Banana Algebra: Compositional Syntactic Language Extension

Jacob Andersen[a], Claus Brabrand[b,*], David Raymond Christiansen[b]

[a] *The Alexandra Institute, Aabogade 34, 8200 Aarhus N, Denmark*
[b] *IT University of Copenhagen, Rued Langgards Vej 7, DK-2300 Copenhagen S, Denmark*

Abstract

We propose an algebra of languages and transformations as a means for extending languages syntactically. The algebra provides a layer of high-level abstractions built on top of *languages* (captured by context-free grammars) and *transformations* (captured by constructive catamorphisms).

The algebra is *self-contained* in that any term of the algebra specifying a transformation can be reduced to a constant catamorphism, before the transformation is run. Thus, the algebra comes "for free" without sacrificing the strong safety and efficiency properties of constructive catamorphisms.

The entire algebra as presented in the paper is implemented as the Banana Algebra Tool which may be used to syntactically extend languages in an incremental and modular fashion via algebraic composition of previously defined languages and transformations. We demonstrate and evaluate the tool via several kinds of extensions.

Keywords: Languages; transformation; syntactic extension; macros; context-free grammars; catamorphisms; bananas; algebra.

1. Introduction and Motivation

We propose a method of defining transformations between context-free languages that is particularly suited for defining language extensions. The method is *simple*, *incremental*, and *modular*, and the defined transformations are *safe* and *efficient*.

Extension is *simple* because we base ourselves on a well-proven and easy-to-use formalism for well-typed syntax-directed transformations known as *constructive catamorphisms*. These transformations are specified relative to a source and a target language which are defined via context-free grammars (CFGs). Catamorphisms have previously been studied and proven sufficiently expressive as a

*Corresponding author
Email addresses: jacob.andersen@alexandra.dk (Jacob Andersen), brabrand@itu.dk (Claus Brabrand), drc@itu.dk (David Raymond Christiansen)

means for extending a large variety of programming languages via transformation [1, 2, 3]. Hence, the main focus of this paper lies not so much in addressing the expressiveness and which transformations can be achieved as on showing how algebraic combination of languages and transformations results in highly modular and incremental language extension. *Incremental* and *modular* means that any previously defined languages or transformations may be composed algebraically to form new languages and transformations. *Safety* means that the tool statically guarantees that the transformations always terminate and only map syntactically legal input terms into syntactically legal output terms; *Efficiency* means that any transformation is guaranteed to run in linear time (in the size of input and generated output).

An important property of the algebra which is built on top of catamorphisms is that it is "self-contained" in the sense that any term of the algebra may be reduced to a constant catamorphism, at compile-time. This means that all high-level constructions offered by the algebra (including composition of languages and transformations) may be dealt with at compile-time, before the transformations are run, without sacrificing the strong safety and efficiency guarantees.

The algebraic nature of the system for composing languages and transformations allows users of the system to understand and reason about its behavior. Combining languages and transformations through addition, for example, has the expected associativity and commutativity properties.

Everything presented in the paper has been implemented in the form of the Banana Algebra Tool. The Banana Algebra Tool takes a transformation term of the algebra as an argument, analyzes it for safety, and then reduces it to a constant transformation which can later be used to efficiently transform an input program. This tool can perform a wide variety of non-trivial transformations, such as transformation between different languages (e.g., for prototyping lightweight domain-specific language compilers), transforming programs within a given language (e.g., the CPS transformation), and format conversion (e.g., converting BibTex to BibTeXML). However, in this paper we will focus on language extension for which we have the following usage scenarios in mind: 1) Programmers may extend existing languages with their own macros; 2) Developers may embed domain-specific languages (DSLs) in host languages; 3) Compiler writers may implement only a small core and specify the rest externally; and 4) Developers or teachers may define languages incrementally by stacking abstractions on top of each other. We will substantiate these usage claims in Sections 7 and 8.

The approach occupies a "sweet spot" where full-scale compiler generators as outlined in Section 9 are too cumbersome and where simpler techniques for syntactic transformation either lack the strong safety and performance guarantees of our approach, or do not have sufficient support for incremental and modular development of language extensions.

Our contributions include the design of an algebra of languages and transformations for incremental and modular syntactic language extension built on top of catamorphisms; a proof-of-concept tool and implementation capable of

working with concrete syntax; and an evaluation of the algebraic approach.

2. Syntactic Language Extension

We begin by presenting examples of the kinds of extensions that we wish to define. These extensions add new syntactic features to a language and define a translation from the extended language to the base language. The examples presented in this section are deliberately simple—see Section 8 for a more involved example. Our base language is the untyped λ-calculus, whose syntactic structure may be defined by the following datatype:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp
```

In the following, we will informally demonstrate two extensions of the λ-calculus: numerals and booleans. In Sections 3 and 4, the informal technique described here will be connected to theoretical tools that enable strong guarantees of performance, safety and compositionality.

2.1. Extension: Numerals

A common extension of the core λ-Calculus is that of numerals; the calculus is extended with a construction representing zero, and unary constructors representing the successor and predecessor of a numeral. These constructions may be combined to represent any natural number in unary encoding and for performing numeric calculations. The syntax of the calculus is then extended to the language, `LN`:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp  |
        Zero  |  Succ exp  |  Pred exp
```

We will now show how to transform the extended language, `LN`, into the core λ-Calculus, `L`, using a basic encoding of numerals which represents zero as the identity function ($\lambda z.z$), and a number n as follows:

$$\overbrace{\lambda\, s\,.\,\lambda\, s\,.\,\cdots\,\lambda\, s\,.}^{n\text{ lambdas}}\ \overbrace{\lambda\, z\,.\,z}^{zero}$$

There are many other possible encodings of numerals, including the more common choice of Church numerals. However, the encoding of `Pred` in Church numerals is significantly more verbose, and the details of the encoding are irrelevant to our demonstration of the technique. Therefore, we use the simpler alternative to illustrate the point. We can now define the translation from `LN` to `N`. In our examples, the "semantic brackets" $[\![$ and $]\!]$ on the left-hand side contain the term whose replacement is being defined while they contain terms to be recursively replaced on the right-hand side.

$$
\begin{array}{rcl}
[\![\texttt{Var } V]\!] & = & \texttt{Var } [\![V]\!] \\
[\![\texttt{Lam } V\ E]\!] & = & \texttt{Lam } [\![V]\!]\ [\![E]\!] \\
[\![\texttt{App } E_1\ E_2]\!] & = & \texttt{App } [\![E_1]\!]\ [\![E_2]\!] \\
[\![\texttt{Zero}]\!] & = & \texttt{Lam z (Var z)} \\
[\![\texttt{Succ } E]\!] & = & \texttt{Lam s } [\![E]\!] \\
[\![\texttt{Pred } E]\!] & = & \texttt{App } [\![E]\!] \texttt{ (Lam z (Var z))}
\end{array}
$$

The first three rules just trivially recurse through the input structure producing an identical output structure. Zero becomes the identity function, successor adds a "lambda s" in front of the encoding of the argument, and predecessor peels off one lambda by applying it to the identity function (note that the predecessor of zero is thus consequently defined as zero). This will, for instance, map `Succ Zero` to its encoding `Lam s (Lam z (Var z))`.

This example is an illustration of how language extension is a special case of language transformation. Additionally, it demonstrates a drawback to this approach - each case that is not modified in the extension must be tediously mapped to itself. In Section 4, we describe a means of eliminating this boilerplate.

2.2. Other Extensions

In addition to numerals, the core λ-Calculus may easily be extended with booleans (via nullary constructors `True` and `False`, and a ternary `If`) yielding a syntactically extended language `LB` which could then be transformed to `L` as follows:

$$
\begin{array}{rcl}
[\![\texttt{True}]\!] & = & \texttt{Lam a (Lam b (Var a))} \\
[\![\texttt{False}]\!] & = & \texttt{Lam a (Lam b (Var b))} \\
[\![\texttt{If } E_1\ E_2\ E_3]\!] & = & \texttt{App (App } [\![E_1]\!]\ [\![E_2]\!]\texttt{) } [\![E_3]\!]
\end{array}
$$

Note that we have omitted the three lines of "identity transformations" for variables, lambda abstraction, and application.

Along similar lines, the λ-Calculus could be further extended with addition, multiplication, negation, conjunction, lists, pairs, and so on, eventually converging on a full-scale programming language. To substantiate the claim that this forms an adequate basis for language extension, we have extended the λ-Calculus towards a language previously used in teaching functional languages; "Fun" (cf. Section 8). First, however, we formalize the above extension technique and explore its properties.

3. Bananas

Transformations as they are defined above, which specify a transformation function for each constructor of a datatype but allow only for simple structural recursion, are instances of what is known as a *catamorphism* (or, more colloquially, a *banana* [4]).

A banana is a generalization of the list folding higher-order function known from functional programming languages which processes a list and builds up a

return value. However, instead of working on lists, it works on any inductively defined datatype. Catamorphisms have a strong category theoretical foundation [4] which we will not explore in this paper; however, the fact that our transformations are catamorphisms is important to our analysis of performance and safety.

For readers not familiar with catamorphisms, it is sufficient for this paper to understand the behavior of their incarnation in programming languages. A catamorphism for a datatype associates each constructor of the datatype with a *replacement evaluation function*. When applied to an input term of the datatype, the catamorphism then performs a recursive descent on the input structure, effectively deconstructing it, and applies the replacement evaluation functions in a bottom-up fashion, recombining intermediate results to obtain the final output result.

To concretize this description, let us consider an inductively defined datatype, `list`, defining non-empty lists of natural numbers:

$$\texttt{list} \;=\; \texttt{Num}\ \mathbb{N} \;\mid\; \texttt{Cons}\ \mathbb{N}\ \texttt{*}\ \texttt{list}$$

The sum of the values in a list of numbers may easily be defined by a catamorphism that replaces the `Num`-constructor by the identity function on numbers ($\lambda n.n$) and the `Cons`-constructor by addition on numbers ($\lambda(n,l).n{+}l$), corresponding to the following recursive definition:

$$\begin{array}{rcl} [\![\texttt{Num}\ n]\!] & = & n \\ [\![\texttt{Cons}\ n\ l]\!] & = & n{+}[\![l]\!] \end{array}$$

One of the main advantages of catamorphisms is that recursion over the structure of the input is completely separated from the construction of the output. In fact, the recursion is completely determined by the input datatype and is for that reason often only specified implicitly. Since the sum catamorphism above maps terms of type `list` to natural numbers $\mathbb{N}$, it may be uniquely identified with its replacement evaluation functions; in this case with a replacement evaluation function for the `Num`-constructor of type $\mathbb{N} \to \mathbb{N}$ and a replacement function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ for `Cons`). Catamorphisms are often written in the so-called banana brackets "$(\!|\cdots|\!)$" [4], containing the replacement evaluation functions:

$$(\!|\ \ \lambda n.n\ \ ,\ \ \lambda(n,l).n{+}l\ \ |\!)$$

3.1. Language-Typed Bananas

Constructive catamorphisms are a restricted form of catamorphisms where only output-typed *reconstructors* are permitted as replacement evaluation functions, rather than arbitrary functions. Reconstructors are just constructor terms from (possibly different) inductively defined datatypes wherein the arguments to the constructive catamorphism may be used. For instance, we can transform the lists into binary trees of the `tree` datatype:

```
tree = Nil | Leaf ℕ | Node ℕ * tree * tree
```

using a constructive catamorphism:

$$\begin{array}{rcl} [\![\texttt{Num } n]\!] & = & \texttt{Leaf } n \\ [\![\texttt{Cons } n\ l]\!] & = & \texttt{Node } n \texttt{ Nil } [\![l]\!] \end{array}$$

Bananas can be assigned types in the same way as other functions. The above example would be assigned the type `list` $\rightarrow$ `tree`, while the sum banana would be assigned the type `list` $\rightarrow \mathbb{N}$. When the banana maps terms of one language (called the source language, written l_s) into terms of another language (called the target language, written l_t), we call the banana and its associated typing $l_s \rightarrow l_t$ a *language-typed banana*. The languages l_s and l_t can be given either as a datatype (at the abstract syntactic level) as above, or as a CFG (at the concrete syntactic level).

Although very simple, capable of trivial recursion only, previous work [1, 2] has shown that this kind of constructive catamorphisms provides a basis for programming language extension. In fact, the language extensions demonstrated in Section 2 are examples of constructive catamorphisms. In the next section, we investigate their safety and efficiency properties.

3.2. Safety and Efficiency

Constructive catamorphisms have a number of interesting properties: they can be statically verified for syntactic safety, they are guaranteed to terminate, and they always run in linear time.

Let $\mathcal{L}(l)$ denote the set of terms that can be derived from l. A constructive catamorphism, x, is said to be *syntactically safe* if it only produces syntactically valid output terms, $\omega_t \in \mathcal{L}(l_t)$, given syntactically valid input terms, $\omega_s \in \mathcal{L}(l_s)$:

$$\forall \omega \in \mathcal{L}(l_s) \Rightarrow x(\omega) \in \mathcal{L}(l_t)$$

In addition to a *language typing* ($l_s \rightarrow l_t$), we also define a *nonterminal typing* τ, which specifies how each nonterminal of the source language is mapped onto nonterminals of the target language.

If we name the source and target languages of the above example `Lists` and `Trees` respectively, the language typing then becomes "`Lists -> Trees`" and the nonterminal typing, τ, is "`[list -> tree]`". The nonterminal typing is written inside square brackets because there may be multiple nonterminals in play, in which case multiple mappings are written as a comma separated list inside the brackets.

A catamorphism, x, with language typing $l_s \rightarrow l_t$, nonterminal typing τ, and reconstructors c is written $(\!|\, l_s \rightarrow l_t\ [\tau]\ c \,|\!)$. In order to verify that this catamorphism is syntactically safe, one simply needs to check that each of the catamorphism's reconstructor terms (e.g., from the example in the previous subsection, "`Node` n `Nil` $[\![l]\!]$") are valid syntax, assuming that each of its argument usages (e.g., $[\![l]\!]$) are valid syntax of the appropriate type (in this case l has source type `list` which means that $[\![l]\!]$ has type $\tau(\texttt{list}) = \texttt{tree}$). We refer to [5] for a formal treatment of how to verify syntactic safety.

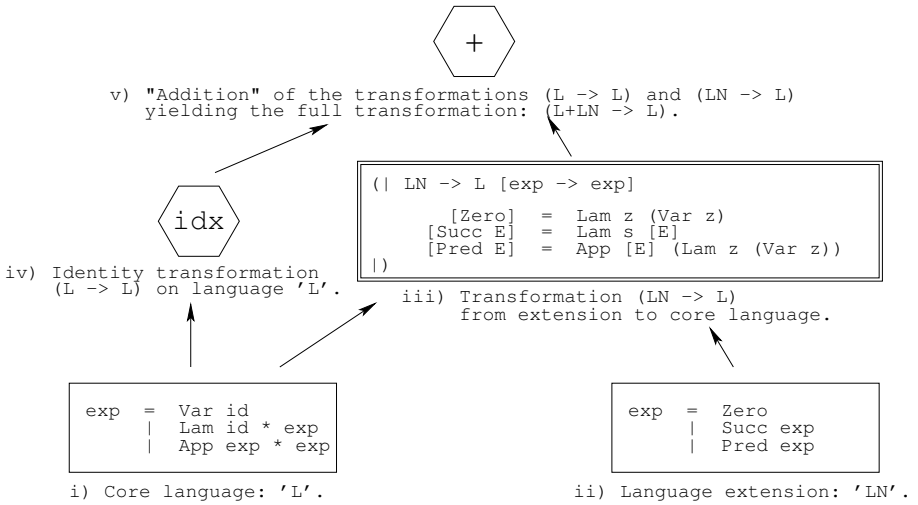

Figure 1: Common pattern in language extension (here extending the λ-Calculus with numerals.)

Constructive catamorphisms are highly efficient. In particular, if the replacement function has at most one recursive use of each sub-tree from the left-hand side, it runs in linear time in the size of the input. Many common transformations conform to this pattern, including all of the extensions to Java mentioned in Section 7. See Appendix C for proofs.

We have demonstrated that, with our simple type system, constructive catamorphisms form a safe, efficient technique for language extension. However, in order to be practical, we also desire a means of eliminating boilerplate code such as the identity transformations in our examples as well as a means of combining disparate extensions.

4. The Banana Algebra

Investigating previous work on syntactic macros and transformations [1, 2, 3] has revealed an interesting and recurring phenomenon in that macro extensions follow a certain pattern. The first hint in this direction is the effort involved in the first three lines of the constructive catamorphisms which are there merely to specify the "identity transformation" on the core λ-Calculus. That effort could be alleviated via explicit language support.

In fact, every such language extension can be broken into the same five ingredients, some of which are languages and some of which are transformations. Figure 1 depicts: i) a core language that is to be extended (e.g., the λ-Calculus); ii) an extension to that language[1] (e.g., the extension with numerals); iii) a

[1]Note that we refer to the extended language as excluding the core language.

transformation that maps the extended language to the core language; and iv) an identity transformation on the core language; v) a notion of "addition" of the identity transformation and the small transformation of the language extension to the core language.

4.1. The Algebra

The five ingredients above can be directly captured by five algebraic operators. First, cases i) and ii) correspond to a constant language, which is defined by a variation of context-free grammars. Second, case iii) corresponds to a constant transformation which may be given as an language-typed banana, c, typed with the source and target languages of the transformation (and a nonterminal typing, τ). Third, case iv) corresponds to an operator taking a language l and turning it into the identity transformation $(l \to l)$ on that language. Fourth, a notion of addition on transformations, taking two transformations $l_s \to l_t$ and $l'_s \to l'_t$ yielding a transformation: $(l_s \oplus_l l'_s) \to (l_t \oplus_l l'_t)$ where "$\oplus_l$" is addition on languages. Language addition is defined as the union of the individual productions (transformation addition as the union of the catamorphic reconstructors), which in both cases ensure that addition is idempotent, associative, and commutative. We refer to Appendix A (for a formal definition of languages) and to Appendix B (for transformations).

Note that with these operations, it is very easy to obtain a transformation combining *both* the extension of numerals and booleans; simply "add" the two transformations.

These operations form a *partial algebra* in the sense that there is a set, operations closed over the set, and a number of properties fulfilled by the operations. The set is our languages and transformations, the operations are those defined in the last paragraph, and the properties (such as commutativity and associativity of addition) are described in Section 5.4. Some of these operations are partial, because languages and transformations can "conflict", rendering them incompatible.

The point of structuring the system as an algebra is not to prove new theorems about it. The point is to provide users of the system with an understandable means for thinking about it. In this context, the algebraic properties (e.g., commutativity of addition) support the intuitions of the user and may assist in generating transformations programmatically.

The grammars that represent languages in the algebra are similar to ordinary context-free grammars, with two differences: there is no start nonterminal, and each alternative production for a nonterminal has a unique name. The lack of a start nonterminal enables language fragments representing extensions to be defined. Users of tools based on this approach are expected to provide a start nonterminal for the final resulting grammar. The naming of productions allows them to be referenced individually when combining extensions.

Although the above algebraic operations are enough to make all the extensions of the previous chapter, we would like to motivate a couple more algebraic operators on languages and transformations. We present the new operators

$[\text{NIL}_L]$	L	:	$\emptyset$	
$[\text{CON}_L]$		:	l	
$[\text{VAR}_L]$		:	v	
$[\text{RES}_L]$		:	$L \setminus L$	
$[\text{ADD}_L]$		:	$L + L$	
$[\text{SRC}_L]$		:	src (X)	
$[\text{TGT}_L]$		:	tgt (X)	
$[\text{LET}_L]$		:	let v=L in L	
$[\text{LTX}_L]$		:	letx w=X in L	
$[\text{REN}_L]$		:	L [R]	

(a) Algebra of languages (L)...

	X	:	0	$[\text{NIL}_X]$
		:	$(\!\| L \to L\ [\tau]\ c \|\!)$	$[\text{CON}_X]$
		:	w	$[\text{VAR}_X]$
		:	$X \setminus L$	$[\text{RES}_X]$
		:	$X + X$	$[\text{ADD}_X]$
		:	$X \circ X$	$[\text{SEQ}_X]$
		:	idx (L)	$[\text{IDX}_X]$
		:	let v=L in X	$[\text{LET}_X]$
		:	letx w=X in X	$[\text{LTX}_X]$
		:	X [R_s | R_t]	$[\text{REN1}_X]$
		:	X [| R_t]	$[\text{REN2}_X]$
		:	X [R_s |]	$[\text{REN3}_X]$

(b) ...transformations (X)...

R	:	m/n
	:	$q/n.p$

(c) ...and renamings (R).

Figure 2: Syntax of the Banana Algebra.

in two categories: operators accommodating respectively modular and incremental language extension. The complete syntax for the algebra is presented in Figure 2. $[\text{CON}_L]$ provides *language constants*, $[\text{CON}_X]$ provides *transformation constants*, $[\text{ADD}_L]$ provides *language addition*, $[\text{ADD}_X]$ provides *transformation addition*, and $[\text{IDX}_X]$ provides *identity transformations*. Of course, it is possible to add even more operators to the algebra; however, the ones we have turn out to be sufficient to conveniently extend the λ-Calculus incrementally all the way to the FUN programming language. These ideas are pursued in the remainder of the paper which also includes an evaluation of the whole algebraic approach. For a formal specification of the semantics of the algebra, see Section 5 along with Appendix A (for languages) and Appendix B (for transformations).

4.2. Modular language extension

To facilitate modular language development and allow reuse of languages and transformations, the Banana algebra has a local definition mechanism via the standard `let-in` functional programming local binder construction. Thus, we add to the syntax of both languages and transformations; *variables* (Figure 2, rules $[\text{VAR}_L]$ and $[\text{VAR}_X]$) and *local definitions* (Figure 2, rules $[\text{LET}_L]$, and $[\text{LTX}_X]$).

In practice, it turns out to be useful to also be able to define (local) transformations while specifying languages; and, orthogonally, to define (local) languages while specifying transformations. Hence, we add the local definitions $[\text{LTX}_L]$ and $[\text{LET}_X]$ (cf. Figure 2). Finally, we add a renaming construction as known from the λ-Calculus, for renaming the names of nonterminals and productions

in languages [REN$_L$]. In a grammar L, a nonterminal n may be renamed to m by: "L `[` m/n `]`". Similarly, an individual production p in nonterminal n may be renamed to q by: "L `[` $q/n.p$ `]`". For transformations we add similar renaming constructions [REN1$_X$], [REN2$_X$] and [REN3$_X$] with syntax "X `[` m_s/n_s `|` m_t/n_t `]`" which can explicit renamings for the source and target languages, respectively. The source or target part may be left empty. As with the language renaming, productions may be renamed as in: "X `[` $q/n.p$ `|` `]`" which renames production p in nonterminal n to q in the source language of transformation X. These renaming constructions are useful when using and extending languages or transformations defined by other programmers, which may use different naming conventions, and to avoid clashes.

4.3. Incremental language extension

Transformations are frequently specified incrementally in terms of previously defined languages and transformations. To accommodate such use we added a means for designating the source and target languages of a transformation along with a means for restricting a language and a transformation (i.e., restricting the source language of a transformation). By restriction, we take "L_1 `\` L_2" to yield a language identical to L_1, but where all productions also mentioned by name in L_2 have been eliminated. (The operators mentioned are listed as rules [SRC$_L$], [TGT$_L$], [RES$_L$], and [RES$_X$] of Figure 2.)

Also, transformations are frequently expressed via intermediate syntactic constructions for either simplicity or legibility. For instance, notice how two of the catamorphic reconstructors in the transformation of Section 2.1 both use the identity lambda abstraction `Lam z (Var z)`. Here, one could specify this transformation incrementally, by using an intermediary language, `LI`, enriched with identity as an explicit nullary construction:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp  |  Id
```

Although on such a small example, there is little to gain in terms of simplicity and/or legibility, it illustrates the general principle of incremental language extension. The transformation ("`LN -> L`") can now be simplified to "`ln2li: LN -> LI`":

$$\begin{array}{rcl} [\![\texttt{Zero}]\!] & = & \texttt{Id} \\ [\![\texttt{Succ } E]\!] & = & \texttt{Lam s } [\![E]\!] \\ [\![\texttt{Pred } E]\!] & = & \texttt{App } [\![E]\!] \texttt{ (Id)} \end{array}$$

Which is subsequently composed with the tiny transformation that desugares the identity-enriched language to the core λ-Calculus, "`li2l: LI -> L`":

$$[\![\texttt{Id}]\!] \quad = \quad \texttt{Lam z (Var z)}$$

Not surprisingly, when we do this experiment using the tool, the transformation "`li2l` $\circ$ `ln2li`" produces the exact same transformation as the directly specified constant transformation "`li2l`" (from Section 2.1). To enable such incremental development, we added composition as an operator on transformations (cf. Figure 2, rule [SEQ$_X$]).

4.4. Examples

Our definitions of extensions providing numerals and booleans for the λ-calculus in Section 2 can be improved using the banana algebra. We replace the identity cases with the identity transformation on the lambda calculus, we compose the extended languages explicitly from the lambda-calculus and the independent extensions, and we finally combine them to form a language with both extensions.

Recall our definition of the language `L`:

```
exp  =  Var id  |  Lam id * exp  |  App exp * exp
```

Omitting the productions of `L` from our original definition of `LN` gives us the following language, which represents unary natural numbers:

```
exp  =  Zero  |  Succ exp  |  Pred exp
```

Our transformation from the terms of `LN` to the terms of `L` can be written in language-typed catamorphism notation as follows:

```
(|  LN → L  [exp → exp]
    Zero()  =  Lam z (Var z),
    Succ(x)  =  Lam s x,
    Pred(x)  =  App x (Lam z (Var z))  |)
```

We can now reconstruct our first example transformation, which we call `LN2L`, without the boilerplate, as follows:

```
letx N2L = (|  LN → L  [exp → exp]
               Zero()  =  Lam z (Var z),
               Succ(x)  =  Lam s x,
               Pred(x)  =  App x (Lam z (Var z))  |)
           in  idx(L) + N2L
```

Similarly, our second example transformation, called `LB2L`, can be written without boilerplate:

```
letx B2L = (|  LB → L  [exp → exp]
               True()  =  Lam a (Lam b (Var a)),
               False()  =  Lam a (Lam b (Var b)),
               If(e1,e2,e3)  =  App (App e1 e2) e3  |)
           in  idx(L) + B2L
```

Finally, should we want to compose these extensions to obtain a λ-calculus with numbers and booleans, we can simply combine both extensions with the identity transformation on `L`:

```
letx N2L = (|  LN → L  [exp → exp]
               Zero()  =  Lam z (Var z),
               Succ(x)  =  Lam s x,
               Pred(x)  =  App x (Lam z (Var z))  |)
           in letx B2L = (|  LB → L  [exp → exp]
                             True()  =  Lam a (Lam b (Var a)),
                             False()  =  Lam a (Lam b (Var b)),
                             If(e1,e2,e3)  =  App (App e1 e2) e3  |)
                         in  idx(L) + N2L + B2L
```

The language typing of the resulting banana is `(L + LN + LB)` $\rightarrow$ `L`. Please note that we could have achieved the *exact same* transformation with the expression: `LN2L + LB2L`. Due to the algebraic properties of the language, both formulations of "the translation from the λ-calculus with booleans and numerals to the plain λ-calculus" are equivalent.

The language `I` contains only the identity function. That is, it is equivalent to `LI` from Section 4.3 with the productions of `L` removed, as follows:

```
exp  =  Id
```

Now, the example from the end of Section 4.3, which uses the explicit identity function as a intermediate translation for `Zero`, can be expressed as follows:

```
letx I2L = (|  I → L  [exp → exp]
               Id() = Lam z (Var z)  |)
         in letx LN2IL = (|  LN → (I+L)  [exp → exp]
                             Zero() = Id,
                             Succ(x) = Lam s x,
                             Pred(x) = App x id  |) + idx(L)
            in  (I2L + idx(L)) ∘ LN2IL
```

This example illustrates a general technique for defining transformations that make use of convenient intermediate representations that do not affect the execution speed of the transformation, as the entire term will reduce to a constant transformation in which the intermediate language does not occur. In fact, as we shall see in the next section, none of the operators of the banana algebra go beyond the expressivity of constructive catamorphisms in that any language term can be statically reduced to a context-free grammar; and any transformation term to a catamorphism.

5. Semantics of the Banana Algebra

In this section we give the semantics of the Banana Algebra. We have deliberately organized it such that anything at the level of the algebra is presented here in this section; and anything at the level below the algebra, that is, at the catamorphism level, is included in the appendices.

Let EXP_L denote the set of all language expressions from the syntactic category L (defined by Figure 2(a)); let EXP_X denote the set of all transformation expressions from the syntactic category X (defined by Figure 2(b)); let *CFG* denote the set of all *production-named context-free grammars* (defined below in Definition 1); and let *CATA* denote the set of all *transformations*, represented by output-typed constructive catamorphisms (defined below in Definition 2).

We exploit the aforementioned self-containedness property and give a big-step reduction semantics for the algebra capable of reducing a language expression, $L \in EXP_L$, to a constant language (context-free grammar), $l \in CFG$; and a transformation expression, $X \in EXP_X$, to a constant transformation (constructive catamorphism), $x = (\!|\, l_s \rightarrow l_t\ [\tau]\ c \,|\!) \in CATA$. Please refer to

$$[\textsc{nil}_L] \quad \frac{}{\alpha, \beta \vdash \emptyset \Downarrow_L l_0}$$

$$[\textsc{con}_L] \quad \frac{}{\alpha, \beta \vdash l \Downarrow_L l} \quad \vdash_{\texttt{wfl}} l$$

$$[\textsc{var}_L] \quad \frac{}{\alpha, \beta \vdash v \Downarrow_L \alpha(v)}$$

$$[\textsc{res}_L] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L \ \backslash \ L' \Downarrow_L l''} \quad l'' = l \ominus_l l',\ l \sim_l l'$$

$$[\textsc{add}_L] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L \texttt{ + } L' \Downarrow_L l''} \quad l'' = l \oplus_l l',\ l \sim_l l'$$

$$[\textsc{src}_L] \quad \frac{\alpha, \beta \vdash X \Downarrow_X (\!| l_s \to l_t\ [\tau]\ c |\!)}{\alpha, \beta \vdash \texttt{src (} X \texttt{)} \Downarrow_L l_s}$$

$$[\textsc{tgt}_L] \quad \frac{\alpha, \beta \vdash X \Downarrow_X (\!| l_s \to l_t\ [\tau]\ c |\!)}{\alpha, \beta \vdash \texttt{tgt (} X \texttt{)} \Downarrow_L l_t}$$

$$[\textsc{let}_L] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha[v \mapsto l], \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash \texttt{let } v\texttt{=}L \texttt{ in } L' \Downarrow_L l'}$$

$$[\textsc{ltx}_L] \quad \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta[w \mapsto x] \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash \texttt{letx } w\texttt{=}X \texttt{ in } L' \Downarrow_L l'}$$

$$[\textsc{ren}_L] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash L \texttt{ [} R \texttt{]} \Downarrow_L l'} \quad l' = l[R], R \sim_r l$$

Figure 3: Semantics of the algebra of languages.

subsections 5.1 and 5.2 for the notation for constant languages and constant transformations.

We take VAR to be the set of all variables. We define environments (which will deal with variables and the `let-in` constructions) in a straightforward way:

$$ENV_L = VAR \to CFG \qquad ENV_X = VAR \to CATA$$

The reduction semantics for the algebra of languages and transformations is defined by two mutually recursive relations, respectively:

$$\Downarrow_L \subseteq ENV_L \times ENV_X \times EXP_L \times CFG \qquad \Downarrow_X \subseteq ENV_L \times ENV_X \times EXP_X \times CATA$$

We will use the syntax "$\alpha, \beta \vdash L \Downarrow_L l$" as a shorthand for "$(\alpha, \beta, L, l) \in \Downarrow_L$" and "$\alpha, \beta \vdash X \Downarrow_X x$" as a shorthand for "$(\alpha, \beta, X, x) \in \Downarrow_X$". The two relations are defined in Figure 3 and Figure 4, respectively.

$$[\textsc{nil}_X] \quad \frac{}{\alpha, \beta \vdash \mathtt{0} \Downarrow_X x_0} \qquad [\textsc{var}_X] \quad \frac{}{\alpha, \beta \vdash w \Downarrow_X \beta(w)}$$

$$[\textsc{let}_X] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha[v \mapsto l], \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \mathtt{let}\ v{=}L\ \mathtt{in}\ X' \Downarrow_X x'} \qquad [\textsc{ltx}_X] \quad \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta[w \mapsto x] \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \mathtt{letx}\ w{=}X\ \mathtt{in}\ X' \Downarrow_X x'}$$

$$[\textsc{res}_X] \quad \frac{\alpha, \beta \vdash X \circ (\mathtt{idx}(\mathtt{src}(X)\ \backslash\ L)) \Downarrow_X x}{\alpha, \beta \vdash X\ \backslash\ L \Downarrow_X x} \qquad [\textsc{con}_X] \quad \frac{\alpha, \beta \vdash L_s \Downarrow_L l_s \qquad \alpha, \beta \vdash L_t \Downarrow_L l_t}{\alpha, \beta \vdash (\!| L_s \to L_t\ [\tau]\ c |\!) \Downarrow_X (\!| l_s \to l_t\ [\tau]\ c |\!)} \quad \vdash_{\mathtt{wfx}} (\!| l_s \to l_t\ [\tau]\ c |\!)$$

$$[\textsc{add}_X] \quad \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash X\ \mathtt{+}\ X' \Downarrow_X x''} \quad x'' = x \oplus_x x',\ x \sim_x x'$$

$$[\textsc{seq}_X] \quad \frac{\alpha, \beta \vdash X \Downarrow_X (\!| l_s \to l_t\ [\tau]\ c |\!) \qquad \alpha, \beta \vdash X' \Downarrow_X (\!| l'_s \to l'_t\ [\tau']\ c' |\!)}{\alpha, \beta \vdash X' \circ X \Downarrow_X (\!| l_s \to l'_t\ [\tau' \circ \tau]\ c'' |\!)} \quad c'' = c' \circ_c c,\ l_t \sqsubseteq_l l'_s$$

$$[\textsc{idx}_X] \quad \frac{\alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash \mathtt{idx}\ (\ L\ )\ \Downarrow_X (\!| l \to l\ [\tau]\ c |\!)} \quad \tau = id_\tau(l),\ c = id_c(l)$$

$$[\textsc{ren1}_X] \quad \frac{\alpha, \beta \vdash (\mathtt{idx}(\mathtt{tgt}(X)[R_t])[\bar{R}_t\ \mathtt{|}\ ] \circ X[R_s\ \mathtt{|}\ ] \Downarrow_X x}{\alpha, \beta \vdash X\ [R_s \mid R_t] \Downarrow_X x} \quad \bar{R}_t = \begin{cases} n\,/\,m & , \text{ if } R_t = m\,/\,n \\ p\,/\,n\,.\,q & , \text{ if } R_t = q\,/\,n\,.\,p \end{cases}$$

$$[\textsc{ren2}_X] \quad \frac{\alpha, \beta \vdash (\mathtt{idx}(\mathtt{tgt}(X)[R_t])[\bar{R}_t\ \mathtt{|}\ ] \circ X \Downarrow_X x}{\alpha, \beta \vdash X\ [\ \mathtt{|}\ R_t] \Downarrow_X x} \quad \bar{R}_t = \begin{cases} n\,/\,m & , \text{ if } R_t = m\,/\,n \\ p\,/\,n\,.\,q & , \text{ if } R_t = q\,/\,n\,.\,p \end{cases}$$

$$[\textsc{ren3}_X] \quad \frac{\alpha, \beta \vdash X \Downarrow_X (\!| l_s \to l_t\ [\tau]\ c |\!)}{\alpha, \beta \vdash X\ [\ R_s\ \mathtt{|}\ ] \Downarrow_X (\!| l'_s \to l_t\ [\tau']\ c' |\!)} \quad l'_s = l_s[R_s],\ \tau' = \tau[R_s],\ c' = c[R_s],\ R_s \sim_r l_s$$

Figure 4: Semantics of the algebra of transformations.

Note that the reduction semantics in Figures 3 and 4 uses a range of operators ($\vdash_{\texttt{wfl}}$, $\sim_l$, $\oplus_l$, $\ominus_l$, $\sqsubseteq_l$, $\sim_r$, $\vdash_{\texttt{wfx}}$, $\sim_x$, $\oplus_x$, $\circ_c$, id_τ, id_c, $[\cdot]$) which all operate on the level below that of the algebra; i.e., on constant languages (context-free grammars) and transformations (constructive catamorphisms). These operators are all formally defined as implemented in our tool in Appendix A.1 (for languages) and Appendix B.3 (for transformations).

5.1. Semantics of Languages

Languages in the Banana Algebra are represented by *production-named context-free grammars*, which are defined as follows:

Definition 1 (Production-named CFG)**.** *A* production-named context-free grammar*, G, is a tuple $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ where:*

- $\mathcal{N}$ *is a finite set of "nonterminals",*
- Σ *is a finite set of symbols (the "alphabet"),*
- $\mathcal{P}$ *is a finite set of "production names"; and*
- $\pi : \mathcal{N} \to 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*}$ *is the "production function".*

Note that a nonterminal $n \in \mathcal{N}$ *either* generates a general context-free language with a production name linked to each production rule ($\pi(n) \in 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} \setminus \{\emptyset\}$) *or* the nonterminal "type" is *undefined* ($\pi(n) = \emptyset \in 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*}$). This allows grammar definitions that use nonterminals without defining them. Usually such a grammar would be regarded as "improper", as such nonterminals serve no purpose; however, in the Banana Algebra, this feature allows grammars to reference fragments of other languages with which they will later be composed.

The nil language $[\textsc{nil}_L]$ is written $\emptyset$ and is the context-free grammar, l_0, which has no terminals, no nonterminals, and no productions (see Appendix A, Definition 5). While seemingly useless, it is included because it is the algebraic identity under addition of languages (see Section 5.4) and, therefore, it supports user intuitions about the system. Additionally, we expect that this element may be useful if transformations are to be generated programmatically.

Language restriction $[\textsc{res}_L]$ "$\ominus_l$" subtracts a language from another, structurally. The result is equal to the first argument language, but where all the productions named in the second argument language have been removed (cf. Appendix A, Definition 8).

Addition $[\textsc{add}_L]$ "$\oplus_l$" is defined as reducing both operands to constant languages, l and l', and constructing the language "$\oplus_l$" which has the union of their nonterminals, terminals, and productions (cf. Appendix A, Definition 7). Note that language restriction as well as addition are only well-defined if the two operands are *addition compatible* (written "$l \sim_l l'$", cf. Appendix A, Definition 6); i.e., that they do not define different production right-hand-sides for the same production names.

For renaming a language $[\textsc{ren}_L]$, the expression "$L[m/n]$" yields the language L, but where nonterminal n has been renamed to m, provided that m does

not occur in the nonterminal alphabet of L. The formal details of how to rename a constant language are given in Appendix A (Definitions 10 and 11 give the details for renaming nonterminals and productions through the operators, respectively).

5.2. Semantics of Transformations

Definition 2 (Transformation)**.** *A transformation is captured by an* output-typed constructive catamorphism*, x, which is a tuple $x = (l_s, l_t, \tau, c)$ where:*

- $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \in CFG$ *is the "source language grammar",*
- $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t) \in CFG$ *is the "target language grammar",*
- $\tau : \mathcal{N}_s \to \mathcal{N}_t$ *is the "nonterminal typing", and*
- $c : \mathcal{N}_s \to 2^{\mathcal{P}_s \times \mathcal{A}_{l_t}}$ *is the "reconstructor function".*

The transformation $x = (l_s, l_t, \tau, c)$ can be written $(\!| l_s \to l_t \ [\tau] \ c |\!)$.

Here, we describe the replacement evaluation function, c, using *reconstructors*: a nameless representation in which the evaluated replacements for the subterms on the left-hand-side are referred to by the natural numbers instead of by named variables.

Definition 3 (Reconstructor)**.** *Given a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in CFG$, the set $\mathcal{A}_l$ of* reconstructors for l *is given by the (least fixed point of the) following recursive set definition:*

$$\mathcal{A}_l = (\mathcal{N} \times \mathcal{P} \times {\mathcal{A}_l}^*) \cup \mathbb{N}$$

This set defines a tree data structure, which can be viewed as abstract syntax trees of the grammar l, but with *gaps*. An $\mathcal{A}_l$ is a node (inner or leaf) of such a tree. An inner node will have one or more children nodes – the list, ${\mathcal{A}_l}^*$, will be non-empty. A leaf node is either a grammar production with no nonterminals (${\mathcal{A}_l}^*$ is empty) or a gap. A gap is a "hole" in the tree where the root of another tree can be attached. Gaps are represented by the natural numbers, where each value may be mapped to a tree.

The nil transformation $[\textsc{nil}_X]$ is written `0` and is the catamorphism, x_0, from the nil language to the nil language, with empty nonterminal typing and empty set of reconstructors (see Appendix B, Definition 19). This construction is mainly included because it is the algebraic identity under addition of transformations. As with the nil language, we expect that the nil transformation may be useful when using the Banana Algebra as a target language for code generation.

For transformation restriction $[\textsc{res}_X]$, the algebraic term "$X \setminus L$" is equivalent to: "$X \circ$ `idx(src(`X`) \ `L`)`".

Addition of transformation expressions $[\textsc{add}_X]$ is defined as reducing both operands to constant transformations, $(\!| l_s \to l_t \ [\tau] \ c |\!)$ and $(\!| l'_s \to l'_t \ [\tau'] \ c' |\!)$, and adding those (using "$\oplus_x$") which yields a transformation from language

$l_s \oplus_l l'_s$ to language $l_t \oplus_l l'_t$ that has the union of the nonterminal types and reconstructors (cf. Appendix B, Definition 21). Addition is only well-defined if the two operands are addition-compatible (written "$x \sim_x x'$"); i.e., the source and target languages are addition-compatible and the transformations do not define different typings for the same nonterminals, nor do they define different reconstructors for the same productions (cf. Appendix B, Definition 20). This property ensures that well-formed transformations are closed under addition (cf. Appendix B, Proposition 10).

Two transformations are sequentially composed $[\texttt{SEQ}_X]$ by reducing both to constants, $(\!| l_s \to l_t\ [\tau]\ c |\!)$ and $(\!| l'_s \to l'_t\ [\tau']\ c' |\!)$, and constructing the composed transformation from l_s directly to l'_t with typing $\tau' \circ \tau$ and reconstructors $c' \circ_c c$ (cf. Appendix B, Definition 22). Obviously, the target language of the first transformation must not be larger than the source language of the second; i.e., $l_t \sqsubseteq l'_s$ (cf. Appendix A, Definition 12).

For renaming a transformation ($[\texttt{REN1}_X]$-$[\texttt{REN3}_X]$), the expression $X\,\texttt{[}R_s\ \texttt{|}\ R_t\texttt{]}$ does the renamings in the source and target language, respectively. The formal details of how to rename the source language of a constant transformation are given in Appendix B (Definitions 24 and 25 give the details for renaming the typing and reconstructor, respectively). Renaming the target language of a transformation can be defined through two source language transformation renamings and a language renaming, combined with a few other algebraic operators:

$$X\,\texttt{[}R_s\ \texttt{|}\ R_t\texttt{]} = \texttt{(idx(tgt(}X\texttt{)[}R_t\texttt{])[}R_t\ \texttt{|}\ \texttt{]}\ \circ\ X\,\texttt{[}R_s\ \texttt{|}\ \texttt{]}$$

5.3. Well-formedness

To avoid nonsensical grammars and to ensure syntactic safety of transformations, we introduce a notion of *well-formedness* and show that it is preserved under the semantics of the Banana Algebra.

Definition 4 (Well-formed grammar $\vdash_{\texttt{wfl}}$)**.** *A grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in$ CFG, is said to be "well-formed" (written "$\vdash_{\texttt{wfl}} l$") iff $\forall n \in \mathcal{N}$:*

(i) $$\forall (p, \alpha), (p', \alpha') \in \pi(n) : p = p' \Rightarrow \alpha = \alpha'$$

(ii) $$\pi(n) \neq \emptyset \ \vee\ \Big(\exists n' \in \mathcal{N}; p \in \mathcal{P}; \alpha, \alpha' \in (\mathcal{N} \cup \Sigma)^* \ :\ \pi(n') = (p, [\alpha n \alpha'])\Big)$$

In other words, a grammar is well-formed if there are no two productions with the same production name, but different right-hand-sides. Additionally, $\mathcal{N}$ may not include any "useless" nonterminal symbols, that is, nonterminal symbols that neither have defined productions nor occur in the right-hand-side of other productions.

Similarly to grammars, we define a well-formedness relation on transformations. The transformation, $x = (l_s, l_t, \tau, c) \in CATA$, is said to be "well-formed" (written "$\vdash_{\texttt{wfx}} x$") iff all of the following conditions are met:

- The source and target language grammars, l_s and l_t, must be well-formed;
- The transformation must be defined uniquely for every production defined in the source language grammar; and

- The transformation must adhere to the non-terminal typing τ.

The formal definition of this well-formedness relation relies on a number of relatively tedious sub-definitions. Therefore, it is formalized in Appendix B (Definition 16). The reconstructors of well-formed transformations always produce valid syntax of the output language, assuming that all arguments are valid syntax of nonterminal types as dictated by the nonterminal typing, τ. Well-formed transformations correspond to constructive catamorphisms.

As each rule given in Figures 3 and 4 preserves well-formedness, by induction, any expression in which all language and transformation constants are well-formed will reduce to a well-formed constant (assuming of course that all contents of the initial α and β environments, if any, are well-formed).

It is straightforward to show, using Propositions 1-6 and 9-13 in the appendix, that Banana Algebra terms whose constant languages and transformations are well-formed and that respect the various restrictions such as addition compatibility and nonterminal typing can be reduced to well-formed language or transformation terms.

5.4. Algebraic Laws

An important advantage of an algebraic approach is that several algebraic laws hold which give rise to simplifications; e.g., related to "`+`", we have:

L `+` L	$\equiv$	L	idempotency of "`+`"
L_1 `+` L_2	$\equiv$	L_2 `+` L_1	commutativity of "`+`"
L_1 `+` (L_2 `+` L_3)	$\equiv$	(L_1 `+` L_2) `+` L_3	associativity of "`+`"

These laws hold since "`+`" corresponds to the union of productions. Being an algebra, we have added the identity element with respect to "`+`", both for languages and transformations (cf. $\lceil \text{NIL}_L \rceil$ and $\lceil \text{NIL}_X \rceil$); i.e.: "$L$ `+` $\emptyset \equiv L \equiv \emptyset$ `+` L".

A number of interesting algebraic laws hold; here we provide only a few:

`src(idx(`L`))`	$\equiv$	L	`src-idx` identity
`let` v`=`L `in` v	$\equiv$	L	trivial `let-in` naming
$X_1 \circ (X_2 \circ X_3)$	$\equiv$	$(X_1 \circ X_2) \circ X_3$	associativity of "$\circ$"

Often algebraic laws give rise to simplifications and optimizations through algebraic rewriting. In our case, however, we do not need such rewritings, since we have the powerful property that *any* algebraic term can be reduced to a constant. Nonetheless, these algebraic laws do help developers in the process of writing languages and transformations; especially when re-using languages and transformations designed by others. For example, programmers can rely on the commutativity of transformation addition when composing language extensions.

6. Tool and Implementation

In order to validate the algebraic approach, we have implemented everything in the form of The Banana Algebra Tool which we have used to experiment with different forms of language extensions.

Below we focus on the interesting tool and implementation issues that arise as a consequence of our approach, such as ambiguity, the choice of using abstract vs. concrete syntax, hygiene, parsing, unparsing, debugging, and error-reporting.

6.1. Ambiguity

For reasons of usability which are discussed in Section 6.2, the Banana Algebra Tool allows concrete syntax to be used in the definitions of transformations. Ambiguities in grammars pose a serious problem when moving between abstract and concrete syntax, as it is possible that the AST resulting from converting an AST to concrete syntax and then parsing it may not be equivalent to the original AST. However, if the grammar of a language is unambiguous and if we choose a canonical unparsing of its abstract syntax, (see Section 6.4), we may move reversibly between its abstract syntax trees and concrete syntactic program strings without loss of information [3].

Note that ambiguous grammars do not exclusively occur at the outermost source and target languages of a transformation expression, X, but also within it. An ambiguity may for instance be present in the grammar of the intermediate step in a composition of two transformations $X_2 \circ X_1$; i.e., as the output term of X_1 is subsequently handed as input term to X_2.

We rely on a recent ambiguity analysis [6] for analyzing whether all of the grammars involved are unambiguous. In that work, context-free grammar ambiguity is fully characterized in that a grammar G is unambiguous if and only if it is *horizontally unambiguous* and *vertically unambiguous*.

A key feature of these two properties is that they are defined in terms of the structure of the grammar: Vertical unambiguity means that two different productions for the same nonterminal ($n \to \alpha_1$ and $n \to \alpha_2$) never derive the same string; i.e., $\mathcal{L}(\alpha_1) \cap \mathcal{L}(\alpha_2) = \emptyset$. Horizontal ambiguity means that the right-hand side of a production ($n \to \alpha$) can never be split in two non-empty parts $\alpha = \alpha_1\alpha_2$ whose languages overlap; i.e., $\mathcal{L}(\alpha_1) \between \mathcal{L}(\alpha_2) = \emptyset$ where $\between$ is the *language overlap* operator defined by $X \between Y = \{\ xay \mid x, y \in \Sigma^* \wedge a \in \Sigma^+ \wedge x, xa \in X \wedge y, ay \in Y\}$.

It has been known since 1962 that the ambiguity problem for context-free grammars is undecidable [7]. However, by over-approximating the context-free languages with possibly larger regular languages, the above characterization gives rise to an effective conservative ambiguity analysis. The analysis is sound in that if there are ambiguities, then they are indeed caught by the analysis; but, it is also incomplete in that the analysis may also produce "spurious ambiguity errors" (which are not ambiguities, but arise from inherent approximations in the analysis). All we need to know here is that the analysis works well in practice, gives good error messages in terms of the grammar analyzed, and does not often give spurious ambiguity errors [6].

Interestingly, the above characterization can be used to tell us exactly which operators in the Banana Algebra are capable of introducing ambiguities. Obviously, ambiguous context-free grammars may be specified directly as constant

```
Exp.or    : Exp1 "||" Exp  ;
   .exp1  : Exp1           ;
Exp1.and  : Exp2 "&&" Exp1 ;
    .exp2 : Exp2           ;
Exp2.add  : Exp3 "+" Exp2  ;
    .exp3 : Exp3           ;
...
Exp7.neg  : "!" Exp8       ;
    .exp8 : Exp8           ;
Exp8.par  : "(" Exp ")"    ;
    .var  : Id             ;
    .num  : IntConst       ;
```

(a) Java grammar fragment.

```
Stm.repeat =
  Stm.do($1,
    Exp.exp1(
      Exp1.exp2(
        Exp2.exp3(
          Exp3.exp4(
            Exp4.exp5(
              Exp5.exp6(
                Exp6.exp7(
                  Exp7.neg(
                    Exp8.par($2)
                  ))))))))) ;
```

(b) *Abstract* syntax.

```
Stm.repeat =
  'do $1 while (!($2));' ;
```

(c) *Concrete* syntax.

Figure 5: Example specifying transformations using *abstract* vs. *concrete* syntax. (For emphasis, we have underlined the negation and parenthesis constructions.)

languages, $[\text{CON}_L]$. Besides that, it follows from the characterization that additions on languages and transformations, $[\text{ADD}_L]$ and $[\text{ADD}_X]$, are the only two operators in the Banana Algebra that can produce new, indirectly constructed ambiguities that were not introduced directly as constants. None of the other operators in the Banana Algebra are capable of producing new (vertical nor horizontal) ambiguities. Thus, when analyzing ambiguity of a transformation, X, we need to test only its constituent constants and addition for ambiguities. The Banana Algebra Tool features an implementation of the above ambiguity analysis [6].

6.2. Abstract vs. Concrete Syntax

A key issue in building the tool was the choice of whether to work with abstract or concrete syntax. Everything we have presented so far has been working exclusively on the abstract syntactic level. For practical usability of the tool, however, it turns out to be more convenient to work with concrete syntax.

Figure 5 illustrates the difference between using abstract and concrete syntax for specifying transformations. Figure 5(a) depicts a fragment of a grammar for a subset of Java that deals with associativity and precedence of expressions by factorizing operators into several distinct levels according to operator precedence (as commonly found in programming language grammars); in this case, there are nine levels from `Exp` and `Exp1` all the way to `Exp8`.

Now suppose we were to extend the syntax of Java by adding a new statement, `repeat-until`, with syntax: `"repeat" Stm "until" "(" Exp ")" ";"`. Such a construction can easily be transformed into core Java by desugaring it into a `do-while` with a negated condition. Figure 5(b) shows how this would be done at the abstract syntactic level, using a representation of Java's abstract syntax tree. Transformation arguments are marked with dollar signs; e.g., `$1` and `$2` (as explained later). Since negation is found at the eighth precedence level (in `Exp7`), the AST fragment for specifying the negated conditional expression would have to take us from `Exp` all the way to `Exp7`, add the negation "`Exp7.neg(...)`", before adding the parentheses "`Exp8.par(...)`" and the

second argument, "`$2`" (which contains the original expression that was to be negated). The relevant productions have been underlined, both in the abstract and the concrete syntaxes. Figure 5(c) specifies the same transformation, but at the concrete syntactic level, using strings instead of ASTs. When working with concrete syntax, there is no need for dealing explicitly with such low-level considerations which are more appropriately dealt with by the parser.

Note that since we test for ambiguity, we can move freely between abstract and concrete syntax. We thus allow transformations to be specified in either concrete syntax (Figure 5(c)) or abstract syntax (Figure 5(b)).

6.3. Hygiene

A well-known problem in macro systems is that of *variable capture*, where new variable bindings introduced by the macro expansion risk coming in conflict with the names used in code being expanded. Some macro systems, such as that used in Scheme [8], implement *hygiene* [9, 10], which ensure that macro-generated variable names cannot conflict with each other or with user-defined variable names.

As macro hygiene techniques are fundamentally connected to the identifier binding rules of the language being transformed, the Banana Algebra Tool does not presently implement any such technique. Authors of transformations must ensure that their results do not capture variables. However, there is nothing fundamental that prevents extension of the Banana Algebra Tool to use any of the standard macro hygiene techniques. Users would simply annotate productions in the grammar as being producers of identifiers, and a standard hygiene technique can be used to ensure the absence of variable capture. Implementers of these techniques must be careful, however, to ensure that addition of languages preserves the hygiene annotations.

6.4. Parsing and Unparsing

Given the nature of the algebra, it is important that the parsing algorithm used is capable of parsing any context-free grammar and that it is closed under union. If the algorithm could only parse some context-free grammars, then we would risk producing grammars that could not be parsed as intermediate results. Our implementation uses an eager variant of Earley's algorithm, which can parse any context-free grammar in worst-case cubic time. Another good choice of algorithm would be generalized LL (GLL) parsing [11], which can generate straightforward, top-down parsing code for any CFG.

We unparse canonically by printing all lexical entities without spaces and by printing a single space between them (provided that is allowed as legal white-space in the language). Additionally, our tool allows the use of special pretty-printing directives to control the display of output. These directives do not approach the level of sophistication and control of unparsing of, for instance, the Generic Pretty Printer, `GPP` [12].

6.5. Debugging and Error Reporting

We have not addressed debugging and error reporting in the tool. This is a fundamental challenge for the entire area of syntactic language transformation and a prerequisite for language extension to be seamlessly integrated in program development, in practice.

It would require maintaining a coherent relationship between input and output terms at runtime (especially between character positions and line numbers in source and target languages of individual transformations). This information should then be used to trace error messages in target language programs back to the original source language input and thereby provide meaningful error messages to a programmer who would ideally be oblivious to the transformations.

We expect that origin tracking, per van Deursen [13], can be incorporated into the tool. Thus, we have not considered these aspects critical for the validation of our algebraic approach.

7. Examples

We have experimentally validated the algebraic approach to language extension by trying out the tool and implementation on various concrete examples. After a concrete example program, we will demonstrate language extensions from each of the "four scenarios" discussed in the introduction.

We will now revisit the example of extending the λ-Calculus with numerals that we have previously seen as a *catamorphism* (in Section 2.1) and later (in Figure 1) as a *general extension pattern*, motivating the algebraic approach.

Figure 6(a) shows the λ-Calculus as a Banana Algebra language constant (with standard whitespace, as defined by: "`$ = [ \n\t\r]*`"). Figure 6(b) defines the transformation from the λ-Calculus extended with numerals to the core calculus (cf., Figure 1). First, the contents of the file "`lambda.l`" (which we assume to contain the constant in Figure 6(a)) is loaded and bound to the Banana Algebra variable, *l*. Then, in that program, *ln* is bound to the language containing the extension (assumed to reside in the file "`lambda-num.l`"). Afterwards, *ln2l* is bound to the constant transformation that transforms the numeral extension to the core λ-Calculus. Finally, that constant transformation is added to the identity transformation on the λ-Calculus, `idx(l)`.

Similarly, The Banana Algebra Tool can be used to extend Java with lots of syntactic constructions which can be desugared into Java itself; e.g., dedicated literal syntax for Map structures, a null-check operator similar to C#'s `??` operator, or design pattern templates. Here, we will give only one simple example of a Java extension; the `repeat-until` of Figure 5(c). Although the Java grammar is big ("`java.l`" is a standard 575-line context-free grammar for Java), extending Java safely with a fully operational `repeat-until` construction requires only seven lines, with no configuration files or other input required:

```
let java = "java.l"
in let repeat  =  { Stm.repeat  :  "repeat" Stm "until" "(" Exp ")" ";" ; }
in letx repeat2java =
   (| repeat -> java [Stm -> Stm, Exp -> Exp]
      Stm.repeat  =  'do $1 while (!($2));'  ;
```

```
{
   $        =  [ \n\t\r]*       ;
   Id       =  [a-z]+           ;

   exp.var  :  Id               ;
   exp.lam  :  "\\" Id "." exp  ;
   exp.app  :  "(" exp exp ")"  ;
}
```

(a) Language: λ-Calculus (with standard whitespace definition: "`[ \n\t\r]*`").

```
let l = "lambda.l"
in let ln = "lambda-num.l"
in letx ln2l =
    (| ln -> l [exp -> exp]

       exp.zero  =  '\z.z'        ;
       exp.succ  =  '\s.$1'       ;
       exp.pred  =  '($1 \z.z)'   ;
    |)
in
   ln2l + idx(l)
```

(b) Transformation: λ-Calculus extended with numerals to core λ-Calculus (cf. Fig 1).

Figure 6: Banana Algebra example programs: a language and a transformation.

```
   |)
in
   repeat2java + idx(java)
```

More ambitiously, The Banana Algebra Tool may be used to embed entire DSLs into a host language. We have used the tool to embed standard `SQL` constructions into the <bigwig> [14] language; e.g., the ubiquitous "`select-from-where`" could be added by:

```
let bigwig = "bigwig.l"
in let select  =  { stm.select  :  "select" id_list "from" exp "where" exp ";" ; }
in letx select2bigwig =
   (| select -> bigwig [stm -> stm, exp -> exp, id_list -> id_list]
      stm.select  =  'factor($2) { if ($3) { return # \+ ($1); } }'  ;
   |)
in
   select2bigwig + idx(bigwig)
```

Once defined, languages and transformations can all be added, composed, or otherwise put together. Thus, a programmer can use the tool to essentially tailor his own macro-extended language; e.g. take `Java`, lift it to an identity transformation from `Java` to `Java`, remove loop constructions, add `SQL` constructions, and maybe give all keywords French names as in:

```
english2french o ((idx(java) \ loops) + sql2java)
```

Relying on the existence of the tool, we have used the tool on itself to add more operators to the algebra. We can easily extend the Banana Algebra with an *overwrite* operator "`<<`" on transformations (defined in terms of the core algebra):

$$[\![X_1 \texttt{ << } X_2]\!]_X \quad = \quad (X_1 \texttt{ \ src(}X_2\texttt{)) + } X_2$$

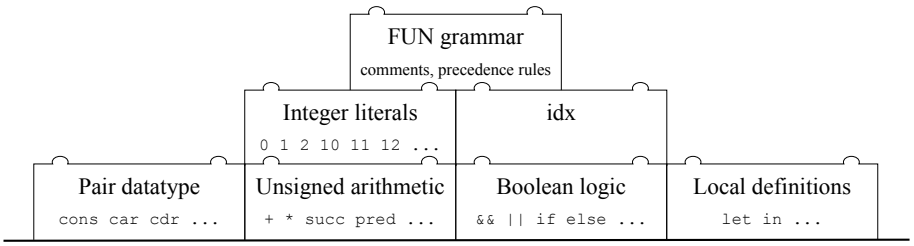

Figure 7: The FUN language built on top of the λ-calculus

8. A Comprehensive Example: "FUN"

To put the algebraic and incremental development approach to the test, we have built a functional language "FUN" (inspired by a language used in an undergraduate course to introduce functional programming at Aarhus University and Aalborg University in Denmark). The syntax of "FUN" is described in Appendix D.

Starting from the core λ-Calculus a number of features are constructed and combined in an iterative process to form increasingly powerful versions of the FUN language. Figure 7 shows the first version built from the following general and independent transformations:

- Unsigned integer arithmetic offering addition and multiplication, using Church numeral representation, but otherwise similar to the transformation presented in Section 2.1.
- Boolean logic operators including the `if`–`else` construction. This is very similar to the transformation found in Section 2.2.
- A pair datatype including the `cons`, `car`, and `cdr` operators, familiar from the Lisp family of languages.
- Variables, local binding (`let`–`in`), function definition (optionally recursive) and application.
- Decimal integer literals encoded on top of the integer arithmetic block using `Succ` to encode digits and the + and `*` operators to encode digit positions.
- The FUN language grammar featuring infix operator precedence, proper whitespacing and (multi-line) comments.

Without changing the existing transformations, three new features were added (one at a time) to the FUN language:

1. Signed integers and the subtraction operator, "−", built on top of the unsigned arithmetic, boolean logic, and pair datatype (integers are encoded as pairs of a boolean sign and an unsigned integer).

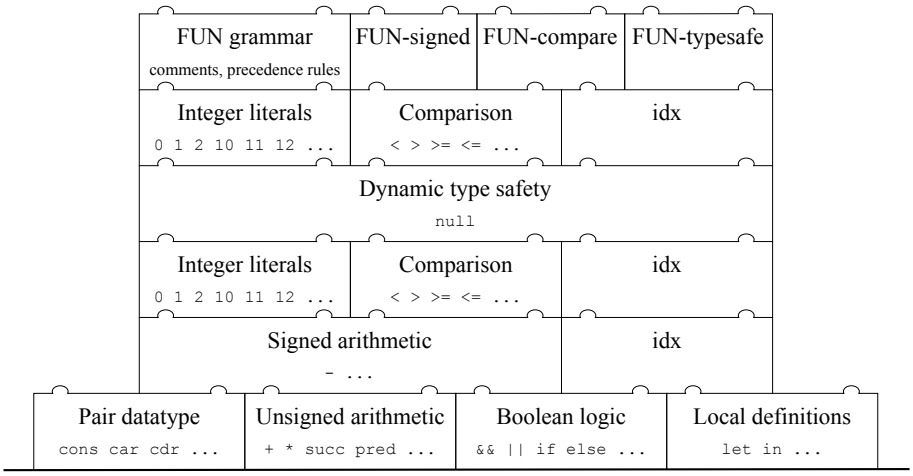

Figure 8: The FUN-typesafe language built in multiple layers on top of the λ-calculus

2. Integer comparison operators ("$<$", "$>$", "$\leq$", etc.) built on top of the signed integer arithmetic using "$-$" and sign checking.
3. Dynamic type safety. All values are encoded as "(*type*,*value*)" pairs and all operators are modified to dynamically test the types of their arguments. A special "`null`" value is returned in case of type errors, and may be used to catch exceptions.

Figure 8 illustrates the final result (FUN-typesafe) after all three extensions were added. Notice how the components from Figure 7 have all been reused; in particular, the decimal integer literal transformation is now placed on top of the signed integer arithmetic, producing signed values. Furthermore, as the dynamic type safety layer has to redefine all operators and value constructors in order to add type annotation and testing, the source language of this layer is kept absolutely minimal to reduce its size and complexity. Hence, the integer literals and comparison operators are re-introduced above this layer in a type-safe version, and therefore these transformations are used twice in different contexts.

For the FUN-typesafe language specification, 245 algebraic operators were used, as tabulated in Figure 9. The entire transformation reduces to a constant (constructive catamorphism) transformation file of size 4MB. This example is very instructive and could be used for teaching functional programming languages. However, due to the size of the right-hand sides of the final transformation, even small FUN-typesafe programs will be transformed to rather voluminous lambda expressions.

count	abbrv.	operator	syntax
58	$[\texttt{CON}_L]$	*constant languages*	l
51		*language inclusions*	"<*filename*>.l"
28	$[\texttt{ADD}_L]$	*language additions*	L + L
23	$[\texttt{VAR}_L]$	*language variables*	v
17	$[\texttt{CON}_X]$	*constant transformations*	$(\!\vert L \to L\ [\tau]\ c \vert\!)$
17	$[\texttt{ADD}_X]$	*transformation additions*	X + X
14		*transformation inclusions*	"<*filename*>.x"
9	$[\texttt{IDX}_X]$	*identity transformations*	idx(L)
8	$[\texttt{LET}_X]$	*local definitions*	let v=L in X
8	$[\texttt{SEQ}_X]$	*compositions*	X ∘ X
4	$[\texttt{RES}_L]$	*language restrictions*	L \ L
4	$[\texttt{VAR}_X]$	*transformation variables*	w
2	$[\texttt{LTX}_X]$	*local definitions*	letx w=X in X
2	$[\texttt{SRC}_L]$	*source extractions*	src(X)
245	Σ	in total	

Figure 9: Operator usage statistics for the FUN example.

9. Related Work

Our work shares many commonalities and goals with that of *syntax macros*, *source transformation systems*, and *catamorphisms* (from a category theory perspective) the relation to which will be outlined below.

Syntax macros [1, 15] provide a means to unidirectionally extend a "host language" on top of which the macro system is hard-wired. Extension by syntactic macros corresponds to having control over only "step iii)" of Figure 1 (some systems also permit limited control over what corresponds to "step ii)"). By contrast, our algebraic approach can be used to extend the syntax of *any* language or transformation; and not just in one direction—extensions may be achieved through addition, composition, or otherwise modular assembly of other previously defined languages or transformations. Uni-directional extension is just one form of incremental definition in our algebraic approach.

Perhaps the best-known example of syntax macros is the macro language found in Scheme [8]. These differ from Banana Algebra in a number of significant ways. Scheme's macros provide none of the static safety nor performance guarantees that are achieved through using language-typed constructive catamorphisms. On the other hand, Scheme's macros are capable of expressing a broader range of transformations. Perhaps the most important distinction is that Scheme macros must operate on the level of the abstract syntax of Scheme, while the Banana Algebra is capable of working with concrete syntax for arbitrary unambiguous context-free languages.

The work on *extensible syntax* [16] improves on the definition flexibility in providing a way of defining grammars incrementally. However, it supports only three general language operations: extension, restriction, and update.

Compiler generator tools, such as Eli [17], Elan [18], Stratego/XT [19], ASF+SDF [20], TXL [21], JastAdd [22], and Silver [23] may all be used for source-to-target language transformation. They all have wider ambitions than our work, supporting specifications of full-scale compilers, many including static and dynamic semantics as well as Turing Complete computation on ASTs of the source language which obviously precludes our level of safety guarantees.

Although many of the tools support modular language development, none of them provide an algebra on top of their languages and transformations.

Systems based on *attribute grammars* (e.g., Eli, JastAdd, and Silver) may be used to indirectly express source-to-target transformations. This can be achieved through Turing Complete computation on the AST of the source language which compute terms of the target language in a downward or upward fashion (through synthesized and inherited attributes), or combinations thereof. In contrast, catamorphisms are restricted to upward inductive recombination of target ASTs. Our transformations could easily be generalized to also construct target AST downwards, by simply allowing catamorphisms to take target typed *AST arguments* (as detailed in [2], p. 17). This corresponds to a notion of *anamorphisms* and *hylomorphisms*, but would compromise compile-time elimination of composition (since anamorphisms and catamorphisms in general cannot be fusioned into one transformation, without an intermediate step).

Systems based on *term rewriting* (e.g., Elan, TXL, ASF+SDF, and Stratego/XT) may also be used to indirectly express source-to-target transformations. However, a transformation from language S to T has to be encoded as a rewriting working on terms of combined type: $S \cup T$ or $S \times T$. Although the tools may syntactically check that each rewriting step respects the grammars, the formalism comes with three kinds of termination problems which cannot be statically verified in either of the tools; a transformation may: i) never terminate; ii) terminate too soon (with unprocessed source terms); and, iii) be capable of producing a forest of output ASTs which means that it is the responsibility of the programmer to ensure that the end result is one single output term. To help the programmer achieve this, rewriting systems usually offer control over the rewriting strategies.

In order to issue strong safety guarantees, in particular termination, we clearly sacrifice expressibility in that the catamorphisms are *not* able to perform Turing Complete transformations. However, previous work using constructive catamorphisms for syntactic transformations [1, 2, 3] indicate that they are sufficiently expressive and useful for a wide range of applications. Our contribution is a technique for using constructive catamorphisms in a modular and incremental fashion.

Of course, catamorphisms may be mimicked by a disciplined style of functional programming, possibly aided by traversal functions automatically synthesized from datatypes [24], or by libraries of combinators [25]. However, as these programs still occur within a general purpose context, such disciplined programming cannot provide our level of safety guarantees and would not be able to compile-time factorize composition (although the functional techniques *deforestation/fusion* [26, 27, 28] may—in some instances—be used to achieve

similar effects in a general purpose context).

There exists a body of work on catamorphisms in a category theoretical setting [29, 4]. However, these are theoretical frameworks that have not been turned into practical tool implementations supporting the notion of addition on languages and transformations which plays a crucial role in the extension pattern of Figure 1 and many of the examples.

10. Conclusion

The algebraic approach offers via 20 operators a *simple*, *incremental*, and *modular* means for specifying syntactic language extensions through algebraic composition of previously defined languages and transformations. The algebra comes "for free" in that any algebraic transformation term can be statically *reduced* to a constant transformation without compromising the strong *safety* and *efficiency* properties offered by catamorphisms.

The tool may be used by: 1) programmers to extend existing languages with their own macros; 2) developers to embed DSLs in host languages; 3) compiler writers to implement only a small core language (and specify the rest externally as extensions); and 4) developers and teachers to build multi-layered languages. The Banana Algebra Tool is available—as 3,600 lines of O'Caml code—along with examples are available on its homepage:

`[ http://www.itu.dk/people/brabrand/banana-algebra/ ]`

Acknowledgments

The authors would like to acknowledge Kevin Millikin, Mads Sig Ager, Per Graa, Kristian Støvring, Anders Møller, Michael Schwartzbach, Martin Sulzmann, Sharjeel Imam, and the anonymous referees of LDTA 2009 and *Science of Computer Programming* for useful comments and suggestions.

[1] C. Brabrand, M. I. Schwartzbach, Growing languages with metamorphic syntax macros, in: Proc. ACM SIGPLAN Workshop on Partial Evaluation and semantics-based Program Manipulation, PEPM'02, ACM, 2002.

[2] C. Brabrand, M. I. Schwartzbach, The metafront system: Safe and extensible parsing and transformation, Science of Computer Programming Journal (SCP) 68 (1) (2007) 2–20. doi:http://dx.doi.org/10.1016/j.scico.2005.06.007.

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

[4] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: J. Hughes (Ed.), Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991, Vol. 523, Springer-Verlag, Berlin, 1991, pp. 124–144.

[5] J. Andersen, C. Brabrand, Syntactic language extension via an algebra of languages and transformations, ITU Technical Report. Available from: `http://www.itu.dk/people/brabrand/banana-algebra/` (2008).

[6] C. Brabrand, R. Giegerich, A. Møller, Analyzing ambiguity of context-free grammars, Science of Computer Programming 75 (3) (2010) 176 – 191. doi:10.1016/j.scico.2009.11.002.

[7] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[8] M. Sperber, M. Flatt, A. V. Straaten, R. K. Dybvig, R. B. Findler, J. Matthews, Revised^6 report on the algorithmic language Scheme, Journal of Functional Programming 19 (S1) (2009) 1–301.

[9] E. Kohlbecker, D. P. Friedman, M. Felleise, B. Duba, Hygienic macro expansion, in: Proceedings of the 1986 ACM conference on LISP and functional programming, LFP '86, ACM, New York, NY, USA, 1986, pp. 151–161.

[10] A. Bawden, J. Rees, Syntactic closures, in: Proceedings of the 1988 ACM conference on LISP and functional programming, LFP '88, ACM, New York, NY, USA, 1988, pp. 86–95. doi:10.1145/62678.62687.

[11] E. Scott, A. Johnstone, GLL Parsing, Electronic Notes in Theoretical Computer Science 253 (7) (2010) 177–189.

[12] M. de Jonge, Pretty-printing for software reengineering, in: Proceedings: International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society Press, 2002, pp. 550–559.

[13] A. van Deursen, Origin tracking in primitive recursive schemes, Tech. Rep. CS-R9401, Centrum voor Wiskunde en Informatica, Amsterdam (1994).

[14] C. Brabrand, A. Møller, M. I. Schwartzbach, The `<bigwig>` project, ACM Transactions on Internet Technology 2 (2) (2002) 79–114.

[15] D. Weise, R. F. Crew, Programmable syntax macros, in: Programming Language Design and Implementation (PLDI), 1993, pp. 156–165.

[16] L. Cardelli, F. Matthes, M. Abadi, Extensible syntax with lexical scoping, sRC Research Report 121 (1994).

[17] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, W. M. Waite, Eli: a complete, flexible compiler construction system, Communications of the ACM 35 (2) (1992) 121–130.

[18] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, C. Ringeissen, An overview of ELAN, in: Second Intl. Workshop on Rewriting Logic and its Applications, Vol. 15, 1998.

[19] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/xt 0.17. a language and toolset for program transformation, Science of Computer Programming 72 (1-2) (2008) 52–70.

[20] M. G. J. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: a component-based language development environment, in: Proc. Compiler Construction 2001, Springer-Verlag, 2001.

[21] J. Cordy, Txl - a language for programming language tools and applications, in: Proceedings of ACM 4th International Workshop on Language Descriptions, Tools and Applications (LDTA'04), 2004, pp. 1–27.

[22] G. Hedin, E. Magnusson, JastAdd - a Java-based system for implementing frontends, in: Electronic Notes in Theoretical Computer Science, Vol. 44(2), Elsevier Science Publishers, 2001.

[23] E. V. Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Electronic Notes in Theoretical Computer Science 203 (2) (2008) 103–116.

[24] R. Lämmel, J. Visser, J. Kort, Dealing with Large Bananas, in: J. Jeuring (Ed.), Proceedings of WGP'2000, Technical Report, Universiteit Utrecht, 2000, pp. 46–59.

[25] S. D. Swierstra, P. R. A. Alcocer, J. Saraiva, Designing and implementing combinator languages, in: Third Summer School on Advanced Functional Programming, volume 1608 of LNCS, Springer-Verlag, 1999, pp. 150–206.

[26] P. Wadler, Deforestation: Transforming programs to eliminate trees, Theoretical Computer Science 73 (1990) 344–358.

[27] J. P. Fernandes, A. Pardo, J. Saraiva, A shortcut fusion rule for circular program calculation, in: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, ACM, 2007, pp. 95–106.

[28] J. Voigtländer, Semantics and pragmatics of new shortcut fusion rules, in: J. Garrigue, M. Hermenegildo (Eds.), Proc. Functional and Logic Programming, Vol. 4989 of LNCS, Springer-Verlag, 2008, pp. 163–179.

[29] R. B. Kieburtz, J. Lewis, Programming with algebras, in: Advanced Functional Programming, number 925 in Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 267–307.

Appendix A. Semantics of Languages

Appendix A.1. Language operators

In order to be able to define algebraic operations on grammars later on, we need the operators defined in the following definitions. We use the term "grammar" to refer to a production-named CFG, as defined in Section 5.

Definition 5 (The nil grammar l_0)**.** *The nil grammar is defined as:*

$$l_0 = (\emptyset, \emptyset, \emptyset, \pi_0)$$

where π_0 is the empty production function with $dom(\pi_0) = \emptyset$.

Proposition 1 (The nil grammar is well-formed)**.** *The nil grammar is well-formed: $\vdash_{\texttt{wfl}} l_0$.*

Proof. Straightforward from Definition 4 and Definition 5. □

Definition 6 (Addition compatibility of grammars $\sim_l$)**.** *Two grammars, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $l' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ are said to be "addition compatible" (written $l \sim_l l'$) iff:*

$$\begin{array}{l} \forall n \in \mathcal{N} \cap \mathcal{N}' : \\ \pi(n) \neq \emptyset \quad \wedge \quad \pi'(n) \neq \emptyset \\ \Downarrow \\ \forall (p, \alpha) \in \pi(n),\ (p', \alpha') \in \pi'(n)\ :\ p = p' \Rightarrow \alpha = \alpha' \end{array}$$

This definition expresses that the production rules of two addition compatible languages are either disjoint or identical, i.e. no conflicts. As a special case, if a nonterminal is undefined but used in production rules in one language (e.g. $\pi(n) = \emptyset$) then this language is still addition compatible with another language in which n is actually defined (with either one or more named production rules or a regular expression). This is conceptually similar to the linking process used when building computer software: a source code file may use procedures defined in other source code files or libraries, and the linking process will bind the using instance to the defining instance.

Definition 7 (Grammar addition $\oplus_l$)**.** *Binary addition on grammars $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $l' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ is defined when $l \sim_l l'$ as:*

$$l \oplus_l l' \ = \ (\mathcal{N} \cup \mathcal{N}', \Sigma \cup \Sigma', \mathcal{P} \cup \mathcal{P}', \pi \oplus_\pi \pi')$$

where $\pi \oplus_\pi \pi' : \mathcal{N} \cup \mathcal{N}' \to 2^{(\mathcal{P} \cup \mathcal{P}') \times ((\mathcal{N} \cup \mathcal{N}') \cup (\Sigma \cup \Sigma'))^*}$ is defined as:

$$(\pi \oplus_\pi \pi')(n) \ = \ \begin{cases} \pi(n) & n \notin \mathcal{N}', \\ \pi'(n) & n \notin \mathcal{N}, \\ \pi(n) \cup \pi'(n) & \text{otherwise} \end{cases}$$

Note that when adding two grammars, it may be the case that one of the grammars refers to a nonterminal which is not defined in this grammar, but *is* defined in the other grammar.

Proposition 2 (Grammar addition preserves well-formedness)**.** *When adding together two well-formed grammars, the resulting grammar will be well-formed:*

$$\forall l, l' \in \mathit{CFG} \ : \ (l \sim_l l' \ \wedge \ \vdash_{\texttt{wfl}} l \ \wedge \ \vdash_{\texttt{wfl}} l') \ \Rightarrow \vdash_{\texttt{wfl}} (l \oplus_l l')$$

Proof. For part (i) of the well-formedness statement, since both l and l' are well-formed, the only thing we need to check is that when $(p, \alpha) \in \pi(n)$ and $(p', \alpha') \in \pi'(n)$ the statement $p = p' \Rightarrow \alpha = \alpha'$ still holds. But since l and l' are addition compatible, and the top part (above the '$\Downarrow$') of definition 6 is true, we have the bottom part, finishing this part of the proof.

Part (ii) of the well-formedness statement states that a nonterminal must be either defined or used in $(l \oplus_l l')$. But since any nonterminal from $(l \oplus_l l')$ originates from either l or l' and these grammars are both well-formed, the nonterminal must be either defined or used in either l or l'. And since all grammar productions from these two grammars are found in $(l \oplus_l l')$, we have this part of the proof as well. □

Definition 8 (Grammar subtraction $\ominus_l$)**.** *Binary subtraction on grammars* $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ *and* $l' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ *is defined when* $l \sim_l l'$ *as:*

$$l \ominus_l l' \ = \ (\widehat{\mathcal{N}}, \Sigma, \mathcal{P}, \pi \ominus_\pi \pi')$$

where $\pi \ominus_\pi \pi' : \mathcal{N} \to 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*}$ *is defined as:*

$$(\pi \ominus_\pi \pi')(n) \ = \ \begin{cases} \pi(n) & n \notin \mathcal{N}', \\ \pi(n) \setminus \pi'(n) & \text{otherwise} \end{cases}$$

and:

$$\begin{aligned} \widehat{\mathcal{N}} \ = \ & \{n \in \mathcal{N} \mid (\pi \ominus_\pi \pi')(n) \neq \emptyset\} \cup \\ & \{n \in \mathcal{N} \mid \exists n' \in \mathcal{N}; p \in \mathcal{P}; \alpha, \alpha' \in (\mathcal{N} \cup \Sigma)^* : (\pi \ominus_\pi \pi')(n') = (p, [\alpha n \alpha'])\} \end{aligned}$$

The purpose of grammar subtraction $l \ominus_l l'$ is to remove a subset of the productions in a grammar l – more precisely the productions found in both l and l'. So fix a nonterminal $n \in \mathcal{N} \cap \mathcal{N}'$. The definition now states that $(\pi \ominus_\pi \pi')(n) \ = \ \pi(n) \setminus \pi'(n)$. Suppose $\pi(n) \neq \emptyset$ and $\pi'(n) \neq \emptyset$ since $\pi(n) = \pi(n) \setminus \pi'(n)$ otherwise. As $l \sim_l l'$, we have that they are both sets of named productions with no conflicts, resulting in the removal of the common subset of productions.

Proposition 3 (Grammar subtraction preserves well-formedness)**.** *When subtracting any grammar from a well-formed grammar, the resulting grammar will be well-formed:*

$$\forall l, l' \in \mathit{CFG} \ : \ (l \sim_l l' \ \wedge \ \vdash_{\texttt{wfl}} l) \ \Rightarrow \vdash_{\texttt{wfl}} (l \ominus_l l')$$

Proof. Part (i) of the well-formedness statement follows directly from the fact that G is well-formed, as we are not adding any new productions to the grammar. Part (ii) follows directly from the definition of $\widehat{\mathcal{N}}$ in Definition 8. □

Definition 9 (Grammar renaming compatibility $\sim_r$)**.** *A renaming, R (as defined in Figure 2(c)) is compatible with a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in CFG$, written $R \sim_r l$, iff:*

$$\begin{array}{ll} m \notin \mathcal{N} & \text{if } R = m \,/\, n \\ \nexists \alpha : \pi(n) = (q, \alpha) & \text{if } R = q \,/\, n \,.\, p \end{array}$$

Definition 10 (Grammar nonterminal renaming $[\,]_{\text{nl}}$)**.** *Renaming a nonterminal n to m in a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in CFG$ (written "$l\ [m \,/\, n]$") is defined when $m \,/\, n \sim_r l$ (i.e. $m \notin \mathcal{N}$) as:*

$$l\ [m \,/\, n] \ = \ \begin{cases} (\mathcal{N}', \Sigma, \mathcal{P}, \pi') & n \in \mathcal{N} \\ l & \text{otherwise} \end{cases}$$

with $\mathcal{N}' = (\mathcal{N} \setminus \{n\}) \cup \{m\}$ and $\pi' : \mathcal{N}' \to 2^{\mathcal{P} \times (\mathcal{N}' \cup \Sigma)^*}$ is defined as:

$$\pi'(n') \ = \ \begin{cases} \pi(\hat{n}) & \pi(\hat{n}) \in 2^{\mathcal{P} \times (\mathcal{N}' \cup \Sigma)^*} \\ \{(p, \underset{n \to m}{\text{seq-subst}}(\alpha)) \mid (p, \alpha) \in \pi(\hat{n})\} & \text{otherwise} \end{cases}$$

where:

$$\hat{n} = \begin{cases} n & n' = m \\ n' & \text{otherwise} \end{cases}$$

and the $\underset{n \to m}{\text{seq-subst}}$ *function, substituting all n occurences by m in a sequence is defined recursively as:*

$$\underset{n \to m}{\text{seq-subst}}(\alpha) = \begin{cases} \epsilon & \alpha = \epsilon \quad \textit{(where } \epsilon \textit{ is the empty sequence)} \\ m \underset{n \to m}{\text{seq-subst}}(\alpha') & \alpha = n\alpha' \\ x \underset{n \to m}{\text{seq-subst}}(\alpha') & \alpha = x\alpha' \wedge x \neq n \end{cases}$$

Proposition 4 (Grammar nonterminal renaming preserves well-formedness)**.** *Renaming a nonterminal of a well-formed grammar produces a grammar which is well-formed:*

$$\forall l \in CFG, \forall n, m : \ m \,/\, n \sim_r l \wedge \ \vdash_{\texttt{wfl}} l \ \Rightarrow \vdash_{\texttt{wfl}} (l\ [m \,/\, n])$$

Proof. Left to the reader. □

Definition 11 (Grammar production name renaming $[]_{\text{pl}}$)**.** *Renaming a production name $n.p$ to $n.q$ in a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in CFG$ (written "$l\ [q/n.p]$") is defined when $q/n.p \sim_r l$ (i.e. $\nexists \alpha : \pi(n) = (q, \alpha)$) as:*

$$l\ [q/n.p] \ = \ (\mathcal{N}, \Sigma, \mathcal{P}', \pi')$$

with $\mathcal{P}' = \mathcal{P} \cup \{q\}$ and $\pi' : \mathcal{N} \to 2^{\mathcal{P}' \times (\mathcal{N} \cup \Sigma)^*}$ is defined as:

$$\pi'(n') \ = \ \begin{cases} \{(\hat{p}, \alpha) \mid (p', \alpha) \in \pi(n)\} & n' = n \\ \pi(n') & \text{otherwise} \end{cases}$$

where:

$$\hat{p} = \begin{cases} q & p' = p \\ p' & \text{otherwise} \end{cases}$$

Proposition 5 (Grammar production name renaming preserves well-formedness)**.** *Renaming a production name of a well-formed grammar produces a grammar which is well-formed:*

$$\forall l \in CFG, \forall n, p, q : \ q/n.p \sim_r l \wedge \vdash_{\texttt{wfl}} l \ \Rightarrow \vdash_{\texttt{wfl}} (l\ [q/n.p])$$

Proof. Left to the reader. □

Definition 12 (Grammar inclusion $\sqsubseteq_l$)**.** *Grammar inclusion is a binary relation on grammars $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $l' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ defined as:*

$$l \sqsubseteq_l l' \ \equiv \ \forall n \in \mathcal{N} : \pi(n) = \emptyset \ \vee \ (n \in \mathcal{N}' \ \wedge \ \pi(n) \subseteq \pi'(n))$$

Note that this definition only considers the relation between the two production functions π and π'. We do not care whether $\mathcal{N} \subseteq \mathcal{N}'$, $\mathcal{P} \subseteq \mathcal{P}'$, and $\Sigma \subseteq \Sigma'$ or not.

Proposition 6 (Grammar inclusion and well-formedness)**.** *Let $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in CFG$ and $l' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi') \in CFG$ then:*

$$\vdash_{\texttt{wfl}} l \ \wedge \ l \sqsubseteq_l l' \ \Rightarrow \ \mathcal{N} \subseteq \mathcal{N}'$$

Proof. Let $n \in \mathcal{N}$. Since l is well-formed, this means that n is either defined or used in π, i.e. there exists a concrete production that involves n. In other words $\exists n' \in \mathcal{N} : \pi(n') \neq \emptyset$ such that either $n = n'$ or n is used in $\pi(n')$. But since $\pi(n') \neq \emptyset$ we have that $n' \in \mathcal{N}' \ \wedge \ \pi(n') \subseteq \pi'(n')$ and therefore $n \in \mathcal{N}'$. □

Appendix B. Semantics of Transformations

Appendix B.1. Transformations defined

We now proceed to formally define well-formedness of transformations, which is informally defined in Section 5.3. This definition is dependent on a number of other relations, as follows:

Definition 13 ("Sequence picker" operator $|_P$)**.** *For any sets, P and Q, we define the "sequence picker" operator $|_P : Q^* \times \mathbb{N} \hookrightarrow P$ using the following inference system:*

$$\frac{}{qs|^1_P = q} \quad q \in P \qquad \frac{s|^{(i-1)}_P = p}{qs|^i_P = p} \quad q \in P,\ i > 1$$

$$\frac{s|^i_P = p}{qs|^i_P = p} \quad q \notin P$$

For a sequence $s \in Q^*$ and a number $i \in \mathbb{N}$, $s|^i_P$ yields the ith element of P found in the sequence s, and it is undefined if such an element does not exist. Of course, if $P \cap Q = \emptyset$, $s|^i_P$ cannot be defined for any i.

Definition 14 ("Sequence counter" operator $|\cdot|_P$)**.** *For any sets, P and Q, we define $|\cdot|_P : Q^* \to \mathbb{N}$ as the number of P elements found in the sequence, by the following inference system:*

$$\frac{}{|[]|_P = 0} \qquad \frac{|s|_P = i}{|qs|_P = i + 1} \quad q \in P$$

$$\frac{|s|_P = i}{|qs|_P = i} \quad q \notin P$$

I.e. $|s|_P$ is the maximal value for i, where $s|^i_P$ is defined, and if $|s|_P = 0$, then $s|^i_P$ is not defined for any i.

Definition 15 (Reconstructor typing $\vdash_l$)**.** *Given a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in$ CFG and a "gap type function" $\rho : \mathbb{N} \hookrightarrow \mathcal{N}$, we say that a reconstructor $A \in \mathcal{A}_l$ has type $n \in \mathcal{N}$, written $\rho \vdash_l A : n$ iff it is provable by the following inference system:*

$$\frac{}{\rho \vdash_l i : n} \quad i \in \mathbb{N},\ \rho(i) = n$$

$$\frac{\forall j \in \{1, \ldots, m\} :\ \rho \vdash_l A_j : (\alpha|^j_{\mathcal{N}})}{\rho \vdash_l (n, p, A_1, A_2, \ldots, A_m) : n} \quad \exists \alpha :\ (p, \alpha) \in \pi(n)\ \wedge\ m = |\alpha|_{\mathcal{N}}$$

Definition 16 (Well-formed transformation $\vdash_{\texttt{wfx}}$)**.** *The transformation, $x = (l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s), l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t), \tau, c) \in$ CATA, is said to be "well-formed" (written "$\vdash_{\texttt{wfx}} x$") iff all of the following conditions are met:*

(i) $\vdash_{\texttt{wfl}} (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \quad \wedge \quad \vdash_{\texttt{wfl}} (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t)$

(ii) $\forall n \in \mathcal{N}_s\ \forall (p, \alpha) \in \pi_s(n)\ \exists A \in \mathcal{A}_{l_t} : (p, A) \in c(n)$

(iii) $$\forall n \in \mathcal{N}_s\ \forall (p, A) \in c(n)\ \exists \alpha \in (\mathcal{N}_s \cup \Sigma_s)^*\ :\ (p, \alpha) \in \pi_s(n)\ \wedge\ (i \mapsto \tau(\alpha|^i_{\mathcal{N}_s})) \vdash_{l_t} A : \tau(n)$$

(iv) $$\forall n \in \mathcal{N}_s\ \forall (p, A), (p', A') \in c(n)\ :\ p = p'\ \Rightarrow A = A'$$

(i) expresses that the source and target language grammars must be well-formed. Condition (ii) checks the completeness of the transformation: that the transformation covers all productions defined in the source language grammar (π_s), so that any valid source program can be transformed. Condition (iii) verifies that the typing of all transformation rules matches the typing defined by τ, and finally (iv) ensures that only one rule exists per production.

Appendix B.2. Applying transformations

Definition 17 (The transformation relation)**.** *Given two grammars and a reconstructor function, $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \in CFG$, $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t) \in CFG$, and $c : \mathcal{N}_s \to 2^{\mathcal{P}_s \times \mathcal{A}_{l_t}}$, we now define the "substitute relation", $\triangleright \subseteq (\mathbb{N} \hookrightarrow \mathcal{A}_{l_t}) \times \mathcal{A}_{l_t} \times \mathcal{A}_{l_t}$, written $\mu \vdash A \triangleright A'$ in the following manner:*

(i) $$\frac{\forall i \in \{1, \ldots, m\} :\ \mu \vdash A_i \triangleright A'_i}{\mu \vdash (n, p, [A_1, \ldots, A_m]) \triangleright (n, p, [A'_1, \ldots, A'_m])}$$

(ii) $$\frac{}{\mu \vdash i \triangleright \mu(i)}\quad i \in \mathbb{N},\ \mu(i)\ \textit{defined}$$

Using this relation, we can define the "transformation relation", $\longrightarrow_c : \mathcal{A}_{l_s} \times \mathcal{A}_{l_t}$, written $A \longrightarrow_c A'$ as:

(iii) $$\frac{\forall i \in \{1, \ldots, m\} :\ A_i \longrightarrow_c A'_i \qquad (i \mapsto A'_i) \vdash A' \triangleright A''}{(n, p, [A_1, \ldots, A_m]) \longrightarrow_c A''}\quad (p, A') \in c(n)$$

(iv) $$\frac{}{i \longrightarrow_c i}\quad i \in \mathbb{N}$$

Proposition 7 (The transformation relation may be viewed as a function)**.** *Let $x = (l_s, l_t, \tau, c) \in CATA$ be a well-formed transformation. The relation $\longrightarrow_c$ defined by c generates a function, $\mathrm{T}_c : \mathcal{A}_{l_s} \hookrightarrow \mathcal{A}_{l_t}$ (such that $\mathrm{T}_c(A) = A' \Leftrightarrow A \longrightarrow_c A'$), which is defined for all reconstructors $A \in \mathcal{A}_{l_s}$ that can be typed, i.e. $\exists \rho, n : \rho \vdash_{l_s} A : n$.*

Proof. First, we need to verify, that the function would be well-defined, i.e. that $A \longrightarrow_c A_1\ \wedge\ A \longrightarrow_c A_2\ \Rightarrow A_1 = A_2$ when x is well-formed. This is easily verified by structural induction first for $\triangleright$ and then for $\longrightarrow_c$, using condition (iv) of definition 16.

Now, we need to show the following:

$$\forall A \in \mathcal{A}_{l_s} :\ (\exists \rho, n : \rho \vdash_{l_s} A : n)\ \Rightarrow\ (\exists A' \in \mathcal{A}_{l_t} :\ A \longrightarrow_c A')$$

which we are going to do by structural induction in A. If $A \in \mathbb{N}$, the statement is trivial. Looking at $A = (n, p, [A_1, \ldots, A_m])$, since A can be typed, all of $A_1, \ldots, A_m$ can be typed as well, and by induction $A_i \longrightarrow_c A'_i$ are defined. Furthermore, as A can be typed, we have $(p, \alpha) \in \pi_s(n)$ for some α, and by well-formedness of x we get $(p, A') \in c(n)$ for some A', which have no gaps except in the interval $\{1, \ldots, m\}$. But since $(i \mapsto A'_i)$ is defined throughout this interval, it is easy to verify by structural induction that $(i \mapsto A'_i) \vdash A' \triangleright A''$, and so we are done. □

Definition 18 (Transformation function T_c)**.** *Let $x = (l_s, l_t, \tau, c) \in CATA$ be a well-formed transformation, and $\widehat{\mathcal{A}}_{l_s} = \{A \in \mathcal{A}_{l_s} \mid \exists \rho, n : \rho \vdash_{l_s} A : n\}$. Define the transformation function, $\mathrm{T}_c : \widehat{\mathcal{A}}_{l_s} \to \mathcal{A}_{l_t}$:*

$$\mathrm{T}_c(A) \;=\; A' \quad \Leftrightarrow \quad A \longrightarrow_c A'$$

Lemma 1 (Typing of substitute relation)**.** *The substitute relation preserves the original type when the substituted reconstructors are correctly typed:*

$$\begin{array}{l} \sigma \vdash_l A : n \;\wedge\; \forall i \in \mathrm{dom}(\sigma) : \rho \vdash_l A_i : \sigma(i) \;\wedge\; [i \mapsto A_i] \vdash A \triangleright A' \\ \Downarrow \\ \rho \vdash_l A' : n \end{array}$$

Proof. We will prove this by induction in the structure of A. If $A \in \mathbb{N}$ by definition 15 $\sigma(A) = n$ and with the rest of the preconditions and (ii) of definition 17 we are done. When $A = (n, p, [A_1, \ldots, A_m])$ from definition 15 we get $\exists \alpha : (p, \alpha) \in \pi(n) \wedge m = |\alpha|_{\mathcal{N}}$, which together with the induction hypothesis finishes the proof. □

Proposition 8 (Typing of transformed reconstructor)**.** *A well-formed transformation, $x = (l_s, l_t, \tau, c)$, where $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s)$ and $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t)$ applied to a reconstructor, $A \in \mathcal{A}_{l_s}$, preserves typing – i.e. if the type of A is n, the type of the transformed result, $\mathrm{T}_c(A)$ will be $\tau(n)$:*

$$\forall A \in \mathcal{A}_{l_s},\; n \in \mathcal{N}_s,\; \rho \in (\mathbb{N} \hookrightarrow \mathcal{N}_s) :\; \rho \vdash_{l_s} A : n \;\Rightarrow\; (\tau \circ \rho) \vdash_{l_t} \mathrm{T}_c(A) : \tau(n)$$

Proof. This is proved by structural induction in the structure of A.

$A \in \mathbb{N}$:

From the definition of the transformation, we have that $\mathrm{T}_c(A) = A$ in this case, so given n and ρ, we assume $\rho \vdash_{l_s} A : n$. Due to this assumption, there must exist an inference proof:

$$\frac{}{\rho \vdash_{l_s} A : n} \quad A \in \mathbb{N},\; \rho(A) = n$$

and we can rewrite this into:

$$\frac{}{(\tau \circ \rho) \vdash_{l_t} \mathrm{T}_c(A) : \tau(n)} \quad \mathrm{T}_c(A) \in \mathbb{N},\; (\tau \circ \rho)(\mathrm{T}_c(A)) = \tau(n)$$

which completes this part of the proof.

$A \in \mathcal{N}_s \times \mathcal{P}_s \times \mathcal{A}^*_{l_s}$:

First, we examine the special case $A = (n, p, [])$. From the definition of the reconstructor typing we have $\exists \alpha : (p, \alpha) \in \pi_s(n)$ and $|\alpha|_{\mathcal{N}_s} = 0$, and since the transformation is well-formed, definition 16 condition (ii) gives us $\exists A' \in \mathcal{A}_{l_t} : (p, A') \in c(n)$, and from condition (iii) and the well-formedness of l_s we get $(\tau \circ \alpha|^{\cdot}_{\mathcal{N}_s}) \vdash_{l_t} A' : \tau(n)$. Since $|\alpha|_{\mathcal{N}_s} = 0$, $(\tau \circ \alpha|^{\cdot}_{\mathcal{N}_s})$

is the function, which is never defined, and therefore A' cannot contain any gaps. Examining the defining rules of the transformation, T_c, we see that $\mathrm{T}_c(A) = A'$, and we are done with this case.

Turning to the case of the non-empty list, let $m \in \mathbb{N}$, $m > 0$, and let $A_i \in \mathcal{A}_{l_s}$ for all $i \in \{1, 2, \ldots, m\}$ so that:

$$\forall n \in \mathcal{N}_s,\ \rho \in (\mathbb{N} \hookrightarrow \mathcal{N}_s) :\ \rho \vdash_{l_s} A_i : n\ \Rightarrow\ (\tau \circ \rho) \vdash_{l_t} \mathrm{T}_c(A_i) : \tau(n)$$

we need to prove the following:

$$\forall n \in \mathcal{N}_s,\ \rho \in (\mathbb{N} \hookrightarrow \mathcal{N}_s) :\ \rho \vdash_{l_s} (n, p, [A_1, \ldots, A_m]) : n\ \Rightarrow \\ (\tau \circ \rho) \vdash_{l_t} \mathrm{T}_c((n, p, [A_1, \ldots, A_m])) : \tau(n)$$

Continuing the proof as in the previous case and using the induction hypothesis, we reach the following:

$$(\tau \circ \alpha|^{\cdot}_{\mathcal{N}_s}) \vdash_{l_t} A' : \tau(n)\ \wedge\ \forall i \in \{1, \cdots, m\} :\ (\tau \circ \rho) \vdash_{l_t} \mathrm{T}_c(A_i) : \tau(\alpha|^{i}_{\mathcal{N}_s})$$

Now, examining the definition of T_c and using lemma 1 we obtain the result.

□

Appendix B.3. Transformation operators

In order to be able to define algebraic operations on transformations later on, we need the operators defined in the following definitions.

Definition 19 (The nil transformation x_0)**.** *The empty transformation is defined as:*

$$x_0 = (l_0, l_0, \tau_0, c_0)$$

where l_0 is the nil grammar (cf. Appendix A, Definition 5) and where τ_0 and c_0 are the empty nonterminal typing with $dom(\tau_0) = \emptyset$ and the empty reconstructor function with $dom(c_0) = \emptyset$.

Proposition 9 (The nil transformation is well-formed)**.** *The nil transformation is well-formed: $\vdash_{\texttt{wfx}} x_0$.*

Proof. Straightforward from Definition 16 and Definition 19. □

Definition 20 (Addition compatibility of transformations $\sim_x$)**.** *Two transformations, $x = (l_s, l_t, \tau, c)$ and $x' = (l'_s, l'_t, \tau', c')$ – where $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s)$, $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t)$, $l'_s = (\mathcal{N}'_s, \Sigma'_s, \mathcal{P}'_s, \pi'_s)$, and $l'_t = (\mathcal{N}'_t, \Sigma'_t, \mathcal{P}'_t, \pi'_t)$ – are said to be "addition compatible" (written $x \sim_x x'$) iff the following conditions are met:*

(i) $$l_s \sim_l l'_s\ \wedge\ l_t \sim_l l'_t$$

(ii) $$\forall n \in \mathcal{N}_s \cap \mathcal{N}'_s\ :\ \tau(n) = \tau'(n)$$

(iii) $$\forall n \in \mathcal{N}_s \cap \mathcal{N}'_s\ \forall (p, A) \in c(n),\ (p', A') \in c'(n)\ :\ p = p'\ \Rightarrow\ A = A'$$

The source and target languages must be addition compatible, and any overlapping nonterminals – and transformation reconstructors – must be identical if the transformations are to be added.

Now, we can define addition of transformations in the following manner:

Definition 21 (Transformation addition $\oplus_x$)**.** *Binary addition on transformations,* $x = (l_s, l_t, \tau, c)$ *and* $x' = (l'_s, l'_t, \tau', c')$ *– where* $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s)$*,* $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t)$*,* $l'_s = (\mathcal{N}'_s, \Sigma'_s, \mathcal{P}'_s, \pi'_s)$*, and* $l'_t = (\mathcal{N}'_t, \Sigma'_t, \mathcal{P}'_t, \pi'_t)$ *– is defined as:*

$$x \oplus_x x' \;=\; (l_s \oplus_l l'_s, l_t \oplus_l l'_t, \tau \oplus_\tau \tau', c \oplus_c c')$$

where $\tau \oplus_\tau \tau' \;:\; \mathcal{N}_s \cup \mathcal{N}'_s \to \mathcal{N}_t \cup \mathcal{N}'_t$ *is defined as:*

$$(\tau \oplus_\tau \tau')(n) \;=\; \begin{cases} \tau(n) & n \in \mathcal{N}_s, \\ \tau'(n) & \text{otherwise} \end{cases}$$

and $c \oplus_c c' \;:\; \mathcal{N}_s \cup \mathcal{N}'_s \to 2^{(\mathcal{P}_s \cup \mathcal{P}'_s) \times \mathcal{A}_{(l_t \oplus_l l'_t)}}$ *is defined as:*

$$(c \oplus_c c')(n) \;=\; \begin{cases} c(n) & n \in \mathcal{N}_s \;\wedge\; n \notin \mathcal{N}'_s, \\ c'(n) & n \in \mathcal{N}'_s \;\wedge\; n \notin \mathcal{N}_s, \\ c(n) \cup c'(n) & \text{otherwise} \end{cases}$$

We now have the following:

Proposition 10 (Transformation addition preserves well-formedness)**.** *When adding together two well-formed transformations, the resulting transformation will be well-formed:*

$$\forall x, x' \in \mathit{CATA} \;:\; (x \sim_x x' \;\wedge\; \vdash_{\texttt{wfx}} x \;\wedge\; \vdash_{\texttt{wfx}} x') \;\Rightarrow \vdash_{\texttt{wfx}} (x \oplus_x x')$$

Proof. We need to establish the 4 conditions of definition 16. The first condition follows directly from the proposition 2 and the fact that x and x' – and hence all 4 grammars in play – are well-formed. (iv) follows from the well-formedness of the transformations as well as the addition compatibility. The proofs for condition (ii) and (iii) are very similar, so we will only show (iii), which is the more elaborate part, here:

Starting with a rewrite of condition (iii) using the symbols from the definition above:

$$\begin{aligned} &\forall n \in \mathcal{N}_s \cup \mathcal{N}'_s \;\forall (p, A) \in (c \oplus_c c')(n) \;\exists \alpha \in (\mathcal{N}_s \cup \mathcal{N}'_s \cup \Sigma_s \cup \Sigma'_s)^* \;: \\ &\quad (p, \alpha) \in (\pi_s \oplus_\pi \pi'_s)(n) \;\wedge\; ((\tau \oplus_\tau \tau') \circ \alpha|^{\cdot}_{\mathcal{N}_s \cup \mathcal{N}'_s}) \vdash_{l_t \oplus_l l'_t} A : (\tau \oplus_\tau \tau')(n) \end{aligned}$$

Picking n and (p, A), we notice that from the definition of $\oplus_c$ we have that $(p, A) \in c(n)$ or $(p, A) \in c'(n)$, and due to the symmetric nature of this argument, we can choose to say without loss of generality that $(p, A) \in c(n)$. Since $\vdash_{\texttt{wfx}} x$ we have that α exists and satisfies the left-hand side of the conjunction. As $x \sim_x x'$, we have that $(\tau \oplus_\tau \tau')(n) = \tau(n)$, so all that remains is to prove:

$$((\tau \oplus_\tau \tau') \circ \alpha|^{\cdot}_{\mathcal{N}_s \cup \mathcal{N}'_s}) \vdash_{l_t \oplus_l l'_t} A : \tau(n)$$

Again, using $\vdash_{\texttt{wfx}} x$, we have that: $(\tau \circ \alpha|_{\mathcal{N}_s}^{\cdot}) \vdash_{l_t} A : \tau(n)$; in particular there exists an inference proof of this fact, and this proof can also be used to prove the above statement. □

Transformations may be composed by the following definition:

Definition 22 (Transformation composition $\circ_x$)**.** *Composition on two transformations, $x = (l_s, l_t, \tau, c)$ and $x' = (l'_s, l'_t, \tau', c')$ – where $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s)$, $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t)$, $l'_s = (\mathcal{N}'_s, \Sigma'_s, \mathcal{P}'_s, \pi'_s)$, and $l'_t = (\mathcal{N}'_t, \Sigma'_t, \mathcal{P}'_t, \pi'_t)$ – is defined when $\vdash_{\texttt{wfl}} l_t$, $\vdash_{\texttt{wfx}} x'$, and $l_t \sqsubseteq_l l'_s$ as:*

$$x' \circ_x x \;=\; (l_s, l'_t, \tau' \circ \tau, c' \circ_c c)$$

Note that $\tau' \circ \tau$ is defined due to proposition 6, which states that $\mathcal{N}_t \subseteq \mathcal{N}'_s$. Composition of reconstructor functions ($\circ_c$) is defined by:

$$(c' \circ_c c)(n) \;=\; \{(p, \mathrm{T}_{c'}(A)) \mid (p, A) \in c(n)\}$$

And, of course we also have:

Proposition 11 (Transformation composition preserves well-formedness)**.** *When composing two well-formed transformations, the resulting transformation will be well-formed:*

$$\forall x = (l_s, l_t, \tau, c),\; x' = (l'_s, l'_t, \tau', c') \in \mathit{CATA} \;:\; (l_t \sqsubseteq_l l'_s \wedge \vdash_{\texttt{wfx}} x \wedge \vdash_{\texttt{wfx}} x') \Rightarrow \vdash_{\texttt{wfx}} (x' \circ_x x)$$

Proof. Assuming that $l_t \sqsubseteq_l l'_s$ and $\vdash_{\texttt{wfx}} x$, which implies that $\vdash_{\texttt{wfl}} l_t$, we have that $x' \circ_x x$ is defined, and we just have to check the 4 conditions of definition 16. Condition (i) follows directly from the well-formedness of x and x'. Condition (iv) can easily be obtained from this assumption as well by serial application. Condition (ii) follows directly from the definition of $\circ_c$ and the well-formedness of x and x', and the same reasoning is used to establish the left-hand side of the conjunction in condition (iii) – for the right-hand side of the conjunction, proposition 8 is used together with the well-formedness of x. □

The identity transformation on a grammar is defined as:

Definition 23 (Identity transformation idx)**.** *The identity transformation on a grammar, $l = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathit{CFG}$ is given as:*

$$\mathrm{idx}(l) \;=\; (l, l, \mathrm{id}_\tau(l), \mathrm{id}_\mathrm{c}(l))$$

where $\mathrm{id}_\tau(l) = \tau \;:\; \mathcal{N} \to \mathcal{N}$ is defined as the identity nonterminal typing $\tau(n) = n$; and where $\mathrm{id}_\mathrm{c}(l) = c \;:\; \mathcal{N} \to 2^{\mathcal{P} \times \mathcal{A}_G}$ is defined as the identity reconstructor:

$$c(n) \;=\; \{(p, A) \mid (p, \alpha) \in \pi(n),\; A = (n, p, [1, 2, \ldots, |\alpha|_{\mathcal{N}}])\}$$

Proposition 12 (Identity transformation preserves well-formedness)**.** *When the identity transformation of a well-formed grammar is constructed, the result is a well-formed transformation:*

$$\forall l \in \mathit{CFG} \ : \vdash_{\texttt{wfl}} l \ \Rightarrow \vdash_{\texttt{wfx}} \mathrm{idx}(l)$$

Proof. It is easy to verify that the 4 conditions of definition 16 holds whenever l is well-formed. □

Definition 24 (Transformation typing renaming $[\,]_\tau$)**.** *Let $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \in \mathit{CFG}$ be a grammar, $\mathcal{N}_t$ a set of nonterminals, $\tau \ : \ \mathcal{N}_s \to \mathcal{N}_t$ a nonterminal typing function, and R a renaming (as defined in Figure 2(c)) with $R \sim_r l_s$. Now, let $(\mathcal{N}'_s, \Sigma'_s, \mathcal{P}'_s, \pi'_s) \ = \ l_s[R]$ and define the renamed transformation typing $\tau[R] \ : \ \mathcal{N}'_s \to \mathcal{N}_t$ as:*

$$\tau[R](\hat{n}) \ = \ \begin{cases} \tau(n) & , \ \textit{if } R = m \,/\, n, \ \hat{n} = m \\ \tau(\hat{n}) & , \ \textit{otherwise} \end{cases}$$

Definition 25 (Transformation reconstructor renaming $[\,]_c$)**.** *Let $l_t \in \mathit{CFG}$ and $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \in \mathit{CFG}$ be grammars, $c \ : \ \mathcal{N}_s \to 2^{\mathcal{P}_s \times \mathcal{A}_{l_t}}$ a reconstructor function, and R a renaming (as defined in Figure 2(c)) with $R \sim_r l_s$. Now, let $(\mathcal{N}'_s, \Sigma'_s, \mathcal{P}'_s, \pi'_s) \ = \ l_s[R]$ and define the renamed reconstructor function $c[R] \ : \ \mathcal{N}'_s \to 2^{\mathcal{P}'_s \times \mathcal{A}_{l_t}}$ as:*

$$c[R](\hat{n}) \ = \ \begin{cases} c(n) & , \ \textit{if } R = m \,/\, n, \ \hat{n} = m \\ \left(\begin{array}{c} \{(\hat{p}, A) \,|\, (\hat{p}, A) \in c(n) \wedge \hat{p} \neq p\} \\ \quad \cup \, \{(q, A) \,|\, (p, A) \in c(n)\} \end{array} \right) & , \ \textit{if } R = q \,/\, n \,.\, p, \ \hat{n} = n \\ c(\hat{n}) & , \ \textit{otherwise} \end{cases}$$

Proposition 13 (Transformation source language renaming preserves well–formedness)**.** *Renaming the source language of a well-formed transformation with a compatible renaming, R, produces a transformation which is well-formed:*

$$\forall x = (l_s, l_t, \tau, c) \in \mathit{CATA} \ : \ R \sim_r l_s \ \wedge \ \vdash_{\texttt{wfx}} x \ \Rightarrow \vdash_{\texttt{wfx}} (x \,[\![\, R \mid]\!])$$

Proof. The transformation source renaming is defined by rule $[\textsc{ren3}_X]$ on figure 4 as $x[\![\, R \mid]\!] \ = \ (l_s[R], \, l_t, \, \tau[R], \, c[R])$. Since x is well-formed, l_s is also well-formed, and by propositions 4 and 5 this goes for $l_s[R]$ as well. Verifying that the last three conditions of definition 16 hold, is easy. □

Appendix C. Performance

Proposition 14 (Linear runtime of simple bananas)**.** *Given a well-formed transformation, $x = (l_s, l_t, \tau, c) \in CATA$, and a reconstructor, $A \in \mathcal{A}_{l_s}$: If the reconstructors of c each contain at most one instance of each gap, then $\mathrm{T}_c(A)$ can be evaluated in time $O(|A|)$, where $|A|$ counts the number of nodes (internal and leaves) in the abstract syntax tree A.*

Proof. The proof of this proposition has two steps: first we prove that if a reconstructor, A' contain at most one instance of each gap and $(i \mapsto A'_i) \vdash A' \triangleright A''$, then $|A''| \leq |A'| + \sum_i |A'_i|$. Next, we use this and structural induction on the structure of A to prove the original claim. We assume that execution time is in the order of the number of recursive calls and symbols read or generated during execution.

For the first part of the proof, we look at the definition of $\triangleright$ in Definition 17 (i) and (ii). By structural induction in A', we can see that if A' is a gap ($A' = j$), then $|A'| = 1$ and $|A''| = |A'_j| < 1 + |A'_j| \leq |A'| + \sum_i |A'_i|$. If, on the other hand, A' is not a gap, by the assumption that each gap appears at most once in A', we can split $\mu = (i \mapsto A'_i)$ in a disjoint set of partial functions μ_j such that each $i \mapsto A'_i$ appears in exactly one μ_j. This will allow us to slightly rewrite Definition 17 (i) into the following:

$$\frac{\forall j \in \{1, \ldots, m\} : \ \mu_j \vdash A_j \triangleright \hat{A}_j}{\mu \vdash (n, p, [A_1, \ldots, A_m]) \triangleright (n, p, [\hat{A}_1, \ldots, \hat{A}_m])}$$

For $\mu = (i \mapsto A'_i)$ we define $|\mu| = \sum_i |A'_i|$ and thus $|\mu| = \sum_j |\mu_j|$ due to the definition of μ_j. Now, assuming the induction hypothesis:

$$\forall j \in \{1, \ldots, m\} : |\hat{A}_j| \leq |A_j| + |\mu_j|$$

we get:

$$|A''| = 1 + \sum_i |\hat{A}_i| \leq 1 + \sum_i (|A_i| + |\mu_i|) = 1 + \sum_i |A_i| + |\mu| = |A'| + |\mu|$$

Which is what we expected. The next step is to prove that $|\mathrm{T}_c(A)| \leq K \cdot |A|$ for some constant K which depends only on the transformation x. Since the reconstructor function, c, has a finite domain and all function values are finite sets (as x is well-formed), there must be some constant, K, such that for any A, if $\exists n, p : (p, A) \in c(n)$ then $|A| \leq K$. Again, using structural induction on the rules in Definition 17 (iii) and (iv) we get the desired result. Looking at (iii), using the previous result and assuming the induction hypothesis, $\forall i \in \{1, \ldots, m\} : |A'_i| \leq K \cdot |A_i|$, we get:

$$|A''| \leq |A'| + \sum_i |A'_i| \leq K + \sum_i |A'_i| \leq K + \sum_i (K \cdot |A_i|) = K \cdot (1 + \sum_i |A_i|) = K \cdot |A|$$

Finally, we calculate the execution time as the number of recursive calls and input/output elements (the size of a node in a reconstructor tree is limited by

some small constant). All in all, the time spent on each node is limited by some constant depending only on x, which means that the total execution time must be $O(|A|)$. □

Proposition 15 (Linear runtime with concrete syntax)**.** *Given a well-formed transformation, $x = (l_s, l_t, \tau, c) \in CATA$, and a reconstructor, $A \in \mathcal{A}_{l_s}$: If the reconstructors of c each contain at most one instance of each gap, then the transformation of $\omega \in \mathcal{L}(l_s)$ to $\mathcal{L}(l_t)$ can be performed in time $O(|\omega|)$.*

Proof. Using Proposition 14 and suitable parser and pretty-print algorithms. □

Appendix D. Syntax of the Fun Language (cf. Section 8)

```
{ Wsp              = [\ \n\r] ;
  Comment          = ("//" [^\n\r]*) | ("/*" ([^*/] | (\* [^/]))* "*/") ;
  Id               = [a-zA-Z][0-9a-zA-Z_]* ;

  Exp.letexp       : LetExp ;
  LetExp.letvar    : "let" Id "=" LetExp "in" LetExp ;
        .letfun    : "let" Id "(" Id ")" "=" LetExp "in" LetExp ;
        .letrec    : "letrec" Id "(" Id ")" "=" LetExp "in" LetExp ;
        .if        : OrExp "?" LetExp ":" LetExp ;
        .orexp     : OrExp ;
  OrExp.or         : OrExp "|" AndExp ;
       .xor        : OrExp "^" AndExp ;
       .andexp     : AndExp ;
  AndExp.and       : AndExp "&" NotExp ;
        .notexp    : NotExp ;
  NotExp.not       : "!" RelExp ;
        .relexp    : RelExp ;
  RelExp.simple    : SimpleExp ;
  SimpleExp.uplus  : "+" Term ;
            .uminus : "-" Term ;
            .add   : SimpleExp "+" Term ;
            .sub   : SimpleExp "-" Term ;
            .term  : Term ;
  Term.mul         : Term "*" Factor ;
      .factor      : Factor ;
  Factor.facparen  : FacParen ;
        .noparen   : FacNoPar ;
  FacParen.paren   : "(" LetExp ")" ;
           .app    : Factor "(" LetExp ")" ;
           .iszero : "zero?" "(" LetExp ")" ;
           .succ   : "++" "(" LetExp ")" ;
           .pred   : "--" "(" LetExp ")" ;
           .cons   : "cons" "(" LetExp "," LetExp ")" ;
           .car    : "car" "(" LetExp ")" ;
           .cdr    : "cdr" "(" LetExp ")" ;
  FacNoPar.app1    : FacNoPar Prim ;
           .app2   : FacParen Prim ;
           .iszero : "zero?" Prim ;
           .succ   : "++" Prim ;
           .pred   : "--" Prim ;
           .car    : "car" Prim ;
           .cdr    : "cdr" Prim ;
           .prim   : Prim ;
  Prim.true        : "#t" ;
      .false       : "#f" ;
      .var         : Id ;
      .intConst    : IntConst ;
  IntConst.digit   : Digit ;
           .more   : IntConst Digit ;
  Digit.zero       :  "0" ;
       .one        :  "1" ;
       .two        :  "2" ;
                  ...
       .nine       :  "9" ;
}
```