Syntactic Language Extension via an Algebra of Languages and Transformations

Jacob Andersen [1]

Department of Computer Science
Aarhus University; Aarhus, Denmark

Claus Brabrand [2]

IT University of Copenhagen
Copenhagen, 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 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.

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

1 Introduction and Motivation

We propose an algebra of 16 operators on languages and transformations as a *simple*, *incremental*, and *modular* way of specifying *safe* and *efficient* syntactic language extensions through algebraic composition of previously defined languages and transformations.

[1] Email: `jacand@cs.au.dk`
[2] Email: `brabrand@itu.dk`

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 means for extending a large variety of programming languages via transformation [5,6,7]. 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.

Everything presented in the paper has been implemented in the form of The Banana Algebra Tool which, as argument, takes a transformation term of the algebra which is then analyzed for safety and *reduced* to a constant catamorphism which may subsequently be run to transform an input program.

The tool may be used for many different transformation purposes, such as transformation between different languages (e.g., for translating Java programs into HTML documentation in the style of JavaDoc or for prototyping lightweight domain-specific language compilers), transforming a given language (e.g., the CPS transformation), 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 Section 6.

The approach captures the niche where full-scale compiler generators as outlined in Section 7 are too cumbersome and where simpler techniques for syntactic transformation are not expressive or safe enough, or do not have sufficient support for incremental development.

Our contributions include the design of an algebra of languages and trans-

formations 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 Catamorphisms

A *catamorphism* (aka., *banana* [16]) 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 [16] which we will not explore in this paper. A catamorphism associates with each constructor of the datatype a *replacement evaluation function* which is used in a transformation. Given an input term of the datatype, a 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.

Many computations may be expressed as catamorphisms. As an example, let us consider an inductively defined datatype, `list`, defining non-empty lists of 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, by replacing 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 from 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 $\mathbb{N} \rightarrow \mathbb{N}$ and a replacement function of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ for `Cons`). Catamorphisms are often written in the so-called banana brackets "$(\!|\cdots|\!)$" [16]:

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

2.1 Constructive Catamorphisms

Constructive catamorphisms are a restricted form of catamorphisms where only output-typed *reconstructors* are permitted as replacement evaluator 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:

$$\texttt{tree} \;=\; \texttt{Nil} \;\mid\; \texttt{Leaf}\ \mathbb{N} \;\mid\; \texttt{Node}\ \mathbb{N}\ \texttt{*}\ \texttt{tree}\ \texttt{*}\ \texttt{tree}$$

using a constructive catamorphism:

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

Although very simple, capable of trivial recursion only, we claim that this kind of constructive catamorphisms provide a basis for programming language extension. We shall investigate this claim in the following section.

2.2 Safety and Efficiency

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

A constructive catamorphism, c, is *typed* with a source language, l_s, and a target language, l_t, as in "$l_s \rightarrow l_t$". The languages can be given either as a *datatype* (at the abstract syntactic level) as above, or as a *CFG* (at the concrete syntactic level). A constructive catamorphism 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, $\omega_s \in \mathcal{L}(l_s)$:

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

In addition to a *language typing* ($l_s \rightarrow l_t$), we also need a *nonterminal typing*, τ, which for each of the nonterminals of the input language specifies onto which nonterminal of target language they are mapped.

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 reason for the angled bracket convention is that there may be multiple nonterminals in play, in which case multiple mappings are written as a comma separated list inside the brackets.)

In order to verify that a catamorphism, $(\!|\, l_s \rightarrow l_t\ [\tau]\ c \,|\!)$ is syntactically safe, one simply needs to check that each of the catamorphism's reconstructor terms (e.g., "`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 [1] for a formal treatment of how to verify syntactic safety.

Constructive catamorphisms are highly efficient. Asymptotically, they run in linear time in the size of the input and output: $O(|\omega| + |c(\omega)|)$.

3 Language Extension

We will now illustrate—using deliberately simple examples—how constructive catamorphisms may be used to extend programming languages and motivate the idea of programming language extension. To this end, let us consider the core λ-Calculus (untyped, without constants) whose syntactic structure may be defined by the following datatype:

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

In the following, we will investigate how to extend the λ-Calculus using catamorphisms; in particular, we will look at two well-known extensions, namely that of numerals and booleans.

3.1 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 numbers 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 a catamorphism may be used 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 commonly used Church numeral representation, but the choice of encoding is not of primary interest here, so we will just use the simpler alternative to illustrate the point. We can now extend the λ-Calculus with numerals as a constructive catamorphism of type "`LN -> L [exp -> exp]`":

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

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))`.

3.2 Other Extensions

Similarly, 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 the core λ-calculus by a constructive catamorphism with typing "`LB -> L [exp -> exp]`":

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

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 6).

4 Algebra of Languages and Transformations

Investigating previous work on syntactic macros and transformations [5,6,7] 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.

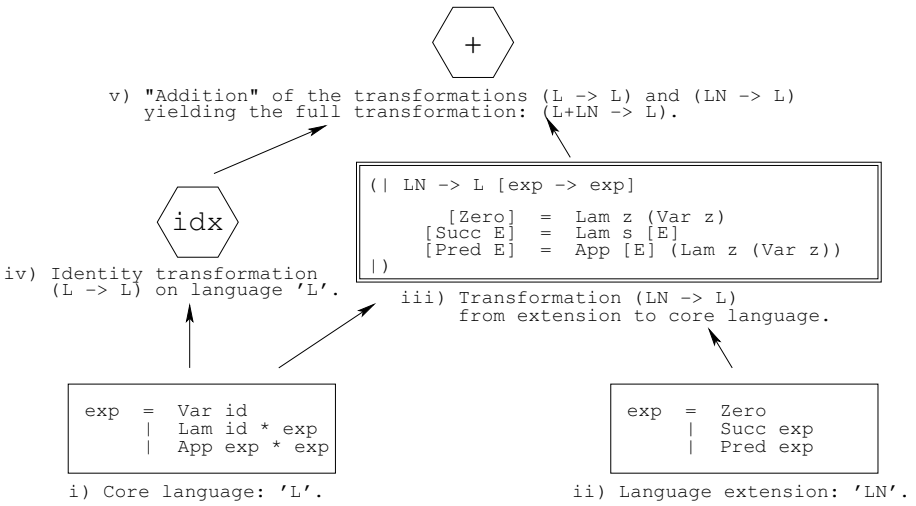

Fig. 1. Common pattern in language extension (here extending the λ-Calculus with numerals.)

In fact, every such language extension can be broken into the same five ingredients (some of which are *languages*, some of which are *transformations*), depicted in Figure 1: i) a core language that is to be extended (e.g., the λ-Calculus); ii) a language extension of that language [3] (e.g., the extension with numerals); iii) an identity transformation on the core language; iv) a transformation that maps the extended language to the core language; and 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* operator which may be modeled by a context-free grammar (with "named productions" for attaching transformations). Second, case iii) corresponds to a constant transformation which may be given as an output-typed constructive catamorphism, 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 $(l \rightarrow l)$ on that language. Fourth, a notion of addition on transformations, taking two transformations $l_s \rightarrow l_t$ and $l'_s \rightarrow l'_t$ yielding a transformation: $(l_s \oplus_l l'_s) \rightarrow (l_t \oplus_l l'_t)$ 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*, *symmetric*, *associative*, and *commutative*. For a formal definition of addition on languages and transformations, we refer to [1].

Note that with these operations, it is very easy to obtain a transformation

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

$$
\begin{array}{lll}
L & \rightarrow_{\texttt{L1}} & l \\
& \rightarrow_{\texttt{L2}} & v \\
& \rightarrow_{\texttt{L3}} & L \setminus L \\
& \rightarrow_{\texttt{L4}} & L \texttt{ + } L \\
& \rightarrow_{\texttt{L5}} & \texttt{src (X)} \\
& \rightarrow_{\texttt{L6}} & \texttt{tgt (X)} \\
& \rightarrow_{\texttt{L7}} & \texttt{let } v\texttt{=}L \texttt{ in } L \\
& \rightarrow_{\texttt{L8}} & \texttt{letx } w\texttt{=}X \texttt{ in } L
\end{array}
$$

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

$$
\begin{array}{lll}
X & \rightarrow_{\texttt{X1}} & (\!| L \rightarrow L\ [\tau]\ c |\!) \\
& \rightarrow_{\texttt{X2}} & w \\
& \rightarrow_{\texttt{X3}} & X \setminus L \\
& \rightarrow_{\texttt{X4}} & X \texttt{ + } X \\
& \rightarrow_{\texttt{X5}} & X \circ X \\
& \rightarrow_{\texttt{X6}} & \texttt{idx (L)} \\
& \rightarrow_{\texttt{X7}} & \texttt{let } v\texttt{=}L \texttt{ in } X \\
& \rightarrow_{\texttt{X8}} & \texttt{letx } w\texttt{=}X \texttt{ in } X
\end{array}
$$

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

Fig. 2. Syntax of the algebra.

combining *both* the extension of numerals and booleans; simply "add" the two transformations.

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. Note that even though the design, and choice of operators arose through an iterative process, we have tried to divide and categorize the motivations for the constructions into two categories; operators accommodating respectively *modular* and *incremental* language extension. The complete syntax for the algebra is presented in Figure 2. (The rules for *language constants*, *transformation constants*, *language addition*, *transformation addition*, and *identity transformations* are numbered `L1`, `X1`, `L4`, `X4`, and `X6`, respectively.) 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 the Appendix (for a specification of the underlying languages and transformations, see [1]).

4.2 Modular language extension

In order to permit modular language development and separate each of the ingredients in a transformation, we added *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 `L2` and `X2`) and *local definitions* (Figure 2, rules `L7`, and `X7`).

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 `L8` and `X8` to Figure 2.

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 `L5`, `L6`, `L3`, and `X3` 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 3.1 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 ∘ li2ln`" produces the exact same transformation as the directly specified constant transformation in Section 3.1. To enable such incremental development, we added *composition* as an operator on transformations (cf. Figure 2, rule `X5`).

Note that none of the operators 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.

An important advantage of an algebraic approach is that several algebraic laws hold which give rise to simplifications (e.g., "L `+` $L \equiv L$", "L_1 `+` $L_2 \equiv$ L_2 `+` L_1", "L_1 `+ (`L_2 `+` L_3`)` $\equiv$ `(`L_1 `+ (`L_2`)` `+` L_3", "`src(id(`L`))` $\equiv$ L") to mention but a few. (For a formal specification of the reductin and semantics of the operators, see the Appendix.)

```
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.

Fig. 3. Example specifying transformations using *abstract* vs. *concrete* syntax. (For emphasis, we have underlined the negation and parenthesis constructions.)

5 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.

5.1 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 on the concrete syntax. Note that because of the addition operators of the algebra, it is important that particular choice of parsing algorithm be closed under union.

Figure 3 illustrates the difference between using abstract and concrete syntax for specifying transformations. Figure 3(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 3(b) shows how this would be done at the abstract syntactic level, using abstract syntax trees (ASTs). Transformation arguments are written in angled brackets; 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

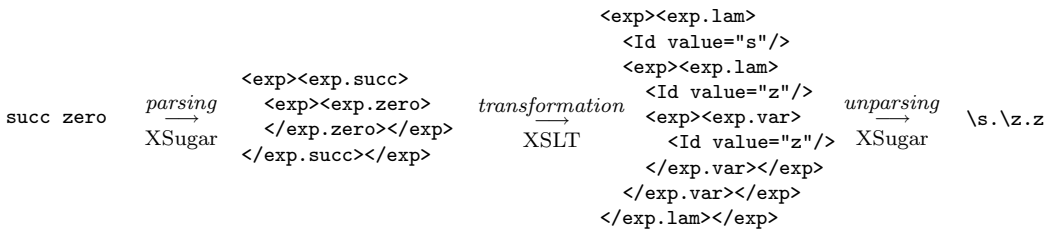

Fig. 4. The transformation process.

be negated). Figure 3(c) specifies the same transformation, but at the concrete syntactic level, using strings instead of ASTs. At this level, there is no need for dealing explicitly with such low-level considerations which are more appropriately dealt with by the parser.

Interestingly, if the grammar of a language is unambiguous and we choose a canonical unparsing, we may move reversibly between abstract syntax trees and concrete syntactic program strings. Since we have such a recent ambiguity analysis [3], we have chosen to base the tool on concrete syntax. However, transformations may also be written in abstract syntax as in Figure 3(b).

5.2 Underlying technologies

Figure 4 depicts the transformation process. The Banana Algebra Tool is currently based on XSugar [5] and XSLT [4], but the tool is easily modified to use other underlying tools (only code generation is affected by these choices). We use XSugar for parsing a concrete term of the source language (e.g., "`succ zero`") to an AST represented in XML. (XSugar uses an eager variant of Earley's algorithm, capable of parsing any CFG, and a conservative ambiguity analysis [3] which may be used to verify unambiguity of all languages involved.) Then, we use XSLT for performing the catamorphic transformation from source AST to target AST. Finally, XSugar unparses the AST into an output term of the target language.

5.3 Other implementation issues

We found it convenient to permit lexical structure to be specified using regular expressions, as often encountered in parser/scanner tools. However, the tool currently considers this an atomic terminal layer that cannot be transformed.

We handle whitespace via permitting a special whitespace terminal named "`$`" to be defined (it defaults to the empty regular expression). The semantics is that the whitespace is interspersed *between* all terminal and nonterminals on the right-hand-side of all productions. For embedded languages, it might be interesting to have finer grained control over this, but that is currently not supported by our tool.

[4] http://www.w3.org/

```
{
   $         =  [ \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).

Fig. 5. Banana Algebra example programs: a *language* and a *transformation*.

In the future, it would be interesting to also add a means for *alpha conversion* and *static semantics* checks on top of the syntactic specifications

6 Examples and Evaluation

The tool can be used for any syntax-directed transformation that can be expressed as catamorphisms (which includes all the transformations of Metafront [7] and XSugar [5]). This includes translation between different languages, transformations on a language, and format conversion, but here we will focus on *language extension* from each of the "four scenarios" from the introduction. Before that, however, we would like to show a concrete example program.

We will now revisit the example of extending the λ-Calculus with numerals that we have previously seen as a *catamorphism* (in Section 3.1) and later (in Figure 1) as a *general extension pattern*, motivating the algebraic approach.

Figure 5(a) shows the λ-Calculus as a Banana Algebra *language* constant (with standard whitespace, as defined by: "`$ = [ \n\t\r]*`"). Figure 5(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 5(a)) is loaded and bound to the Banana Algebra variable, *l* in the rest of the program. Then, in that program, *ln* is bound to the language containing the extension (assumed to reside in the file "`lambda-num.l`"). After this, *ln2l* is bound to the constant transformation that transforms the numeral extension to the core λ-Calculus. Finally, that constant transformation is added to `idx(l)` which is the identity transformation on the λ-Calculus.

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., `for-each` control structures, enumeration declarations, design patterns templates, and so on. Here, we will give only one simple example of a Java extension; the `repeat-until` of Figure 3(c):

```
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>));'  ;
   |)
in repeat2java + idx(java)
```

Although the Java grammar is big ("`java.l`" is a standard 575-line context-free grammar for Java), the `repeat-until` transformation is only seven lines.

More ambitiously, The Banana Algebra Tool may used to embed entire DSLs into a host language. We have used the tool to embed standard SQL constructions into the <bigwig> [4] language; e.g., the "`select-from-where`" construction may be captured by the following simple transformation:

```
stm.select = 'factor(<2>) { if (<3>) { return # \+ (<1>); } }' ;
```

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., "`(java \ while) + sql`".

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 languages and transformations (defined in terms of the core algebra):

$$\begin{array}{rcl} [\![L_1 \texttt{ << } L_2]\!]_L & = & (L_1 \texttt{ \ } L_2) \texttt{ + } L_2 \\ [\![X_1 \texttt{ << } X_2]\!]_X & = & (X_1 \texttt{ \ src(} X_2\texttt{)) + } X_2 \end{array}$$

To put the algebraic and incremental development approach to the test, we have built an entire existing functional language "Fun" (used in an undergraduate course on teaching functional programming at Aarhus University and Aalborg University). The language extends the λ-Calculus with *arithmetic*, *lists*, *pairs*, *local definitions*, *numerals* in terms of arithmetic, *signed arithmetic* in terms of booleans and pairs, *fixed-point iterators* in terms of local definitions, *types* in terms of arithmetic and pairs. The entire language is specified incrementally using 245 algebraic operators (i.e., 58 constant languages, 51 language inclusions, 28 language additions, 23 language variables, 17 constant transformations, 17 transformation additions, 14 transformation inclusions, 10 local definitions, 9 identity transformations, 8 compositions, 4 language restrictions, 4 transformation variables, and 2 source extractions). The entire transformation reduces to a constant (constructive catamorphism) transformation of size 4MB. (For more on this transformation, we refer to [1].)

7 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 [6,21] 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.

The work on *extensible syntax* [9] 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 [12], Elan [2], Stratego/XT [8], ASF+SDF [18], TXL [10], JastAdd [13], and Silver [22] 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 [7], 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 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 (e.g., Metafront [7] and XSugar [5]) indicate that they are sufficiently expressive and useful for a wide range of applications.

Of course, catamorphisms may be mimicked by disciplined style of functional programming, possibly aided by traversal functions automatically synthesized from datatypes [15], or by libraries of combinators [17]. However, since within a general purpose context, it cannot provide our level of safety guarantees and would not be able to compile-time factorize composition (although the functional techniques *deforestation/fusion* [20,11,19] may—in some instances—be used to achieve similar effects).

There exists a body of work on catamorphisms in a category theoretical setting [14,16]. 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.

8 Conclusion

The algebraic approach offers via 16 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 from 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, and Martin Sulzmann for useful comments and suggestions.

References

[1] Jacob Andersen and Claus 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.

[2] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and C. Ringeissen. An overview of elan. In *Second Intl. Workshop on Rewriting Logic and its Applications*, volume 15, 1998.

[3] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, July 2007.

[4] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

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

[6] Claus Brabrand and Michael 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.

[7] Claus Brabrand and Michael I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming Journal (SCP)*, 68(1):2–20, 2007.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[9] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, 1994.

[10] J.R. 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)*, pages 1–27, April 2004.

[11] João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 95–106. ACM, 2007.

[12] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[13] Görel Hedin and Eva Magnusson. Jastadd - a java-based system for implementing frontends. In *Electronic Notes in Theoretical Computer Science*, volume 44(2). Elsevier Science Publishers, 2001.

[14] Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In *Advanced Functional Programming, number 925 in Lecture Notes in Computer Science*, pages 267–307. Springer-Verlag, 1995.

[15] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[16] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[17] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

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

[19] Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In Jacques Garrigue and Manuel Hermenegildo, editors, *Proc. Functional and Logic Programming*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, April 2008.

[20] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:344–358, 1990.

[21] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, pages 156–165, 1993.

[22] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.

A Semantics of the algebra

We will now exploit the aforementioned self-containedness property and give a *big-step reduction semantics* for the algebra capable of reducing any language expression, L, to a constant language (context-free grammar), l; and any transformation expression, X, to a constant transformation (constructive catamorphism), $x = ⦇ l_s \rightarrow l_t \ [\tau] \ c ⦈$.

Let EXP_L denote the set of all language expressions from the syntactic category, L; and let EXP_X denote the set of all transformation expressions from the syntactic category, X. Also, we take VAR to be the set of all variables. We define environments in a straightforward way:

$$ENV_L : VAR \to EXP_L \qquad ENV_X : VAR \to EXP_X$$

The reduction semantics for the algebra of languages is defined by the relation $\Downarrow_L \subseteq ENV_L \times ENV_X \times EXP_L \times EXP_L$ (cf. Figure 1(a)). We will use the syntax "$\alpha, \beta \vdash L \Downarrow_L l$" as a shorthand for "$(\alpha, \beta, L, l) \in \Downarrow_L$". Similarly, the reduction semantics for the algebra of transformations is defined by the relation $\Downarrow_X \subseteq ENV_L \times ENV_X \times EXP_X \times EXP_X$ (cf. Figure 1(b)). Again, we will use the short-hand syntax "$\alpha, \beta \vdash X \Downarrow_X x$" instead of "$(\alpha, \beta, X, x) \in \Downarrow_X$".

Note that the reduction semantics in Figure A.1 uses a range of operators ($\vdash_{wfl}$, $\sim_l$, $\oplus_l$, $\ominus_l$, $\sqsubseteq_l$, $\vdash_{wfx}$, $\sim_x$, $\oplus_x$, $\ominus_x$, id_τ, id_c) which all operate on the level below that of the algebra; i.e., on constant languages (context-free grammars) and transformations (constructive catamorphisms). They can all be defined either at a *concrete* or *abstract* syntactic level. We refer to [1], for a formal specification of these lower-level operators in terms of abstract syntax.

$$[\text{CON}]_L \ \frac{}{\alpha, \beta \vdash l \Downarrow_L l} \quad \vdash_{wfl} l$$

$$[\text{VAR}]_L \ \frac{}{\alpha, \beta \vdash v \Downarrow_L \alpha(v)}$$

$$[\text{RES}]_L \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L \ \backslash \ L' \Downarrow_L l \ominus_l l'}$$

$$[\text{ADD}]_L \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha, \beta \vdash L' \Downarrow_L l'}{\alpha, \beta \vdash L + L' \Downarrow_L l \oplus_l l'} \quad l \sim_l l'$$

$$[\text{SRC}]_L \ \frac{\alpha, \beta \vdash X \Downarrow_X \llbracket l_s \rightarrow l_t \ [\tau] \ c \rrbracket}{\alpha, \beta \vdash \texttt{src (} X \texttt{)} \ \Downarrow_L l_s}$$

$$[\text{TGT}]_L \ \frac{\alpha, \beta \vdash X \Downarrow_X \llbracket l_s \rightarrow l_t \ [\tau] \ c \rrbracket}{\alpha, \beta \vdash \texttt{tgt (} X \texttt{)} \ \Downarrow_L l_t}$$

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

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

(a) Semantics for the algebra of languages.

$$[\text{CON}]_X \ \frac{\alpha, \beta \vdash L_s \Downarrow_L l_s \qquad \alpha, \beta \vdash L_t \Downarrow_L l_t}{\alpha, \beta \vdash \llbracket L_s \rightarrow L_t \ [\tau] \ c \rrbracket \Downarrow_X \llbracket l_s \rightarrow l_t \ [\tau] \ c \rrbracket} \quad \vdash_{wfx} \llbracket l_s \rightarrow l_t \ [\tau] \ c \rrbracket$$

$$[\text{VAR}]_X \ \frac{}{\alpha, \beta \vdash w \Downarrow_X \beta(w)}$$

$$[\text{RES}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash X \ \backslash \ L \Downarrow_X x \ominus_x l}$$

$$[\text{ADD}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash X + X' \Downarrow_X x \oplus_x x'} \quad x \sim_x x'$$

$$[\text{COMP}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X \llbracket l_s \rightarrow l_t \ [\tau] \ c \rrbracket \qquad \alpha, \beta \vdash X' \Downarrow_X \llbracket l'_s \rightarrow l'_t \ [\tau'] \ c' \rrbracket}{\alpha, \beta \vdash X' \circ X \Downarrow_X \llbracket l_s \rightarrow l'_t \ [\tau' \circ \tau] \ c' \circ_c c \rrbracket} \quad l_t \sqsubseteq_l l'_s$$

$$[\text{IDX}]_X \ \frac{\alpha, \beta \vdash L \Downarrow_L l}{\alpha, \beta \vdash \texttt{idx (} L \texttt{)} \ \Downarrow_X \llbracket l \rightarrow l \ [id_\tau(l)] \ id_c(l) \rrbracket}$$

$$[\text{LET}]_X \ \frac{\alpha, \beta \vdash L \Downarrow_L l \qquad \alpha[v \mapsto l], \beta \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \texttt{let } v\texttt{=}L \texttt{ in } X' \Downarrow_X x'}$$

$$[\text{LETX}]_X \ \frac{\alpha, \beta \vdash X \Downarrow_X x \qquad \alpha, \beta[w \mapsto x] \vdash X' \Downarrow_X x'}{\alpha, \beta \vdash \texttt{letx } w\texttt{=}X \texttt{ in } X' \Downarrow_X x'}$$

(b) Semantics for the algebra of transformations.

Fig. A.1. Semantics of the algebra.

In the following, we will briefly explain each of the rules of Figure A.1:

- $[\text{CON}]_L$: A constant language, l, is just a *context-free grammar* with explicitly named productions; i.e., a set of nonterminals, a set of terminals, a set of production names, and a production function (cf. B.1). Such a grammar l is *well-formed* (written "$\vdash_{wfl} l$") if there are no two productions with same production name, but different right-hand-sides (cf. B.2).
- $[\text{VAR}]_L$: A language variable, v, is simply looked up in the language environment, α, which contains the resulting constant language; $l = \alpha(v)$.
- $[\text{RES}]_L$: Language restriction "$\ominus_l$" subtracts a language from another. The result is equal to the first argument language, but where all the productions appearing in the second argument language have been removed (cf. B.6).
- $[\text{ADD}]_L$: Addition "$\oplus_l$" 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. B.4). The operation is only well-defined if the two operands are *addition compatible* (written "$l \sim_l l'$", cf. B.3); i.e., that they do not define different production right-hand-sides for the same production name.
- $[\text{SRC}]_L$: A transformation expression, X, is reduced to a constant transformation, $x = \text{⦇}\, l_s \rightarrow l_t\ [\tau]\ c \,\text{⦈}$ from which the *source language*, l_s, is extracted.
- $[\text{TGT}]_L$: As above, except that the result is the *target language*: l_t.
- $[\text{LET/LETX}]_L$: Straightforward (as found in many functional languages).

For the transformations, the rules are:

- $[\text{CON}]_X$: A constant transformation is just an *output-typed constructive catamorphism*, $x = \text{⦇}\, L_s \rightarrow L_t\ [\tau]\ c \,\text{⦈}$ (cf. C.2). In case the transformation is specified using composite source and target language expressions, L_s and L_t, they will be reduced to the constant languages, l_s and l_t, respectively. A constant transformation is *well-formed* (written "$\vdash_{wfx} x$") if all its reconstructors are valid syntax of the output language, assuming that all arguments are valid syntax of nonterminal types as dictated by the nonterminal typing function, τ (cf. C.6).
- $[\text{VAR}]_X$: A transformation variable, w, is looked up in the transformation environment, β, which contains the resulting constant transformation; $x = \beta(w)$.
- $[\text{RES}]_X$: "X `\` L" is equivalent to: "$X \circ$ `(idx(src(`X`) \ `L`))`".
- $[\text{ADD}]_X$: Addition on transformations "$\oplus_x$" is defined as reducing both operands to constant transformations, $\text{⦇}\, l_s \rightarrow l_t\ [\tau]\ c \,\text{⦈}$ and $\text{⦇}\, l'_s \rightarrow l'_t\ [\tau']\ c' \,\text{⦈}$, and constructing the 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. C.13). Addition is only well-defined if the two operands addition compatible (written "$x \sim_x x'$"); i.e., the source and target languages are addition compatible and that the transformations do not define different typings for the same nonterminals, nor that they define different reconstructors for the same productions (cf. C.12).
- $[\text{COMP}]_X$: Two transformations are sequentially composed by reducing both to constants, $\text{⦇}\, l_s \rightarrow l_t\ [\tau]\ c \,\text{⦈}$ and $\text{⦇}\, l'_s \rightarrow l'_t\ [\tau']\ c' \,\text{⦈}$, and constructing the composed transformation from l_s directly to l'_t with typing $\tau' \circ \tau$ and reconstructors $c' \circ c$ (cf. C.15). Note that 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. B.8).

- $[\text{IDX}]_X$: A language expression, L, is reduced to a constant language, l, which is then lifted to an identity transformation from l to l with identity nonterminal typing (all nonterminals are mapped to themselves) and identity reconstructors (all reconstructors are equal to constructors, cf. C.17).
- $[\text{LET/LETX}]_X$: Straightforward (as found in many functional languages).

B Semantics of Languages

B.1 Production-named CFG

We begin by introducing a variant of context-free grammars with explicitly named productions. A *production-named context-free grammar* is like a context-free grammar, except there is no start nonterminal, and productions are explicitly named, with the exception, that when a nonterminal generates a regular language, this may be abbreviated into an anonymous regular expression (which does not carry production names).

Definition B.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
- $\pi : \mathcal{N} \to 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} \cup \text{Reg}(\Sigma)$ is the "production function", where $\text{Reg}(\Sigma)$ is the set of regular languages over Σ

Finally, let $\mathbb{G}$ be the set of all production-named CFGs.

Note from this definition, how a nonterminal $n \in \mathcal{N}$ *either* generates a regular language *without* a production name ($\pi(n) \in \text{Reg}(\Sigma) \setminus \{\emptyset\}$) *or* 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* ($\pi(n) = \emptyset \in 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} \cap \text{Reg}(\Sigma)$). 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 this work it will prove very useful, as we will see below.

Henceforth, we will use the term "grammar" for a production-named CFG.

In order to avoid duplicate use of production names for multiple different production rules for the same nonterminal, we need the following:

Definition B.2 [Well-formed grammar $\vdash_{wfl}$] A grammar, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$, is said to be "well-formed" (written "$\vdash_{wfl} G$") iff $\forall n \in \mathcal{N}$:

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

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

In words, a production name may be used for at most one production rule per nonterminal, and all nonterminals must be either defined or used by π.

B.2 Language operators

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

Definition B.3 [Addition compatibility of grammars $\sim_l$] Two grammars, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $G' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ are said to be "addition compatible" (written $G \sim_l G'$) iff:

$$
\begin{array}{l}
\forall n \in \mathcal{N} \cap \mathcal{N}' : \\
\pi(n) \neq \emptyset \quad \wedge \quad \pi'(n) \neq \emptyset \\
\Downarrow \\
\left(\begin{array}{ll}
\pi(n) \in \mathrm{Reg}(\Sigma) & \Rightarrow \ \left(\pi'(n) \in \mathrm{Reg}(\Sigma \cap \Sigma') \quad \wedge \quad \pi(n) = \pi'(n) \right) \\
& \wedge \\
\pi(n) \in 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} & \Rightarrow \left(\begin{array}{c} \pi'(n) \in 2^{\mathcal{P}' \times (\mathcal{N}' \cup \Sigma')^*} \quad \wedge \\ \forall (p, \alpha) \in \pi(n),\ (p', \alpha') \in \pi'(n) \ : \ p = p' \Rightarrow \alpha = \alpha' \end{array} \right)
\end{array} \right)
\end{array}
$$

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 use instance to the defining instance.

Definition B.4 [Grammar addition $\oplus_l$] Binary addition on grammars $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $G' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ is defined when $G \sim_l G'$ as:

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

where $\pi \oplus_\pi \pi' : \mathcal{N} \cup \mathcal{N}' \to 2^{(\mathcal{P} \cup \mathcal{P}') \times ((\mathcal{N} \cup \mathcal{N}') \cup (\Sigma \cup \Sigma'))^*} \cup \mathrm{Reg}(\Sigma \cup \Sigma')$ 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 B.5 (Grammar addition preserves well-formedness) *When adding together two well-formed grammars, the resulting grammar will be well-formed:*

$$\forall G, G' \in \mathbb{G} \ : \ (G \sim_l G' \wedge \ \vdash_{wfl} G \wedge \ \vdash_{wfl} G') \ \Rightarrow \vdash_{wfl} (G \oplus_l G')$$

Proof. For part (i) of the well-formedness statement, since both G and G' 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 G and G' are addition compatible, and the top part (above the '$\Downarrow$') of definition B.3 is true, we have the bottom part (inside the big parentheses); and since the left-hand side of the '$\Rightarrow$' on the last line holds, the right-hand side must hold as well, finishing this part of the proof.

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

Definition B.6 [Grammar subtraction $\ominus_l$] Binary subtraction on grammars $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $G' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ is defined when $G \sim_l G'$ as:

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

where $\pi \ominus_\pi \pi' : \mathcal{N} \to 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} \cup \mathrm{Reg}(\Sigma)$ 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 $G \ominus_l G'$ is to remove a subset of the productions in a grammar G – more precisely the productions found in both G and G'. 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 $G \sim_l G'$, we have that either $\pi(n)$ and $\pi'(n)$ are both regular languages, in which case they are equal, and $\pi(n) \setminus \pi'(n) = \emptyset$ (removing the single production that generated n, leaving it undefined), or they are both sets of named productions with no conflicts, resulting in the removal of the common set of productions.

Proposition B.7 (Grammar subtraction preserves well-formedness) *When subtracting a grammar from a well-formed grammar, the resulting grammar will be well-formed:*

$$\forall G, G' \in \mathbb{G} \;:\; (G \sim_l G' \;\wedge\; \vdash_{wfl} G) \;\Rightarrow \vdash_{wfl} (G \ominus_l G')$$

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 $\widehat{\mathcal{N}}$. □

Definition B.8 [Grammar inclusion $\sqsubseteq_l$] Grammar inclusion is a binary relation on

grammars $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi)$ and $G' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi')$ defined as:

$$G \sqsubseteq_l G' \;\equiv\; \forall n \in \mathcal{N} : \pi(n) = \emptyset \;\vee \left(\begin{array}{l} n \in \mathcal{N}' \\ \wedge \\ \pi(n) \in \mathrm{Reg}(\Sigma) \;\Rightarrow\; \pi(n) = \pi'(n) \\ \wedge \\ \pi(n) \in 2^{\mathcal{P} \times (\mathcal{N} \cup \Sigma)^*} \;\Rightarrow\; \pi(n) \subseteq \pi'(n) \end{array} \right)$$

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 B.9 (Grammar inclusion and well-formedness) *Let* $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$ *and* $G' = (\mathcal{N}', \Sigma', \mathcal{P}', \pi') \in \mathbb{G}$ *then:*

$$\vdash_{wfl} G \;\wedge\; G \sqsubseteq_l G' \;\Rightarrow\; \mathcal{N} \subseteq \mathcal{N}'$$

Proof. Let $n \in \mathcal{N}$. Since G 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 $n \in \mathcal{N}'$. □

C Semantics of Transformations

C.1 Transformations defined

We now proceed toward defining transformations as a way to express a syntactic (structural) translation of concrete expressions of a source language given by a grammar (production-named CFG) into expressions of a target language given by another grammar.

In order to define such transformations, we need to be able to represent *reconstructors*:

Definition C.1 [Reconstructor] Given a grammar, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$, the set $\mathcal{A}_G$ of *reconstructors for* G is given by the (least fixed point of the) following recursive set definition:

$$\begin{aligned} \mathcal{A}_G : & \mathcal{N} \times \mathcal{P} \times {\mathcal{A}_G}^* \\ & : \Sigma^* \\ & : \mathbb{N} \end{aligned}$$

assuming, of course, that $\Sigma^* \cap \mathbb{N} = \emptyset$.

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

- $l_s = (\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s) \in \mathbb{G}$ is the "source language grammar",
- $l_t = (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t, \pi_t) \in \mathbb{G}$ 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".

Now, let $\mathbb{C}$ be the set of all transformations.

As a number of things can go wrong in this definition, resulting in transformations that do not yield valid output, we proceed to define a well-formedness relation on the set $\mathbb{C}$ of transformations. In order to define this relation, however, we need some relations:

Definition C.3 ["Sequence picker" operator $|_P$] In order to pick elements out of a sequence, we define the "sequence picker" operator $|_P : Q^* \times \mathbb{N} \hookrightarrow P$. For a sequence $s \in Q^*$ and a number $i \in \mathbb{N}$, $s|_P^i$ yields the ith element of P found in the sequence s, and it is undefined if such an element does not exist. Of course, for this operator to be defined for any i, we must have $P \cap Q \neq \emptyset$. A formal definition of this operator using an inference system would look like the following:

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

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

Definition C.4 ["Sequence counter" operator $|\cdot|_P$] Given a sequence, s like the previous definition, we define $|\cdot|_P : Q^* \to \mathbb{N}$ as the number of P elements found in the sequence, i.e. $|s|_P$ is the maximal value for i, where $s|_P^i$ is defined, and if $|s|_P = 0$, then $s|_P^i$ is not defined for any i. A formal definition would be the following:

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

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

Definition C.5 [Reconstructor typing $\vdash_G$] Given a grammar, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$ and a "gap type function" $\rho : \mathbb{N} \hookrightarrow \mathcal{N}$, we say that a reconstructor $A \in \mathcal{A}_G$ has type $n \in \mathcal{N}$, written $\rho \vdash_G A : n$ iff it is provable by the following inference system:

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

$$\frac{}{\rho \vdash_G \omega : n} \quad \omega \in \Sigma^*,\ \omega \in \pi(n) \in \mathrm{Reg}(\Sigma)$$

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

Definition C.6 [Well-formed transformation $\vdash_{wfx}$] The transformation, $C = ((\mathcal{N}_s, \Sigma_s, \mathcal{P}_s, \pi_s), (\mathcal{N}_t, \Sigma_t, \mathcal{P}_t,$ $\mathbb{C}$, is said to be "well-formed" (written "$\vdash_{wfx} C$") iff all of the following conditions are met:

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

(ii) $$\forall n \in \mathcal{N}_s\ :\ \left(\begin{array}{c} \Big(\pi_t(\tau(n)) \in \mathrm{Reg}(\Sigma_t)\ \Rightarrow\ \pi_s(n) \in \mathrm{Reg}(\Sigma_s)\Big) \\ \wedge \\ \Big(\pi_t(\tau(n)) \in 2^{\mathcal{P}_t \times (\mathcal{N}_t \cup \Sigma_t)^*}\ \Rightarrow\ \pi_s(n) \in 2^{\mathcal{P}_s \times (\mathcal{N}_s \cup \Sigma_s)^*}\Big) \end{array} \right)$$

(iii) $$\forall n \in \mathcal{N}_s \ : \ \pi_s(n) \in \mathrm{Reg}(\Sigma_s) \setminus \{\emptyset\} \ \Rightarrow \ \pi_s(n) \subseteq \pi_t(\tau(n))$$

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

(v) $$\forall n \in \mathcal{N}_s \ \forall (p,A) \in c(n) \ \exists \alpha \in (\mathcal{N}_s \cup \Sigma_s)^* \ : \\ (p,\alpha) \in \pi_s(n) \ \wedge \ (\tau \circ \alpha|^{\cdot}_{\mathcal{N}_s}) \vdash_{l_t} A : \tau(n)$$

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

(i) expresses that the source and target language grammars must be well-formed. Condition (ii) states that nonterminals defined by a regular expression in the source grammar must map to a nonterminal defined by a regular expression in the target grammar, and conversely nonterminals defined by named productions must map to nonterminals defined by named productions, and finally, undefined nonterminals in the source language may map to any type. This restriction is somewhat limiting to the utility of the algebra, but unfortunately attempts to avoid it have failed so far. Condition (iii) takes care of type safety when a nonterminal defined as a regular expression is transformed. In this case the transformation process reduces to copying the value, so the transformation is well-formed if any value accepted by the source language nonterminal will also be accepted by the target language nonterminal. The final two conditions deal with the transformation of nonterminals defined by named productions. Condition (iv) 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 (v) verifies that the typing of all transformation rules matches the typing defined by τ, and finally (vi) ensures that only one rule exists per production.

C.2 Applying transformations

Definition C.7 [The transformation relation] Given a grammar and a reconstructor function, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$, and $c : \mathcal{N} \to 2^{\mathcal{P} \times \mathcal{A}_G}$, we now define the "substitute relation", $\triangleright \subseteq (\mathbb{N} \to \mathcal{A}_G) \times \mathcal{A}_G \times \mathcal{A}_G$, written $\rho \vdash A \triangleright A'$ in the following manner:

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

(ii) $$\frac{}{\rho \vdash \omega \triangleright \omega} \quad \omega \in \Sigma^*$$

(iii) $$\frac{}{\rho \vdash i \triangleright \rho(i)} \quad i \in \mathbb{N}$$

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

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

(v) $$\frac{}{\omega \longrightarrow_c \omega} \quad \omega \in \Sigma^*$$

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

Proposition C.8 (The transformation relation may be viewed as a function) *Let $C = (l_s, l_t, \tau, c) \in \mathbb{C}$ 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 C is well-formed. This is easily verified by structural induction first for $\triangleright$ and then for $\longrightarrow_c$, using condition (vi) of definition C.6.

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 $A \in \mathbb{N}$ or $A \in \Sigma_s^*$, 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 C 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 C.9 [Transformation function T_c] Let $C = (l_s, l_t, \tau, c) \in \mathbb{C}$ 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, $\mathrm{T}_c : \widehat{\mathcal{A}}_{l_s} \rightarrow \mathcal{A}_{l_t}$:

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

Lemma C.10 (Typing of substitute relation) *The substitute relation preserves the original type when the substituted reconstructors are correctly typed:*

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

Proof. We will prove this by induction in the structure of A. If $A \in \Sigma^*$ the statement is trivial as $A = A'$. If $A \in \mathbb{N}$ by definition C.5 $\sigma(A) = n$ and with the rest of the preconditions and (iii) of definition C.7 we are done. When $A = (n, p, [A_1, \ldots, A_m])$ from definition C.5 we get $\exists \alpha : (p, \alpha) \in \pi(n) \wedge m = |\alpha|_{\mathcal{N}}$, which together with the induction hypothesis finishes the proof. □

Proposition C.11 (Typing of transformed reconstructor) *A well-formed transformation, $C = (l_s, l_t, \tau, c)$, 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.

$A \in \Sigma_s^*$:

From the definition of the reconstructor typing we have $\pi_s(n) \in \mathrm{Reg}(\Sigma_s)$ and $A \in \pi_s(n)$, so $\pi_s(n) \neq \emptyset$. Since the transformation is well-formed, condition (iii) of definition C.6 states that $\pi_s(n) \subseteq \pi_t(\tau(n))$, and from the definition of the transformation, we have that $\mathrm{T}_c(A) = A$ in this case, so $\mathrm{T}_c(A) \in \pi_t(\tau(n))$, and we are done.

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

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 C.6 condition (iv) gives us $\exists A' \in \mathcal{A}_{l_t} : (p, A') \in c(n)$, and from condition (v) and the well-formedness of l_s we get $(\tau \circ \alpha|_{\mathcal{N}_s}^{\cdot}) \vdash_{l_t} A' : \tau(n)$. Since $|\alpha|_{\mathcal{N}_s} = 0$, $(\tau \circ \alpha|_{\mathcal{N}_s}^{\cdot})$ 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:

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

Now, examining the definition of T_c and using lemma C.10 we obtain the result. □

C.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 C.12 [Addition compatibility of transformations $\sim_x$] Two transformations, $C = (l_s, l_t, \tau, c)$ and $C' = (l'_s, l'_t, \tau', c')$ – 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 $C \sim_x C'$) iff the following conditions are met:

$$\text{(i)} \qquad l_s \sim_l l'_s \ \wedge \ l_t \sim_l l'_t$$

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

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

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 C.13 [Transformation addition $\oplus_x$] Binary addition on transformations, $C = (l_s, l_t, \tau, c)$ and $C' = (l'_s, l'_t, \tau', c')$ – 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:

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

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 C.14 (Transformation addition preserves well-formedness) *When adding together two well-formed transformations, the resulting transformation will be well-formed:*

$$\forall C, C' \in \mathbb{C} \ : \ (C \sim_x C' \ \wedge \ \vdash_{wfx} C \ \wedge \ \vdash_{wfx} C') \ \Rightarrow \vdash_{wfx} (C \oplus_x C')$$

Proof. We need to establish the 6 conditions of definition C.6. The first condition follows directly from the proposition B.5 and the fact that C and C' – and hence all 4 grammars in play – are well-formed. (vi) follows from the well-formedness of the grammars as well as the addition compatibility, and the proofs for condition

(ii)–(v) are very similar, so we will only show (v), which is the more elaborate part, here:

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

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

Picking n and (p, A), we notice that 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 $\vdash_{wfx} C$ we have that α exists and satisfies the left-hand side of the conjunction. As $C \sim_x C'$, 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 $\vdash_{wfx} C$, we have that: $(\tau \circ \alpha|^{\cdot}_{\mathcal{N}_s}) \vdash_{l_t} A : \tau(n)$; 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 C.15 [Transformation composition $\circ_x$] Composition on two transformations, $C = (l_s, l_t, \tau, c)$ and $C' = (l'_s, l'_t, \tau', c')$ – 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 $\vdash_{wfl} l_t$ and $l_t \sqsubseteq_l l'_s$ as:

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

Note that $\tau' \circ \tau$ is defined due to proposition B.9, 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 C.16 (Transformation composition preserves well-formedness) *When composing two well-formed transformations, the resulting transformation will be well-formed:*

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

Proof. Assuming that $l_t \sqsubseteq_l l'_s$ and $\vdash_{wfx} C$, which implies that $\vdash_{wfl} l_t$, we have that $C' \circ_x C$ is defined, and we just have to check the 6 conditions of definition C.6. Condition (i) follows directly from the well-formedness of C and C'. Condition (ii), (iii) and (vi) can easily be obtained from this assumption as well by serial application. Condition (iv) follows directly from the definition of $\circ_c$ and the well-formedness of C, and the same reasoning is used to establish the left-hand side of the conjunction in condition (v) – for the right-hand side of the conjunction, proposition C.11 is used together with the well-formedness of C. □

The identity transformation on a grammar is defined as:

Definition C.17 [Identity transformation idx_l] The identity transformation on a grammar, $G = (\mathcal{N}, \Sigma, \mathcal{P}, \pi) \in \mathbb{G}$ is given as:

$$\text{idx}_l(G) \;=\; (G, G, (n \mapsto n), \text{idc}_G)$$

where $\text{idc}_G \;:\; \mathcal{N} \to 2^{\mathcal{P} \times \mathcal{A}_G}$ is defined by:

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

Proposition C.18 (Identity transformation preserves well-formedness) *When the identity transformation of a well-formed grammar is constructed, the result is a well-formed transformation:*

$$\forall G \in \mathbb{G} \;:\vdash_{wfl} G \;\Rightarrow \vdash_{wfx} \text{idx}_l(G)$$

Proof. It is easy to verify that the 6 conditions of definition C.6 holds whenever G is well-formed. □