\overline{u})]\!] = [\![f(\overline{v})]\!] \wedge [\![f^{\mathrm{c}}(\overline{u})]\!] = [\![f^{\mathrm{c}}(\overline{v})]\!] \Rightarrow \overline{u} = \overline{v}$$

To prove the above, we prove the following lemma.

lemma 1. Let e be an expressions and $\overline{e^{\mathrm{c}}}$ a sequence of expressions such that $\Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}}$. Then

$$\begin{aligned}&\forall\theta,\theta'.\\ &\quad\forall x\in\mathsf{vars}(e).\ \theta(x)\in\Gamma(x)\wedge\theta'(x)\in\Gamma(x)\wedge(\llbracket e\theta\rrbracket=\llbracket e\theta'\rrbracket)\wedge(\llbracket\overline{e^{\mathrm{c}}}\theta\rrbracket=\llbracket\overline{e^{\mathrm{c}}}\theta'\rrbracket)\\ &\qquad\Rightarrow\forall x\in\mathsf{vars}(e).\ \theta(x)=\theta'(x)\end{aligned}$$

holds.

Proof. We prove the lemma by induction on the evaluation of $e\theta$ and $e\theta'$ and the structure of e.

Base Case: Empty Forest

Since there no variable in an expression ε, the statement trivially holds.

Base Case: Variable

Since there is only one variable in an expression x, the statement trivially holds.

Inductive Step: Constructor

In this case, we have

$$\Gamma,\sigma(e)\overset{\mathrm{c}}{\rightsquigarrow}\overline{e^{\mathrm{c}}}\quad\text{where}\quad\Gamma,e\overset{\mathrm{c}}{\rightsquigarrow}\overline{e^{\mathrm{c}}}$$

and

$$\llbracket\sigma(e)\theta\rrbracket=\llbracket\sigma(e)\theta'\rrbracket\qquad\llbracket\overline{e^{\mathrm{c}}}\theta\rrbracket=\llbracket\overline{e^{\mathrm{c}}}\theta'\rrbracket.$$

Since we have $e\theta = e\theta'$, from the induction hypothesis, we have

$$\forall x\in\mathsf{vars}(e).\ \theta(x)=\theta'(x),$$

which implies the statement.

Inductive Step: Non-Horizontally-Overlapped Concatenation

In this case, we have

$$\Gamma,e_1\centerdot e_2\overset{\mathrm{c}}{\rightsquigarrow}\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}}\quad\text{where}\quad\Gamma,e_1\overset{\mathrm{c}}{\rightsquigarrow}\overline{e_1{}^{\mathrm{c}}}\quad\Gamma,e_2\overset{\mathrm{c}}{\rightsquigarrow}\overline{e_2{}^{\mathrm{c}}}.$$

and

$$\llbracket(e_1\centerdot e_2)\theta\rrbracket=\llbracket(e_1\centerdot e_2)\theta'\rrbracket\qquad\llbracket(\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta\rrbracket=\llbracket(\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta'\rrbracket.$$

Since we have $\llbracket e_1\theta\rrbracket=\llbracket e_1\theta\rrbracket\wedge\llbracket\overline{e_1{}^{\mathrm{c}}}\theta\rrbracket=\llbracket\overline{e_1{}^{\mathrm{c}}}\theta'\rrbracket$, from the induction hypothesis, we obtain

$$\forall x\in\mathsf{vars}(e_1).\ \theta(x)=\theta'(x).$$

Similarly, since we have $\llbracket e_2\theta\rrbracket=\llbracket e_2\theta\rrbracket\wedge\llbracket\overline{e_2{}^{\mathrm{c}}}\theta\rrbracket=\llbracket\overline{e_2{}^{\mathrm{c}}}\theta'\rrbracket$, from the induction hypothesis, we obtain

$$\forall x\in\mathsf{vars}(e_2).\ \theta(x)=\theta'(x).$$

Hence, we have $\theta(x)=\theta'(x)$ for any x in $\mathsf{vars}(e_1\centerdot e_2)$.

Inductive Step: Horizontally-Overlapped Concatenation

In this case, we have

$$\Gamma,e_1\centerdot e_2\overset{\mathrm{c}}{\rightsquigarrow}\textsc{len}(e_1),\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}}\quad\text{where}\quad\Gamma,e_1\overset{\mathrm{c}}{\rightsquigarrow}\overline{e_1{}^{\mathrm{c}}}\quad\Gamma,e_2\overset{\mathrm{c}}{\rightsquigarrow}\overline{e_2{}^{\mathrm{c}}}.$$

and

$$\llbracket(e_1\centerdot e_2)\theta\rrbracket=\llbracket(e_1\centerdot e_2)\theta'\rrbracket\qquad\llbracket(\textsc{len}(e_1),\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta\rrbracket=\llbracket(\textsc{len}(e_1),\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta'\rrbracket.$$

Unlike the above case, we may have the case when $e_1\theta\neq e_1\theta'$ and $e_1\theta'\neq e_2\theta'$ but $(e_1\centerdot e_2)\theta=(e_1\centerdot e_2)\theta'$. However, such a case never occurs because in such a case $\textsc{len}(e_1\theta)\neq\textsc{len}(e_1\theta')$ holds, which implies $\llbracket(\textsc{len}(e_1),\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta\rrbracket\neq\llbracket(\textsc{len}(e_1),\overline{e_1{}^{\mathrm{c}}},\overline{e_2{}^{\mathrm{c}}})\theta'\rrbracket$. The rest of the proof for this case is the same as the above case.

Inductive Step: Injective Function Call

Clear from the Theorem 8. Note that all the functions in *INJ* are injective.

Inductive Step: Function Call

In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\leadsto} f^{\mathrm{c}}(\overline{x})$$

and

$$[\![f(\overline{x}\theta)]\!] = [\![f(\overline{x}\theta')]\!] \wedge [\![f^{\mathrm{c}}(\overline{x}\theta)]\!] = [\![f^{\mathrm{c}}(\overline{x}\theta')]\!].$$

Here, we safely assume that there exists only one rule of the form $f(\overline{p}) \mathrel{\hat{=}} e$ such that $\overline{p}\eta = \overline{x}\theta$, $\overline{p}\eta' = \overline{x}\theta'$, $\eta(x), \eta'(x) \in \Gamma_{\overline{p}}$ for any x in $\mathsf{vars}(\overline{p})$ because the complement derivation rules introduce different tags to different tags; if there are two rules $f(\overline{p}) \mathrel{\hat{=}} e$ and $f(\overline{p}') \mathrel{\hat{=}} e'$ satisfying the properties, then no substitutions η and η' satisfy $[\![f^{\mathrm{c}}(\overline{p}\eta)]\!] = [\![f^{\mathrm{c}}(\overline{p}'\eta')]\!]$ because the two rules of complement functions have the forms $f^{\mathrm{c}}(\overline{p}) \mathrel{\hat{=}} \mathsf{R}_k\langle\ldots\rangle$ and $f^{\mathrm{c}}(\overline{p}') \mathrel{\hat{=}} \mathsf{R}_l\langle\ldots\rangle$ with $l \neq k$. The rule $f(\overline{p}) \mathrel{\hat{=}} e$ has the corresponding rule of the complement function f^{c}

$$f^{\mathrm{c}}(\overline{p}) \mathrel{\hat{=}} R_k\langle\overline{e^{\mathrm{c}}}, \mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e)\rangle$$

where $\overline{e^{\mathrm{c}}}$ is obtained by $\Gamma_{\overline{p}}, e \overset{\mathrm{c}}{\leadsto} \overline{e^{\mathrm{c}}}$. Since we have $[\![e\eta]\!] = [\![e\eta']\!]$ and $[\![\overline{e^{\mathrm{c}}}\eta]\!] = [\![\overline{e^{\mathrm{c}}}\eta]\!]$, from the induction hypothesis, we have

$$\forall x \in \mathsf{vars}(e).\ \eta(x) = \eta'(x).$$

Since we also have $(\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e))\eta = (\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e))\eta'$ because $[\![f^{\mathrm{c}}(\overline{x}\theta)]\!] = [\![f^{\mathrm{c}}(\overline{x}\theta')]\!]$, we have

$$\forall x \in (\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e)).\ \eta(x) = \eta'(x).$$

Hence, we have

$$\forall x \in \mathsf{vars}(\overline{p}).\ \eta(x) = \eta'(x).$$

Since $\overline{x}\theta = \overline{p}\eta$ and $\overline{x}\theta' = \overline{p}\eta'$ hold, we have

$$\forall x \in \mathsf{vars}(f(\overline{x})).\ \theta(x) = \theta'(x).$$

It is worth noting that the proof for this case also shows that the lemma implies the statement

$$[\![f(\overline{u})]\!] = [\![f(\overline{v})]\!] \wedge [\![f^{\mathrm{c}}(\overline{u})]\!] = [\![f^{\mathrm{c}}(\overline{v})]\!] \Rightarrow \overline{u} = \overline{v}$$

which establishes Theorem 10. □

Proof of Theorem 11

For a function that is not in *INJ*, the proof is trivial because the complement derivation algorithm introduces the different tags for different rules. For a function in *INJ*, for any rule $f(\overline{p}) \mathrel{\hat{=}} e$ and $f^{-1}(p) \mathrel{\hat{=}} \overline{p}$ **where** W, we have $[\![p]\!] = \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(\overline{e})$, which guarantees that an obtained program is deterministic.

What is more interesting is that the above proof says that our derived inverse functions are deterministic after the optimizations proposed in the our previous work because we have $[\![p]\!] = \widetilde{\mathsf{ran}}_{\Gamma_{\overline{p}}}(\overline{e})$ for any rule $f(\overline{p}) \mathrel{\hat{=}} e$ and the corresponding inverse of the tupled function have the form $\langle f, f^{\mathrm{c}}\rangle^{-1}(p, p') \mathrel{\hat{=}} \overline{p}$ **where** W.

Proof of Theorem 12

We first define the semantics of program containing **where**.

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

$$\frac{\exists f(\overline{u}) \mathrel{\hat{=}} e \textbf{ where } W, \overline{p}\theta = \overline{v}.\ \theta(x) \in \Gamma_{\overline{p}}(x) \ \ W, \theta, e\theta \downdownarrows \overline{v}}{f(\overline{u}) \Downarrow \overline{v}}\textsc{Fun}$$

$$\frac{\mathsf{vars}(\overline{x}) \subseteq \mathsf{dom}(\theta), f(\overline{x}\theta) \Downarrow \overline{u}, \overline{p}\sigma = \overline{u} \quad W, \theta \cup \sigma, e\sigma \downdownarrows \overline{v}}{(\overline{p} \mathrel{\hat{=}} f(\overline{x})) \cup W, \theta, e \downdownarrows v}\text{W1}$$

$$\frac{e \Downarrow v}{\emptyset, \theta, e \downdownarrows \overline{v}}\text{W2}$$

We prove Theorem 12 by proving the following four lemmas.

lemma 2. For any expression e in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[e]$ in $f \in \mathit{INJ}$,

$$\begin{aligned}&\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W\\ &\Rightarrow (e\theta \Downarrow t, \forall x \in \mathsf{vars}(e).\ \theta(x) \in \Gamma(x) \Rightarrow \exists \eta.\ \eta p = t \wedge \forall x \in \mathsf{vars}(e).\ W, \eta, x\eta \downdownarrows \theta(x))\end{aligned}$$

holds

lemma 3. For any expression e

$$\begin{aligned}&\Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \wedge \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p' \wedge \Gamma, e, L \overset{\mathrm{w}}{\rightsquigarrow} W, (L \subseteq \mathsf{vars}(p'))\\ &\Rightarrow \begin{pmatrix} e\theta \Downarrow t, \overline{e^{\mathrm{c}}}\theta \Downarrow t', \forall x \in \mathsf{vars}(e).\ \theta(x) \in \Gamma(x) \\ \qquad \Rightarrow \exists \eta.\ \eta p = t, \eta p' = t' \wedge \forall x \in \mathsf{vars}(e).\ W, \eta, x\eta \downdownarrows \theta(x)\end{pmatrix}\end{aligned}$$

holds

lemma 4. For any expression e in $f(\overline{p}) \mathrel{\hat{=}} \mathcal{C}[e]$ in $f \in \mathit{INJ}$,

$$\begin{aligned}&\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W\\ &\Rightarrow \begin{pmatrix} e\theta \Downarrow t, \forall x \in \mathsf{vars}(e).\ \theta(x) \in \Gamma(x) \\ \qquad \Leftarrow \eta p = t \wedge \forall x \in \mathsf{dom}(p).\ \eta(x) \in \Gamma_p \\ \qquad\quad \wedge\, \theta = \{x \mapsto v \mid W, \eta, x\eta \downdownarrows v \mid x \in \mathsf{vars}(e)\} \wedge \mathsf{dom}(\theta) = \mathsf{vars}(e)\end{pmatrix}\end{aligned}$$

holds

lemma 5. For any expression e

$$\begin{aligned}&\Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \wedge \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p' \wedge \Gamma, e, L \overset{\mathrm{w}}{\rightsquigarrow} W, (L \subseteq \mathsf{vars}(p'))\\ &\Rightarrow \begin{pmatrix} e\theta \Downarrow t, \overline{e^{\mathrm{c}}}\theta \Downarrow t', \forall x \in \mathsf{vars}(e).\ \theta(x) \in \Gamma(x) \\ \qquad \Leftarrow \eta p = t, \eta p' = t', \forall x \in \mathsf{vars}(p).\ \theta(x) \in \Gamma_p(x) \\ \qquad\quad \wedge\, \theta = \{x \mapsto v \mid W, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e)\} \wedge \mathsf{dom}(\theta) = \mathsf{vars}(e)\end{pmatrix}\end{aligned}$$

holds

The above lemmas correspond to

- $[\![f(\overline{u})]\!] = t \Rightarrow [\![f^{-1}(t)]\!] = \overline{u} \qquad (f \in \mathit{INJ})$
- $[\![f(\overline{u})]\!] = t, [\![f^{\mathrm{c}}(\overline{u})]\!] = t' \Rightarrow [\![\langle t, t^{\mathrm{c}}\rangle^{-1}(t, t')]\!] = \overline{u} \qquad (f \notin \mathit{INJ})$
- $[\![f(\overline{u})]\!] = t \Leftarrow [\![f^{-1}(t)]\!] = \overline{u} \qquad (f \in \mathit{INJ})$
- $[\![f(\overline{u})]\!] = t, [\![f^{\mathrm{c}}(\overline{u})]\!] = t' \Leftarrow [\![\langle t, t^{\mathrm{c}}\rangle^{-1}(t, t')]\!] = \overline{u} \qquad (f \notin \mathit{INJ})$

respectively.

Proof of Lemma 2. We prove Lemma 2 by induction on the structure and the evaluation of e.

Base Case: Empty Forest

In this case, we have

$$\Gamma, \varepsilon \overset{\mathrm{p}}{\rightsquigarrow} \varepsilon \qquad \Gamma, \varepsilon, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} \emptyset \qquad \varepsilon\theta \Downarrow \varepsilon$$

for any θ. Then, the statement of the lemma trivially holds.

Base Case: Variable

In this case, we have

$$\Gamma, x \overset{\mathrm{p}}{\rightsquigarrow} x \qquad \Gamma, x, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} x \qquad x\theta \Downarrow t$$

for a substitution θ such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$ and a forest $t = \theta(x)$. Then, the statement of the lemma trivially holds.

Inductive Step: Constructor

In this case, we have

$$\begin{array}{ll} \Gamma, \sigma(e) \overset{\mathrm{p}}{\rightsquigarrow} \sigma(p) & \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \\ \Gamma, \sigma(e), \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W & \Gamma, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W \\ \sigma(e)\theta \Downarrow \sigma(t) & e \Downarrow t \end{array}$$

for a substitution θ and a forest t such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$. From the induction hypothesis, there exists η such that $p\eta = t$ and

$$\forall x \in \mathsf{vars}(e).\ W, \eta, x\eta \downdownarrows \theta(x).$$

Then, we have $\sigma(p)\eta = \sigma(t)$, which implies the statement of the lemma.

Inductive Step: Non-Horizontally-Overlapped Concatenation

In this case, we have

$$\begin{array}{lll} \Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \centerdot p_2 & \Gamma, e_1, \emptyset \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\ \Gamma, e_1 \centerdot e_2, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W_1 \cup W_2 & \Gamma, e_1, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2 \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W_2 \\ e_1 \centerdot e_2\theta \Downarrow t_1 \centerdot t_2 & e_1\theta \Downarrow t_1 & e_2\theta \Downarrow t_2 \end{array}$$

for a substitution θ and forests t_1 and t_2 such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1 \centerdot e_2)$. From the induction hypothesis, there exist substitutions η_1 and η_2 such that $p_1\eta_1 = t_1$ and $p_2\eta_2 = t_2$

$$\begin{array}{l} \forall x \in \mathsf{vars}(e_1).\ W_1, \eta_1, x\eta_1 \downdownarrows \theta(x), \\ \forall x \in \mathsf{vars}(e_2).\ W_2, \eta_2, x\eta_2 \downdownarrows \theta(x). \end{array}$$

Then, since we have $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$ and $\mathsf{vars}(W_1) \cap \mathsf{vars}(W_2) = \emptyset$, taking $\eta = \eta_1 \centerdot \eta_2$, we have $p\eta = t_1 \centerdot t_2$ and

$$\forall x \in \mathsf{vars}(p).\ W, \eta, x\eta \downdownarrows \theta(x),$$

which implies the statement of the lemma.

Inductive Step: Injective Function

Let k be the ID of an expression $f(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k :: T_k \qquad \Gamma, f(\overline{x}), \emptyset \overset{\mathrm{p}}{\rightsquigarrow} \overline{(x :: \Gamma(x))} \mathrel{\hat{=}} f^{-1}(w_k) \qquad f(\overline{x}\theta) \Downarrow t$$

for a forest t and a substitution θ such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(\overline{x})$. Since $f(\overline{x}\theta) \Downarrow t$ holds, there exists a rule $f(\overline{p}) \hat{=} e$ such that $\overline{p}\zeta = \overline{x}\theta$ and $e\zeta \Downarrow t$ for some substitution ζ, which also implies that there exist a rule $f^{-1}(p) \hat{=} \overline{p}$ **where** W where $\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} p$ and $\Gamma_{\overline{p}}, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W$. From the induction hypothesis, there exists a substitution ξ such that $p\xi = t$ and

$$\forall x \in \mathsf{vars}(\overline{p}).\ W, \xi, x\xi \downdownarrows \zeta(x).$$

Then, taking $\eta(w_k) = p\xi$, we have $w_k\eta = t$ and

$$\forall x \in \mathsf{vars}(e).\ W, \eta, x\eta \downdownarrows \theta(x),$$

which implies the statement of the lemma.

Note that the proof of this case also shows that Lemma 2 implies the following statement.

$$[\![f(\overline{u})]\!] = t \Rightarrow [\![f^{-1}(t)]\!] = \overline{u} \qquad (f \in \mathit{INJ})$$

This statement is what we want to prove by Lemma 2. □

Proof Lemma 3. Some part of the proof is omitted because the proof is similar to the proof of Lemma 2 except for two induction steps: horizontally-overlapped concatenation case and function call case.

Inductive Step: Horizontally-Overlapped Concatenation

Let k be the ID of an expression $e_1 \centerdot e_2$. We have

$$\begin{array}{lll}
\Gamma, e_1 \centerdot e_1 \overset{\mathrm{c}}{\rightsquigarrow} \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} & \Gamma, e_1 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_1{}^{\mathrm{c}}} & \Gamma, e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_2{}^{\mathrm{c}}} \\
\Gamma, \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} l_k, \overline{p_1}', \overline{p_2}' & \Gamma, \overline{e_1{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} \overline{p_1}' & \Gamma, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} \overline{p_2} \\
\Gamma, e_1 \centerdot e_2, \mathsf{vars}(l_k, \overline{p_1}', \overline{p_2}') \overset{\mathrm{w}}{\rightsquigarrow} W \cup W_1 \cup W_2 & \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\
W = \{(p_1, p_2) \hat{=} \textsc{spl}(l_k, v_k)\} & \Gamma, e_1, \mathsf{vars}(l_k, \overline{p_1}', \overline{p_2}') \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \mathsf{vars}(l_k, \overline{p_1}', \overline{p_2}') \overset{\mathrm{w}}{\rightsquigarrow} W_2 \\
(e_1 \centerdot e_2)\theta \Downarrow (t_1 \centerdot t_2) & e_1\theta \Downarrow t_1 & e_2\theta \Downarrow t_2 \\
(\textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}})\theta \Downarrow (l, t_1', t_2') & \overline{e_1{}^{\mathrm{c}}}\theta \Downarrow t_1' & \overline{e_2{}^{\mathrm{c}}}\theta \Downarrow t_2'
\end{array}$$

and

$$\Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} v_k$$

for a substitution θ and forests t_1 and t_2 such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$. From the induction hypothesis, there exist substitutions η_1 and η_2 such that $p_1\eta_1 = t_1 \wedge p_1'\eta_1 = t_1'$, $p_2\eta_2 = t_2 \wedge p_2'\eta_2 = t_2'$, and satisfying

$$\forall x \in \mathsf{vars}(e_1).\ W_1, \eta_1, x\eta_1 \downdownarrows \theta(x),$$
$$\forall x \in \mathsf{vars}(e_2).\ W_2, \eta_1, x\eta_2 \downdownarrows \theta(x).$$

Then, taking $\eta = \{v_k \mapsto t_1 \centerdot t_2, l_k \mapsto l\} \cup \eta_1 \cup \eta_2$, we have

$$\forall x \in \mathsf{vars}(e_1 \centerdot e_2).\ W \cup W_1 \cup W_2, \eta, x\eta \downdownarrows \theta(x),$$

which implies the statement of the lemma.

Inductive Step: Function Call

Let k be the ID of an expression $f(\overline{x})$, and k' the ID of the corresponding expression $f^{\mathrm{c}}(\overline{x})$ where $\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \qquad \Gamma, f^{\mathrm{c}}(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_{k'} \qquad \Gamma, f(\overline{x}) \overset{\mathrm{w}}{\rightsquigarrow} \{\overline{(x :: \Gamma(x))} \hat{=} \langle f, f^{\mathrm{c}} \rangle^{-1}(w_k, w_{k'})\}$$
$$f(\overline{x}\theta) \Downarrow t \qquad f^{\mathrm{c}}(\overline{x}\theta) \Downarrow t'$$

for a substitution θ and forests t, t' such that $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(\overline{x})$. Since $f(\overline{x}\theta) \Downarrow t$ and $f^{\mathrm{c}}(\overline{x}\theta) \Downarrow t'$ hold, there exist a rule

$$f(\overline{p}) \mathrel{\hat{=}} e$$

and

$$f^{\mathrm{c}}(\overline{p}) \mathrel{\hat{=}} \mathsf{R}_i\langle \overline{e^{\mathrm{c}}}, V\rangle$$

where $\Gamma_{\overline{p}}, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}}$ and $V = \overline{p} \setminus \mathsf{vars}(e)$, which also implies there exists a rule

$$\langle f, f^{\mathrm{c}}\rangle^{-1}(p, p') \mathrel{\hat{=}} \overline{p} \textbf{ where } W$$

where $\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} p$, $\Gamma_{\overline{p}}, \mathsf{R}_i\langle \overline{e^{\mathrm{c}}}, V\rangle \overset{\mathrm{p}}{\rightsquigarrow} p'$ and $\Gamma_{\overline{p}}, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W$. For these rules, there exists a substitution ζ such that $\overline{p}\zeta = \overline{x}\theta$, $e\zeta \Downarrow t$ and $\mathsf{R}_i\langle \overline{e^{\mathrm{c}}}, V\rangle\zeta \Downarrow t'$. From the induction hypothesis, there exist substitution ξ such that

$$\forall x \in \mathsf{vars}(e).\ W, \xi, x\xi \downdownarrows \zeta(x).$$

Then, taking $\eta = \{w_k \mapsto t, w_{k'} \mapsto t'\}$, we have

$$\forall x \in \mathsf{vars}(\overline{x}).\ W, \eta, x\eta \downdownarrows \theta(x),$$

which implies the statement of the lemma.

Note that the proof of this case also shows that Lemma 3 implies the following statement.

$$[\![f(\overline{u})]\!] = t, [\![f^{\mathrm{c}}(\overline{u})]\!] = t' \Rightarrow [\![\langle t, t^{\mathrm{c}}\rangle^{-1}(t, t')]\!] = \overline{u} \qquad (f \notin \mathit{INJ})$$

This statement is what we want to prove by Lemma 3. □

Proof of Lemma 4. We prove the Lemma 4 by induction on the evaluation of **where**-clauses and the structure of the expression that generates the **where**-clauses.

Base Case: Empty Forest

In this case, we have

$$\Gamma, \varepsilon \overset{\mathrm{p}}{\rightsquigarrow} \varepsilon \qquad \Gamma, \varepsilon, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} \emptyset.$$

Here, the statement of the lemma trivially holds.

Base Case: Variable

In this case, we have

$$\Gamma, x \overset{\mathrm{p}}{\rightsquigarrow} x :: \Gamma(x) \qquad \Gamma, x, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} \emptyset.$$

Let t be a forest and η a substitution such that $x\eta = t$ and $t \in \Gamma(x)$. Then, θ is defined as follows.

$$\theta = \{x \mapsto v \mid \emptyset, \eta, x\eta \downdownarrows v\} = \{x \mapsto t\}.$$

Therefore, $x\theta \Downarrow t$ and $\theta(x) \in \Gamma(x)$ hold, which implies the statement of the lemma.

Inductive Step: Constructor

In this case, we have

$$\begin{array}{ll} \Gamma, \sigma(e) \overset{\mathrm{p}}{\rightsquigarrow} \sigma(p) & \quad \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \\ \Gamma, \sigma(e), \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W & \quad \Gamma, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W. \end{array}$$

Let $\sigma(t)$ be a forest and η a substitution such that $\sigma(p)\eta = \sigma(t)$ and $\eta(x) \in \Gamma(x)$ for any x in $\mathsf{vars}(p)$, and θ be a substitution satisfying

$$\theta = \{x \mapsto v \mid W, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(\sigma(e))\}$$

and $\mathsf{dom}(\theta) = \mathsf{vars}(e)$. Since we have $\mathsf{vars}(\sigma(e)) = \mathsf{vars}(e)$, from the induction hypothesis, we have $e\theta \Downarrow t$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$. Therefore, $\sigma(e)\theta \Downarrow t$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$ hold, which implies the statement of the lemma.

Inductive Step: Non-Horizontally-Overlapped Concatenation

In this case, we have

$$\begin{array}{lll} \Gamma, e_1 \mathbin{.} e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \mathbin{.} p_2 & \text{where} \;\; \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\ \Gamma, e_1 \mathbin{.} e_2, \emptyset \overset{\mathrm{p}}{\rightsquigarrow} W_1 \cup W_2 & \text{where} \;\; \Gamma, e_1, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W_2. \end{array}$$

Let t_1, t_2 be forests and η a substitution such that $(p_1 \mathbin{.} p_2)\eta = t_1 \mathbin{.} t_2$ and $\eta(x) \in \Gamma(x)$ for any x in $\mathsf{vars}(p)$, and θ a substitution satisfying

$$\theta = \{x \mapsto v \mid W_1 \cup W_2, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e_1 \mathbin{.} e_2)\}$$

and $\mathsf{dom}(\theta) = \mathsf{vars}(e_1 \mathbin{.} e_2)$. Since our target language is affine, we have $\mathsf{vars}(W_1) \cap \mathsf{vars}(W_2) = \emptyset$, $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$ and $\mathsf{vars}(p_1) \cap \mathsf{vars}(p_2) = \emptyset$, which implies that θ_1 and θ_2 defined by

$$\theta_1 = \{x \mapsto \theta(x) \mid x \in \mathsf{vars}(e_1)\}, \quad \theta_2 = \{x \mapsto \theta(x) \mid x \in \mathsf{vars}(e_2)\}$$

satisfies

$$\theta_1 = \{x \mapsto v \mid W_1, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e_1)\}, \quad \theta_2 = \{x \mapsto v \mid W_2, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e_2)\}$$

and $\mathsf{dom}(\theta_1) = \mathsf{vars}(e_1)$ and $\mathsf{dom}(\theta_2) = \mathsf{vars}(e_2)$. Hence, from the induction hypothesis, we have

$$e_1\theta_1 \Downarrow t_1, \quad e_2\theta_2 \Downarrow t_2,$$

$\theta_1(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1)$ and $\theta_2(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_2)$. Since we have $\theta = \theta_1 \cup \theta_2$, we have $(e_1 \mathbin{.} e_2)\theta \Downarrow t_1 \mathbin{.} t_2$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1 \mathbin{.} e_2)$, which implies the statement of the lemma.

Inductive Step: Injective Function

Let k be the ID of an expression $f(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k :: T_f \quad \Gamma, f(\overline{x}), \emptyset \overset{\mathrm{w}}{\rightsquigarrow} \{(\overline{x :: \Gamma(x)}) \mathrel{\hat{=}} f^{-1}(w_k)\}.$$

Let t be a forest and η a substitution such that $w_k\eta = t$, and θ be a substitution satisfying

$$\theta = \{x \mapsto v \mid \{(\overline{x :: \Gamma(x)}) \mathrel{\hat{=}} f^{-1}(w_k)\}, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(\overline{x})\}$$

and $\mathsf{dom}(\theta) = \mathsf{vars}(e) = \mathsf{vars}(\overline{x})$. Since $\mathsf{dom}(\theta) = \mathsf{vars}(\overline{x})$, there exists a rule of f^{-1} of the form

$$f^{-1}(q) \mathrel{\hat{=}} \overline{p} \;\textbf{where}\; W$$

such that there exists a rule of f of the form

$$f(\overline{p}) \mathrel{\hat{=}} e$$

satisfying

$$\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} p \quad \Gamma_{\overline{p}}, e, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W,$$

$q\zeta = t$, $\overline{x}\theta = [\![f^{-1}(q\zeta)]\!]$ and $\zeta(x) \in \Gamma_{\overline{p}}$ for any $x \in \mathsf{vars}(\overline{p})$. Since $f^{-1}(t){\downarrow}$ holds, η satisfies

$$\overline{p}\{x \mapsto v \mid W, \zeta, x\zeta \downdownarrows v, x \in \mathsf{vars}(e)\} = [\![f^{-1}(t)]\!]$$

which implies $\mathsf{dom}(\{x \mapsto v \mid W, \zeta, x\zeta \downdownarrows v, x \in \mathsf{vars}(e)\}) = \mathsf{vars}(e)$. So, taking ξ as $\{x \mapsto v \mid W, \zeta, x\zeta \downdownarrows v, x \in \mathsf{vars}(e)\}$, from the induction hypothesis, we have $e\xi \Downarrow t$. Since $\overline{x}\theta = \overline{p}\xi$, $e\xi \Downarrow t$ and $\eta(w_k) = q\zeta$ hold, we have $e\theta \Downarrow t$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(f(\overline{x}))$, which implies the statement of the lemma.

It is worth noting that the proof also shows that Lemma 4 implies the following statement.

$$[\![f(\overline{u})]\!] = t \Leftarrow [\![f^{-1}(t)]\!] = \overline{u} \qquad (f \in \mathit{INJ})$$

This statement is what we want to prove by Lemma 4. □

Lemma 5. Some part of the proof is omitted because the proof is similar to the proof of Lemma 4 except for two induction steps: horizontally-overlapped concatenation case and function call case.

We prove Lemma 5 by induction on the evaluation of W and the expression e that generates W.

Inductive Step: Horizontally-Overlapped Concatenation

Let k be the ID of an expression $e_1 \centerdot e_2$. In this case, we have

$$\begin{array}{lll}
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{c}}{\rightsquigarrow} \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} & \Gamma, e_1 \overset{\mathrm{c}}{\rightsquigarrow} e_1{}^{\mathrm{c}} & \Gamma, e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_2{}^{\mathrm{c}}} \\
\Gamma, \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} l_k, p'_1, p'_2 & \Gamma, \overline{e_1{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p'_1 & \Gamma, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p'_2 \\
\Gamma, e_1 \centerdot e_2, \mathsf{vars}(l_k, p'_1, p'_2) \overset{\mathrm{w}}{\rightsquigarrow} W' \cup W_1 \cup W_2 & \Gamma, e_1, \mathsf{vars}(p'_1) \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \mathsf{vars}(p'_2) \overset{\mathrm{w}}{\rightsquigarrow} W_2 \\
W'_1 = \{(p_1, p_2) \mathrel{\hat{=}} \textsc{spl}(l_k, v_k)\} & \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2
\end{array}$$

and

$$\Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} v_k.$$

Let t and t' be forests, η a substitution such that $v_k\eta = t$ and $(l_k, p'_1, p'_2)\eta = t'$, and θ a substitution satisfying

$$\theta = \{x \mapsto v \mid W' \cup W_1 \cup W_2, \eta, x\eta \downdownarrows v\}$$

and $\mathsf{dom}(\theta) = \mathsf{vars}(e)$. Since a substitution θ is defined for all $x \in \mathsf{vars}(e)$, we safely assume that there exists a ζ such that $(p_1, p_2)\zeta = (t_1, t_2)$ where $t = t_1 \centerdot t_2$ and $\textsc{len}(t_1) = l_k\eta$. Since our target language is affine, we have $\mathsf{vars}(W_1) \cap \mathsf{vars}(W_2) = \emptyset$, $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$ and $\mathsf{vars}(p_1) \cap \mathsf{vars}(p_2) = \emptyset$, which implies that θ_1 and θ_2 defined by

$$\theta_1 = \{x \mapsto \theta(x) \mid x \in \mathsf{vars}(e_1)\}, \quad \theta_2 = \{x \mapsto \theta(x) \mid x \in \mathsf{vars}(e_2)\}$$

satisfies

$$\theta_1 = \{x \mapsto v \mid W_1, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e_1)\}, \quad \theta_2 = \{x \mapsto v \mid W_2, \eta, x\eta \downdownarrows v, x \in \mathsf{vars}(e_2)\}$$

and $\mathsf{dom}(\theta_1) = \mathsf{vars}(e_1)$ and $\mathsf{dom}(\theta_2) = \mathsf{vars}(e_2)$. Hence, from the induction hypothesis, we have

$$e_1\theta_1 \Downarrow t_1, \quad e_2\theta_2 \Downarrow t_2, \quad \overline{e_1{}^{\mathrm{c}}}\theta_1 \Downarrow p'_1\eta, \quad \overline{e_2{}^{\mathrm{c}}}\theta_2 \Downarrow p'_2,$$

$\theta_1(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_1)$ and $\theta_2(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e_2)$. Since we have $\theta = \theta_1 \cup \theta_2 \cup \{l_k \mapsto l_k\eta\}$, we have $(e_1 \centerdot e_2)\theta \Downarrow t$ and $(l_k, \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}})\theta = t'$, which implies the statement of the lemma.

Inductive Step: Function Call

Let k be the ID of an expression $f(\overline{x})$, and k' the ID of the expression $f^{\mathrm{c}}(\overline{x})$ that is obtained by $\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x}) \quad \Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \quad \Gamma, f^{\mathrm{c}}(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_{k'}$$
$$\Gamma, f(\overline{x}), L \overset{\mathrm{w}}{\rightsquigarrow} \{\overline{(x :: \Gamma(x))} \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(w_k, w_{k'})\}.$$

Let t and t' be forests, η a substitution such that $w_k\eta = t, w_{k'}\eta = t'$, and θ a substitution such that

$$\theta = \{x \mapsto v \mid \{\overline{(x :: \Gamma(x))} \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(w_k, w_{k'})\}, \eta, x\eta \downdownarrows v \mid x \in \mathsf{vars}(e)\}$$

and $\mathsf{dom}(\theta) = \mathsf{vars}(e)$. Since $\mathsf{dom}(\theta) = \mathsf{vars}(e)$, by the definition of $\downdownarrows$ and the $\langle f, f^{\mathrm{c}}\rangle^{-1}$, there exists a rule

$$\langle f, f^{\mathrm{c}}\rangle^{-1}(q, q') \mathrel{\hat{=}} \overline{p} \textbf{ where } W,$$

a rule for forward transformations

$$f(\overline{p}) \mathrel{\hat{=}} e,$$

and a rule for complement functions

$$f^{\mathrm{c}}(\overline{p}) \mathrel{\hat{=}} \mathsf{R}_l\langle \overline{e^{\mathrm{c}}}, \overline{p} \setminus \mathsf{vars}(e)\rangle$$

satisfying the following.

$$\Gamma_{\overline{p}}, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}}$$
$$\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} p \quad \Gamma_{\overline{p}}, \mathsf{R}_l\langle \overline{e^{\mathrm{c}}}, \overline{p} \setminus \mathsf{vars}(e)\rangle \overset{\mathrm{p}}{\rightsquigarrow} p'$$
$$\Gamma_{\overline{p}}, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W$$
$$\exists \zeta.\ p\zeta = w_k\eta = t, p'\zeta = w_{k'}\eta = t'.$$

From the induction hypothesis, there is a substitution ξ such that

$$\xi = \{x \mapsto v \mid W, \zeta, x\zeta \Downarrow\!\!\Downarrow v, x \in \mathsf{vars}(e)\}$$

satisfying $e\xi \Downarrow t$ and $\overline{e^{\mathrm{c}}}\xi \Downarrow \overline{s}$. Since we have

$$\overline{x}\theta = \overline{p}\zeta,$$
$$[\![f(\overline{x})\theta]\!] = [\![e\xi]\!] = t,$$
$$[\![f^{\mathrm{c}}(\overline{x})\theta]\!] = R_l\langle [\![\overline{e^{\mathrm{c}}}\xi]\!], (\mathsf{vars}(\overline{p}) \setminus \mathsf{vars}(e))\eta\rangle = t',$$

we have $f(\overline{x})\theta \Downarrow t$, $f^{\mathrm{c}}(\overline{x})\theta \Downarrow t'$ and $\theta(x) \in \Gamma(x)$ for any $x \in \mathsf{vars}(e)$, which implies the statement of this lemma.

Note that the proof of this case also shows that Lemma 5 implies the following statement.

$$f(\overline{u}) = t, f^{\mathrm{c}}(\overline{u}) = t' \Leftarrow \langle f, f^{\mathrm{c}}\rangle^{-1}(t, t') = \overline{u}$$

This statement is what we want to prove by Lemma 5. $\square$

A.9 Proof of Theorem 13

We prove the following lemma.

lemma 6. Let s' be a initial return value of a complement function. For a expression e in a rule r,

$$\begin{aligned}&\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \wedge \Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p' \wedge \Gamma, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W \\ &\Rightarrow \begin{pmatrix} p\eta = t, t' = \mathcal{C}[p']\eta, \mathcal{C}'[t'] = s', \forall w \in \mathsf{vars}(p).\ \eta(x) \in \Gamma_p(x), \forall x \in \mathsf{vars}(e), \exists v.\ W, \eta, x\eta \Downarrow\!\!\Downarrow v \\ \Rightarrow \forall w \in \mathsf{vars}(p).\ X^{w}_{r,t'} \overset{*}{\rightarrow} \eta(w) \end{pmatrix}\end{aligned}$$

holds.

Proof. We prove the lemma by induction on the evaluation of W and the structure of e that generates W.

Base Case: Empty Forest

In this case, since $\Gamma, \varepsilon \overset{\mathrm{p}}{\rightsquigarrow} \varepsilon$ and the expression ε contains no variable, the statement trivially holds.

Base Case: Variable

In this case, we have

$$\Gamma, x \overset{\mathrm{p}}{\rightsquigarrow} x :: \Gamma(x) \quad \Gamma, x \overset{\mathrm{c}}{\rightsquigarrow} \epsilon \quad \Gamma, \epsilon \overset{\mathrm{p}}{\rightsquigarrow} \epsilon \quad \Gamma, x, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} \emptyset.$$

Since we have a production rule $X^{x}_{r,t'} \rightarrow \Gamma(x)$ in $G^{t'}_{\mathrm{U}}$, the statement of the lemma holds.

Inductive Step: Constructor

In this case, we have

$$\Gamma, \sigma(e) \overset{\mathrm{p}}{\rightsquigarrow} \sigma(p) \quad \text{where} \quad \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} e$$
$$\Gamma, \sigma(e) \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \quad \text{where} \quad \Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}}$$
$$\Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p'$$
$$\Gamma, \sigma(e), \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W \quad \text{where} \quad \Gamma, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W.$$

Let t and t' be forests, η a substitution such that $t = p\eta$, $t' = \mathcal{C}[p']\eta$, $\eta(x) \in \Gamma_p(x)$ for any x in p and

$$\forall x \in \mathsf{vars}(\sigma(e)).\ \exists v.\ W, \eta, x\eta \downdownarrows v.$$

From the induction hypothesis, we have

$$\forall w \in \mathsf{vars}(p).\ X^w_{r,t'} \xrightarrow{*} \eta(w),$$

which also implies the statement of the lemma.

Inductive Step: Non-Horizontally-Overlapped Concatenation

In this case, we have

$$\begin{array}{lll}
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \centerdot p_2 & \text{where}\ \ \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} & \text{where}\ \ \Gamma, e_1 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_1{}^{\mathrm{c}}} & \Gamma, e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_2{}^{\mathrm{c}}} \\
\Gamma, (\overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}}) \overset{\mathrm{p}}{\rightsquigarrow} p_1', p_2' & \text{where}\ \ \Gamma, \overline{e_1{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_1' & \Gamma, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_2' \\
\Gamma, e_1 \centerdot e_2, \mathsf{vars}(p_1', p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_1 \cup W_2 & \text{where}\ \ \Gamma, e_1, \mathsf{vars}(p_1') \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \mathsf{vars}(p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_2
\end{array}$$

Let t and t' be forests, and η a substitution such that $t = (p_1 \centerdot p_2)\eta$, $t' = \mathcal{C}[(p_1', p_2')]\eta$, $\eta(x) \in \Gamma_p(x)$ for any $x \in \mathsf{vars}(p)$

$$\forall x \in \mathsf{vars}(e_1 \centerdot e_2).\ \exists v.\ W_1 \cup W_2, \eta, x\eta \downdownarrows v.$$

Since $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$ and $\mathsf{vars}(W_1) \cap \mathsf{vars}(W_2) = \emptyset$ hold because our target language is affine, we have

$$\forall x \in \mathsf{vars}(e_1).\ \exists v.\ W_1, \eta, x\eta \downdownarrows v, \quad \forall x \in \mathsf{vars}(e_2).\ \exists v.\ W_2, \eta, x\eta \downdownarrows v.$$

From the induction hypothesis, we have

$$\forall w \in \mathsf{vars}(p_1).\ X^w_{r,t'} \xrightarrow{*} \eta(w), \quad \forall w \in \mathsf{vars}(p_2).\ X^w_{r,t'} \xrightarrow{*} \eta(w).$$

Therefore, we have

$$\forall w \in \mathsf{vars}(p_1, p_2).\ X^w_{r,t'} \xrightarrow{*} \eta(w),$$

which implies the statement of the lemma.

Inductive Step: Horizontally-Overlapped Concatenation

The proof is almost the same as the above because the proof does not require the length information.

Inductive Step: Injective Function Call

Let k be the ID of an expression $f(\overline{x})$. In this case, we have

$$\begin{array}{c} \Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \quad \Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} \epsilon \\ \Gamma, f(\overline{x}), \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W \end{array}$$

where $W = \{\overline{(x :: \Gamma(x))} \mathrel{\hat{=}} f^{-1}(w_k)\}$. Let t be a forest and η be a substitution such that $w_k\eta = t$ and

$$\forall x \in \mathsf{vars}(f(\overline{x})).\ \exists v.\ W, \eta, x\eta \downdownarrows v.$$

In this case, since $\forall x \in \mathsf{vars}(\overline{x}), \exists v.\ W, \eta, x\eta \downdownarrows v$ holds, we safely assume that there exists a substitution ζ and a rule

$$r' = f^{-1}(q) \mathrel{\hat{=}} \overline{p} \textbf{ where } W'$$

such that $q\zeta = w_k\eta$ and

$$\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q \qquad \Gamma_{\overline{p}}, e \overset{\mathrm{w}}{\rightsquigarrow} W'$$

where e is a right-hand side of the corresponding rule of the form

$$f(\overline{p}) = e.$$

Since we have $\forall x \in \mathsf{vars}(\overline{p}), \exists v.\ W', \zeta, x\zeta \downdownarrows v$ because $f^{-1}(q\zeta){\downarrow}$ holds, from the induction hypothesis, we have

$$\forall w \in \mathsf{vars}(q).\ X^{w}_{r,\varepsilon} \xrightarrow{*} \zeta(w)$$

which implies

$$S^{\varepsilon}_{f} \rightarrow T_{r',\varepsilon}[\![q]\!] \xrightarrow{*} q\zeta$$

because the production of CFFGs is closed for substitution. Since there is a production rule $X^{w_k}_{r,t} \rightarrow S^{\varepsilon}_{f}$ in $G^{s'}_{\mathrm{U}}$ by definition, we have

$$X^{w_k}_{r,t'} \xrightarrow{*} \eta(w_k),$$

which implies the statement of the lemma.

Note that the proof for this case also shows that this lemma implies

$$S^{\varepsilon}_{f} \xrightarrow{*} t \Rightarrow f^{-1}(t){\downarrow}$$

for a function $f \in \mathit{INJ}$.

Inductive Step: Function Call

Let k be the ID of an expression $f(\overline{x})$, k' be the ID of the expression $f^{\mathrm{c}}(\overline{x})$ that is obtained by $\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x})$). In this case, we have

$$\begin{array}{c} \Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x}) \quad \Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \quad \Gamma, f^{\mathrm{c}}(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_{k'} \\ \Gamma, f(\overline{x}), L \overset{\mathrm{w}}{\rightsquigarrow} W \end{array}$$

where $W = \{\overline{(x :: \Gamma(x))} \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(w_k, w_{k'})\}$. Let t and t' be forests, and η a substitution such that $t = w_k\eta, t' = \mathcal{C}[w_{k'}]\eta$ and

$$\forall x \in \mathsf{vars}(f(\overline{x})), \exists v.\ W, \eta, x\eta \downdownarrows \eta(x).$$

Here, we safely assume that there exist a substitution ζ and a rule

$$r' = \langle f, f^{\mathrm{c}}\rangle^{-1}(q, q') \mathrel{\hat{=}} \overline{p} \textbf{ where } W'$$

such that $q\zeta = w_k\eta, q'\zeta = w_{k'}\eta$ and

$$\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q \quad \Gamma_{\overline{p}}, e \overset{\mathrm{w}}{\rightsquigarrow} W' \quad \Gamma_{\overline{p}}, \mathsf{R}_l\langle \overline{e^{\mathrm{c}}}, \overline{p} \setminus \mathsf{vars}(e)\rangle \overset{\mathrm{p}}{\rightsquigarrow} q'$$

where e is a right-hand side of the corresponding rule

$$f(\overline{p}) = e.$$

Since we have $\forall x \in \mathsf{vars}(\overline{p}), \exists v.\ W', \zeta, x\zeta \downdownarrows v$ because $\langle f, f^{\mathrm{c}}\rangle^{-1}(q\zeta, q'\zeta)\downarrow$ holds, from the induction hypothesis, we have

$$\forall w \in \mathsf{vars}(q, q')\ X^{w}_{r,\eta(w_{k'})} \xrightarrow{*} \zeta(w)$$

which implies

$$S_f^{\eta(w_{k'})} \to T_{r',\eta(w_{k'})}[\![q]\!] \xrightarrow{*} q\zeta$$

because the production of CFFGs is closed for substitution. Since there is a production rule $X^{w_k}_{r,t} \to S_f^{\eta(w_{k'})}$, we have

$$X^{w_k}_{r,t'} \xrightarrow{*} \eta(w_k).$$

which implies the statement of the lemma.

Note that the proof for this case also shows that the lemma implies

$$S_f^{t'} \xrightarrow{*} t \Rightarrow \langle f, f^{\mathrm{c}}\rangle^{-1}(t, t')\downarrow$$

for a function $f \notin \mathit{INJ}$.

□

A.10 Proof of Theorem 14

We prove the following lemma.

lemma 7. Let s' be a initial complement. For a expression e in a rule r,

$$\begin{array}{l} \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} p \wedge \Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \wedge \Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p' \wedge \Gamma, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W \\ \Rightarrow \begin{pmatrix} p\eta = t, \mathcal{C}[p']\eta = t', \forall w \in \mathsf{vars}(p).\ X^{w}_{r,t'} \xrightarrow{*} \eta(w) \\ \Rightarrow \forall w \in \mathsf{vars}(p).\ \eta(x) \in \Gamma(x) \\ \quad \wedge \forall x \in \mathsf{vars}(e), \exists v.\ W, \eta, x\eta \downdownarrows v \end{pmatrix} \end{array}$$

holds.

Proof. We prove the above lemma by induction on the structure of e and the derivation relation of $\xrightarrow{*}$.

Base Case: Empty Forest

The statement of the lemma trivially holds because $\mathsf{vars}(e) = \emptyset$.

Base Case: Variable

In this case, we have $\Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} x :: \Gamma(x)$. Let t be a forest such that $X^{x}_{r,t'} \to \eta(x)$, η be a substitution such that $t = x\eta$ and $t \in \Gamma(x)$, which implies $\eta(x) = t$. Since there exists a production rule $X^{x}_{r,t'} \to \Gamma(x)$, we have $\eta(x) \in \Gamma(x)$. Since we have $\emptyset, \eta, t \downdownarrows t$, the statement of the lemma holds.

Inductive Step: Constructor

In this case, we have

$$\begin{gathered}
\Gamma, \sigma(e) \overset{\mathrm{p}}{\rightsquigarrow} \sigma(p) \ \text{ where } \ \Gamma, e \overset{\mathrm{p}}{\rightsquigarrow} e \\
\Gamma, \sigma(e) \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \ \text{ where } \ \Gamma, e \overset{\mathrm{c}}{\rightsquigarrow} \overline{e^{\mathrm{c}}} \\
\Gamma, \overline{e^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p' \\
\Gamma, \sigma(e), \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W \ \text{ where } \ \Gamma, e, \mathsf{vars}(p') \overset{\mathrm{w}}{\rightsquigarrow} W.
\end{gathered}$$

Let t and t' be forests, and η a substitution such that $\sigma(p)\eta = t$, $\mathcal{C}[p']\eta = t'$ and $\forall w \in \overline{\sigma(p)}.\ \eta(w) \in \Gamma(w) \wedge X^{w}_{r,t'} \xrightarrow{*} \eta(w)$. Then, from the induction hypothesis, we have

$$\forall x \in \mathsf{vars}(e), \exists v.\ W, \eta, x\eta \downdownarrows v$$

which implies

$$\forall x \in \mathsf{vars}(\sigma(e)), \exists v.\ W, \eta, x\eta \downdownarrows v$$

i.e., the statement of the lemma.

Inductive Step: Non-Horizontally-Overlapped Concatenation

In this case, we have

$$\begin{array}{llll}
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_1 \centerdot p_2 & \text{where} & \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} & \text{where} & \Gamma, e_1 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_1{}^{\mathrm{c}}} & \Gamma, e_2 \overset{\mathrm{c}}{\rightsquigarrow} \overline{e_2{}^{\mathrm{c}}} \\
\Gamma, (\overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}}) \overset{\mathrm{p}}{\rightsquigarrow} p_1', p_2' & \text{where} & \Gamma, \overline{e_1{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_1' & \Gamma, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_2' \\
\Gamma, e_1 \centerdot e_2, \mathsf{vars}(p_1', p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_1 \cup W_2 & \text{where} & \Gamma, e_1, \mathsf{vars}(p_1') \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \mathsf{vars}(p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_2
\end{array}$$

Let t and t' be forests, and η a substitution such that $p_1 \centerdot p_2 \eta = t$, $\mathcal{C}[(p_1', p_2')]\eta = t'$ and $\forall w \in \mathsf{vars}(\sigma(p)).\ X^{w}_{r,t'} \xrightarrow{*} \eta(w)$. Then, from the induction hypothesis, we have

$$\begin{gathered}
\forall w \in \mathsf{vars}(p_1).\ \eta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(e_1), \exists v.\ W_1, \eta, x\eta \downdownarrows v \\
\forall w \in \mathsf{vars}(p_2).\ \eta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(e_2), \exists v.\ W_2, \eta, x\eta \downdownarrows v
\end{gathered}$$

which implies

$$\forall w \in \mathsf{vars}(p_1 \centerdot p_2).\ \eta(w) \in \Gamma(x) \wedge \forall x \in \mathsf{vars}(e_1, e_2), \exists v.\ W_1 \cup W_2, \eta, x\eta \downdownarrows v$$

because $\mathsf{vars}(W_1) \cup \mathsf{vars}(W_2) = \emptyset$ and $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$. Hence, the statement of the lemma holds.

Inductive Step: Horizontally-Overlapped Concatenation

Let k be the ID of an expression $e_1 \centerdot e_2$. In this case, we have

$$\begin{array}{lll}
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{c}}{\rightsquigarrow} \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} & \Gamma, e_1 \overset{\mathrm{c}}{\rightsquigarrow} e_1{}^{\mathrm{c}} & \Gamma, e_2 \overset{\mathrm{c}}{\rightsquigarrow} e_2{}^{\mathrm{c}} \\
\Gamma, \textsc{len}(e_1), \overline{e_1{}^{\mathrm{c}}}, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} l_k, p_1', p_2' & \Gamma, \overline{e_1{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_1' & \Gamma, \overline{e_2{}^{\mathrm{c}}} \overset{\mathrm{p}}{\rightsquigarrow} p_2' \\
\Gamma, e_1 \centerdot e_2 \overset{\mathrm{p}}{\rightsquigarrow} v_k & \Gamma, e_1 \overset{\mathrm{p}}{\rightsquigarrow} p_1 & \Gamma, e_2 \overset{\mathrm{p}}{\rightsquigarrow} p_2 \\
\Gamma, e_1 \centerdot e_2, \mathsf{vars}(l_k, p_1', p_2'), \overset{\mathrm{w}}{\rightsquigarrow} W' \cup W_1 \cup W_2 & \Gamma, e_1, \mathsf{vars}(l_k, p_1', p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_1 & \Gamma, e_2, \mathsf{vars}(l_k, p_1', p_2') \overset{\mathrm{w}}{\rightsquigarrow} W_2
\end{array}$$

where $W' = \{(p_1, p_2) \mathrel{\hat{=}} \textsc{spl}(l_k, v_k)\}$. Let t and t' be forests, and η a substitution such that $v_k\eta = t$, $\mathcal{C}[l_k, p_1', p_2']\eta = t'$ and $X^{v_k}_{r,t'} \xrightarrow{*} \eta(v_k)$. By the definition of $\rightarrow$, the only production rule of $X^{v_k}_{r,t'}$ is

$$X^{v_k}_{r,t'} \rightarrow (T_{r,t'}[\![p_1]\!] \wedge L_{\eta(l_k)})T_{r,t'}[\![p_2]\!]$$

which implies that $t = t_1 \centerdot t_2$ such that $\textsc{len}(t_1) = l_k\eta$ and there exists a substitution ζ satisfying

$$\forall w \in \mathsf{vars}(p_1).\ X^w_{r,t'} \xrightarrow{*} \zeta(w)$$
$$\forall w \in \mathsf{vars}(p_2).\ X^w_{r,t'} \xrightarrow{*} \zeta(w).$$

Then, from the induction hypothesis, we obtain

$$\forall w \in \mathsf{vars}(p_1).\ \eta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(e_1), \exists v.\ W_1, \eta, x\eta \Downarrow v$$
$$\forall w \in \mathsf{vars}(p_2).\ \eta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(e_2), \exists v.\ W_2, \eta, x\eta \Downarrow v$$

which implies

$$\begin{array}{l}\forall w \in \mathsf{vars}(v_k).\ \eta(w) \in \Gamma(w) \\ \wedge \forall x \in \mathsf{vars}(e_1, e_2), \exists v.\ W' \cup W_1 \cup W_2, \eta, x\eta \Downarrow v\end{array}$$

because $\mathsf{vars}(W_1) \cup \mathsf{vars}(W_2) = \emptyset$ and $\mathsf{vars}(e_1) \cap \mathsf{vars}(e_2) = \emptyset$.

Inductive Step: Injective Function Call

Let k be the ID of an expression $f(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \quad \Gamma, f(\overline{x}), \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W$$

where $W = \{\overline{(x :: \Gamma(x))} \mathrel{\hat{=}} f^{-1}(w_k)\}$. Let t be a forest, η a substitution such that $w_k\eta = t$ and $X^{w_k}_{r,t'} \xrightarrow{*} \eta(w_k)$. Since there exists the unique production rule $X^{w_k}_{r,t'} \to S^\varepsilon_f$ starting from $X^{w_k}_{r,t'}$, we have $S^\varepsilon_f \xrightarrow{*} \eta(w_k)$. We safely assume that $S^\varepsilon_f \to T_{r',\varepsilon}[\![q]\!] \xrightarrow{*} \eta(w_k)$ where

$$r' = f^{-1}(q) \mathrel{\hat{=}} \overline{p} \ \textbf{where}\ W'$$

derived from

$$f(\overline{p}) \mathrel{\hat{=}} e$$

by $\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q$ and $\Gamma_{\overline{p}}, \emptyset \overset{\mathrm{w}}{\rightsquigarrow} W$. In this case, we have $\forall w \in \mathsf{vars}(q).\ X^w_{r',\varepsilon} \xrightarrow{*} \zeta(w)$ where $q\zeta = \eta(w_k)$. Then, from the induction hypothesis, we obtain

$$\forall w \in \mathsf{vars}(q).\ \zeta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(\overline{p}), \exists v.\ W', \zeta, x\zeta \Downarrow v$$

which implies $f^{-1}(q\zeta)\downarrow$. Property 2 implies $f^{-1}(q\zeta) = \overline{v} \Rightarrow v_i \in \Gamma(x_i)$. Hence, we obtain

$$\forall x \in \mathsf{vars}(f(\overline{x})), \exists v.\ W, \eta, x\eta \Downarrow v.$$

Note that the existence of v above is obtained from Property 2.

It is worth noting that this proof also shows that the lemma implies

$$S^\varepsilon_f \xrightarrow{*} t \Rightarrow f^{-1}(t)\downarrow$$

for $f \in \mathit{INJ}$.

Inductive Step: Function Call

Let k be the ID of an expression $f(\overline{x})$, k' be the ID of the expression $f^{\mathrm{c}}(\overline{x})$ that is obtained by $\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x})$. In this case, we have

$$\Gamma, f(\overline{x}) \overset{\mathrm{c}}{\rightsquigarrow} f^{\mathrm{c}}(\overline{x}) \quad \Gamma, f(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_k \quad \Gamma, f^{\mathrm{c}}(\overline{x}) \overset{\mathrm{p}}{\rightsquigarrow} w_{k'}$$
$$\Gamma, f(\overline{x}), L \overset{\mathrm{w}}{\rightsquigarrow} W$$

where $W = \{\overline{(x :: \Gamma(x)) \mathrel{\hat{=}} \langle f, f^{\mathrm{c}}\rangle^{-1}(w_k, w_{k'})}\}$. Let t and t' be forests, and η a substitution such that $t = w_k\eta, t' = \mathcal{C}[w_{k'}]\eta$. Since there exists the unique production rule $X^{w_k}_{r,t'} \to S^{\eta(w_{k'})}_f$ starting from $X^{w_k}_{r,t'}$, we have $S^{\eta(w_{k'})}_f \xrightarrow{*} \eta(w_k)$. We safely assume that $S^{\eta(w_{k'})}_f \to T_{r',\eta(w_{k'})}[\![q]\!] \xrightarrow{*} \eta(w_k)$ where

$$\langle f, f^{\mathrm{c}}\rangle^{-1}(q, q') \mathrel{\hat{=}} \overline{p} \textbf{ where } W'$$

such that $q\zeta = \eta(w_k), q'\zeta = \eta(w_{k'})$ for some ζ and

$$\Gamma_{\overline{p}}, e \overset{\mathrm{p}}{\rightsquigarrow} q \quad \Gamma_{\overline{p}}, e \overset{\mathrm{w}}{\rightsquigarrow} W' \quad \Gamma_{\overline{p}}, \mathsf{R}_l\langle\overline{e^{\mathrm{c}}}, \overline{p} \setminus \mathsf{vars}(e)\rangle \overset{\mathrm{p}}{\rightsquigarrow} q'$$

where e is a right-hand side of the corresponding rule

$$f(\overline{p}) = e.$$

In this case, we have $\forall w \in \mathsf{vars}(q).\ X^{w}_{r',\eta(w_{k'})} \xrightarrow{*} \zeta(w)$. Then, from the induction hypothesis, we obtain

$$\forall w \in \mathsf{vars}(q).\ \zeta(w) \in \Gamma(w) \wedge \forall x \in \mathsf{vars}(\overline{p}), \exists v.\ W', \zeta, x\zeta \downdownarrows v$$

which implies $\langle f, f^{\mathrm{c}}\rangle^{-1}(q\zeta, q'\zeta)\downarrow$. From Property 2, we have $\langle f, f^{\mathrm{c}}\rangle^{-1}(q\zeta, q'\zeta) = \overline{v} \Rightarrow f(\overline{v})\downarrow \Rightarrow v_i \in \Gamma(x_i)$. Hence, we obtain

$$\forall x \in \mathsf{vars}(f(\overline{x})), \exists v.\ W, \eta, x\eta \downdownarrows v.$$

Note that the existence of v above is obtained from Property 2.

It is worth noting that this proof also shows that the lemma implies

$$S^{t'}_f \xrightarrow{*} t \Rightarrow \langle f, f^{\mathrm{c}}\rangle^{-1}(t, t')\downarrow$$

for $f \notin \mathit{INJ}$. □

B Forest Version of Mohri and Nederhof's Regular Approximation Algorithm

Algorithm (Regular Approximation)**.**
Input: A CFFG G.
Output: A strongly-regular CFFG G'.
Procedure:

1. For each non-terminal A in G, repeat the following.

2. If A does not have a rule $A \to f$ such that $f = \mathcal{C}[Bt]$ for some tree t where A and B are mutually defined, then, add all the production rules of A to G'.

3. Introduce a fresh non-terminal A', and add a rule $A' \to \varepsilon$ to G'.

4. For each rules of $A \to f$, repeat the following procedures 5–7.

5. If $f = \mathcal{C}[\overline{Bf}] \centerdot C_1 g_1 \centerdot \cdots \centerdot C_n g_n$ with $|\overline{Bf}| \geq 0, n \geq 0$ where $\mathcal{C}$ does not contain any hole that no empty forest follows, $A, \overline{B}, C_1, \ldots, C_n$ are mutually defined, and $g_1, \ldots, g_n$, $\overline{f}$ and $\mathcal{C}$ does not contain any non-terminal that is mutually defined with A, then add rules

$$A \to \mathcal{C}[\overline{B}]C_1 \quad C_1' \to g_1 C_2 \quad \ldots \quad C_n' \to g_n A'$$

   to G' and proceed to the step 7.

6. If $f = \mathcal{C}[\overline{Bf}]$ with $|\overline{Bf}| \geq 0$ where $\mathcal{C}$ does not contain any hole that no empty forest follows, $A, \overline{B}, \overline{C}$ are mutually defined, and $\overline{g}$, $\overline{f}$ and $\mathcal{C}$ does not contain any non-terminal that is mutually defined with A then add a rule

$$A \to \mathcal{C}[\overline{B}]A'$$

   to G' and proceed to the step 7.

7. For each B_i and f_i for each $i \in \{1, \ldots, |\overline{Bf}|\}$ in the step 6 or 7, apply the following sub-procedure.

Sub-Procedure (Input: B, f):

1'. Let $\mathcal{C}'$ be a context such that $f = \mathcal{C}'[\overline{Cg}]$ such that $\mathcal{C}'$ does not contain any hole that no empty forest follows, B and $\overline{C}$ are mutually defined, $\overline{g}$ and $\mathcal{C}'$ does not contain any non-terminal that is mutually defined with B.

2'. Add a rule $B' \to \mathcal{C}'[\overline{C}]$ to G'.

3'. Recursively apply this sub-procedure to C_i and g_i for each $i \in \{1, \ldots, |\overline{Cg}|\}$. □

C Conversion a obtained View Update Checker to a CFFG

C.1 Conversion to CFFG

Algorithm (Elimination of $\wedge$).
Input: A view update checker $G_{\mathrm{U}}^{s'}$
Output: A CFFG for view update checking
Procedure:

1. Replace each occurrence of $A \wedge L_k$ to A_k
2. Add production rules for A_k defined as

$$\Delta = \{A_k \to t' \mid t' \in T, t \rightsquigarrow_k T, A \to t \in G_{\mathrm{U}}^{s'}\}$$

where $\rightsquigarrow_k$ is defined by following derivation rules.

$$\frac{k = 0}{\varepsilon \rightsquigarrow_k \{\varepsilon\}}\textsc{Eps}_{k=0} \quad \frac{k \neq 0}{\varepsilon \rightsquigarrow_k \emptyset}\textsc{Eps}_{k\neq 0} \quad \frac{k = 1}{\sigma(A) \rightsquigarrow_k \{\sigma(A)\}}\textsc{Con}_{k=1} \quad \frac{k \neq 1}{\sigma(A) \rightsquigarrow_k \emptyset}\textsc{Con}_{k\neq 1}$$

$$\frac{k = m}{A \wedge L_m \rightsquigarrow_k \{A_k\}}\mathrm{NL}_{k=m} \quad \frac{k \neq m}{A \wedge L_m \rightsquigarrow_k \emptyset}\mathrm{NL}_{k\neq m} \quad \frac{}{A \rightsquigarrow_k \{A_k\}}\mathrm{N}$$

$$\frac{T = \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, t_1 \rightsquigarrow_{k_1} T_1, t_2 \rightsquigarrow_{k_2} T_2, k_1 + k_2 = k\}}{t_1 \centerdot t_2 \rightsquigarrow_k \{t_1' \centerdot t_2' \mid (t_1', t_2') \in T\}}\textsc{Cat}$$

3. Recursively remove the non-terminals that never produce a forest. □

C.2 Conversion to RFG

Consider the case when Condition 1 holds for an original program for forward transformations. Note that the algorithm in the previous section does not change the structure of production rules of CFFG. Hence, recursively substituting non-terminals of the form $X_{r,t}^{w}$ with the right-hand side of its production rule, we obtain a strongly-regular CFFG. Recall that a strongly-regular CFFG can be converted to the RFG of which languages are the same as the strongly-regular CFFG.