Parsing and Generation as Datalog Query Evaluation

Makoto Kanazawa
National Institute of Informatics, 2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430, Japan

Abstract

Parsing and generation (or surface realization) are two of the most important tasks in the processing of natural language by humans and by computers. This paper studies both tasks in the style of formal language theory, using typed λ-terms to represent meanings. It is shown that the problems of parsing and surface realization for grammar formalisms with "context-free" derivations, coupled with a kind of Montague semantics (satisfying a certain restriction) can be reduced in a uniform way to Datalog query evaluation. This makes it possible to apply to parsing and surface realization known efficient evaluation methods for Datalog. Moreover, the reduction has the following complexity-theoretic consequences for all such formalisms: (i) the decision problem of recognizing grammaticality (surface realizability) of an input string (logical form) is in LOGCFL; and (ii) the search problem of computing all derivation trees (in the form of shared forest) from an input string or input logical form is in functional LOGCFL. These bounds are tight. The reduction is carried out by way of "context-free" grammars on typed λ-terms, a relaxation of the second-order fragment of de Groote's abstract categorial grammar. The method works whenever a grammar uses only "almost linear" λ-terms.

Keywords: Generation, Surface Realization, Parsing, Datalog, LOGCFL, Montague Semantics, Abstract Categorial Grammar, Typed Lambda Calculus, Almost Linear Lambda Term.

1 Introduction

The representation of context-free grammars (augmented with features) in terms of definite clause programs is well-known. In the case of a bare-bone CFG, the

This paper originally evolved from an inspiration I got from a discussion with Sylvain Salvati on his work [63]. I was supported by the Japan Society for the Promotion of Science under the Grant-in-Aid for Scientific Research (KAKENHI), Grant Numbers 19500019 and 21500025.

corresponding program is in the function-free subset of logic programming, known as *Datalog*. For example, determining whether a string John found a unicorn belongs to the language of the CFG in (1) is equivalent to deciding whether the Datalog program in (2) together with the *database* in (3) can derive the goal or *query* (4):[1]

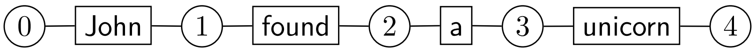
$$\begin{array}{lll} \mathsf{S} \to \mathsf{NP}\ \ \mathsf{VP} & \mathsf{NP} \to \mathsf{John} & \mathsf{Det} \to \mathsf{a} \\ \mathsf{VP} \to \mathsf{V}\ \ \mathsf{NP} & \mathsf{V} \to \mathsf{found} & \mathsf{N} \to \mathsf{man} \\ \mathsf{V} \to \mathsf{V}\ \ \mathsf{and}\ \ \mathsf{V} & \mathsf{V} \to \mathsf{caught} & \mathsf{N} \to \mathsf{unicorn} \\ \mathsf{NP} \to \mathsf{Det}\ \ \mathsf{N} & \mathsf{V} \to \mathsf{is} & \end{array} \tag{1}$$

$$\begin{array}{ll} \mathsf{S}(i,j) :\!- \mathsf{NP}(i,k), \mathsf{VP}(k,j). & \mathsf{V}(i,j) :\!- \mathsf{caught}(i,j). \\ \mathsf{VP}(i,j) :\!- \mathsf{V}(i,k), \mathsf{NP}(k,j). & \mathsf{V}(i,j) :\!- \mathsf{is}(i,j). \\ \mathsf{V}(i,j) :\!- \mathsf{V}(i,k), \mathsf{and}(k,l), \mathsf{V}(l,j). & \mathsf{Det}(i,j) :\!- \mathsf{a}(i,j). \\ \mathsf{NP}(i,j) :\!- \mathsf{Det}(i,k), \mathsf{N}(k,j). & \mathsf{N}(i,j) :\!- \mathsf{man}(i,j). \\ \mathsf{NP}(i,j) :\!- \mathsf{John}(i,j). & \mathsf{N}(i,j) :\!- \mathsf{unicorn}(i,j). \\ \mathsf{V}(i,j) :\!- \mathsf{found}(i,j). & \end{array} \tag{2}$$

$$\mathsf{John}(0,1).\ \ \mathsf{found}(1,2).\ \ \mathsf{a}(2,3).\ \ \mathsf{unicorn}(3,4). \tag{3}$$

$$?\!- \mathsf{S}(0,4). \tag{4}$$

In the Datalog representation, terminals and nonterminals of the CFG are interpreted as binary predicates on positions within the input string. The database representing a string can be viewed as a certain type of directed graph (called a *string graph*). We depict a string graph by a diagram like (5), where circles represent nodes (string positions) and boxes are labels of directed edges, which, by convention, point from left to right.



(5)

By *naive* (or *seminaive*) bottom-up evaluation (see, e.g., [76] or [1]), the answer to a query like (4) can be computed in polynomial time in the size of the database, for any fixed Datalog program. This method of evaluation generates all facts derivable from the program together with the input database in the order of the height of the Datalog derivation tree, until no new fact is derivable. By recording ground instances of rules used to derive facts, a packed representation of the complete set of Datalog derivation trees for a given query can also be obtained in polynomial time using this technique. Since a Datalog derivation tree uniquely determines a grammar derivation tree and vice versa (Figure 1), the translation gives a reduction

[1]The term *query* means different things in logic programming/Prolog and relational database theory/finite model theory. The use of the term in this paper follows the logic programming/Prolog tradition.

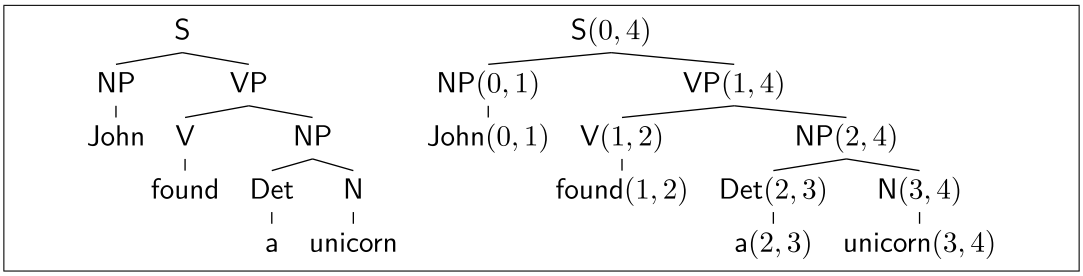


Figure 1: A CFG derivation tree (left) and a Datalog derivation tree (right).

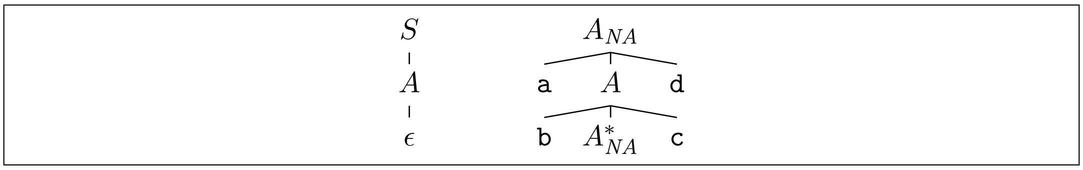


Figure 2: A TAG with one initial tree (left) and one auxiliary tree (right)

of context-free recognition and parsing to query evaluation in Datalog. This is of course all well known and well understood, even though the Datalog parlance is not universally adopted.

In this paper, I extend this reduction in two directions. First, I show that a similar reduction to Datalog is possible for more powerful grammar formalisms that have "context-free" derivations, such as *(multi-component) tree-adjoining grammars* [37, 80], *IO macro grammars* [24], and *(parallel) multiple context-free grammars* [66]. For instance, the tree-adjoining grammar in Figure 2 is represented by the Datalog program in (6).

$$\begin{aligned}
&S(i_1, i_3) :\!- A(i_1, i_3, i_2, i_2). \\
&A(i_1, i_8, i_4, i_5) :\!- \mathsf{a}(i_1, i_2), \mathsf{b}(i_3, i_4), \mathsf{c}(i_5, i_6), \mathsf{d}(i_7, i_8), A(i_2, i_7, i_3, i_6). \\
&A(i_1, i_2, i_1, i_2).
\end{aligned} \tag{6}$$

Second, I extend the technique to the problem of *tactical generation* (*surface realization*) for such "context-free" grammar formalisms supplemented with a kind of *Montague semantics* [57], under a certain restriction to be made precise below. The method of reduction is uniform in both cases, and essentially relies on the encoding of different formalisms in terms of *abstract categorial grammars* [17].

The reduction to Datalog makes it possible to apply to parsing and generation sophisticated evaluation techniques for Datalog queries; in particular, an application of *generalized supplementary magic-sets* rewriting [8] automatically yields Earley-style algorithms for both parsing and generation. The reduction can also be used to obtain a tight upper bound, namely *LOGCFL*, on the computational complexity of the problem of recognition of input strings as well as of the problem of checking surface realizability of input logical forms.[2] This means that, in rough complexity-theoretic terms, these problems are no more difficult than the recognition problem for context-free languages.

With regard to parsing and recognition of input strings, polynomial-time algorithms and the LOGCFL upper bound on the computational complexity are already known for the grammar formalisms covered by our results [22]. Also, efficient tabular algorithms have already been obtained for many of these formalisms, and a general perspective on tabular parsing, in the names of *deductive parsing* [69] and *parsing schemata* [70], which can be equivalently expressed in terms of Datalog, is already available. Nevertheless, I believe that my method of reduction to Datalog is of independent interest, as it shows that efficient tabular parsing (recognition) algorithms are automatically obtained from various types of grammars in a uniform way. Concerning generation, where the input is a structured expression involving *binding*, the present results seem to be entirely new.[3]

Since the precise statement of my method of reduction and the proof of its correctness are quite technical, I first give an informal exposition of the method in Section 2. I develop the theory formally, complete in all details, in Section 3. I then discuss some consequences and extensions of the main results in Section 4, before giving a brief conclusion in Section 5.

The main results of the present paper were first announced in [42]; Sections 1 and 2.1, part of Section 2.2, and Section 4.3 are based on that paper.

[2]LOGCFL is the class of decision problems that can be reduced to some context-free language by a deterministic Turing machine operating in logarithmic space, and lies between the complexity classes NL and AC^1 (see [35]). Since LOGCFL is a subclass of NC, problems in LOGCFL are efficiently parallelizable. There are context-free languages that are complete for LOGCFL under log-space reduction (see [27]).

[3]As I explain below, the present method primarily applies to *exact generation* only, where the input logical form is supposed to exactly match the logical form produced by the grammar.

2 An informal exposition

2.1 Context-free grammars on λ-terms

Let us consider an augmentation of the CFG (1) with Montague semantics, which uses λ-terms as representations of meanings:[4]

$$\begin{array}{l}
\mathsf{S}(X_1 X_2) \to \mathsf{NP}(X_1)\ \ \mathsf{VP}(X_2) \\
\mathsf{VP}(\lambda x.X_2(\lambda y.X_1 y x)) \to \mathsf{V}(X_1)\ \ \mathsf{NP}(X_2) \\
\mathsf{V}(\lambda yx.\wedge^{t\to t\to t}(X_1 y x)(X_2 y x)) \to \mathsf{V}(X_1)\ \ \mathsf{and}\ \ \mathsf{V}(X_2) \\
\mathsf{NP}(X_1 X_2) \to \mathsf{Det}(X_1)\ \ \mathsf{N}(X_2) \\
\mathsf{NP}(\lambda u.u\,\mathbf{John}^{e}) \to \mathsf{John} \\
\mathsf{V}(\mathbf{find}^{e\to e\to t}) \to \mathsf{found} \\
\mathsf{V}(\mathbf{catch}^{e\to e\to t}) \to \mathsf{caught} \\
\mathsf{V}(=^{e\to e\to t}) \to \mathsf{is} \\
\mathsf{Det}(\lambda uv.\exists^{(e\to t)\to t}(\lambda y.\wedge^{t\to t\to t}(uy)(vy))) \to \mathsf{a} \\
\mathsf{N}(\mathbf{man}^{e\to t}) \to \mathsf{man} \\
\mathsf{N}(\mathbf{unicorn}^{e\to t}) \to \mathsf{unicorn}
\end{array} \tag{7}$$

Here, the left-hand side of each rule is annotated with a λ-term that tells how the meaning of the left-hand side is composed from the meanings of the right-hand side nonterminals, represented by upper-case variables $X_1, X_2, \dots$. Note that λ-terms may contain any number of constants, whose types are indicated by superscripts.[5] In such a grammar, the meaning of a sentence is computed from its derivation tree. For example, given the derivation tree of John found a unicorn (the left tree in Figure 1), we can decorate each nonterminal node with a λ-term in accordance with the grammar rule being applied at that node, obtaining the decorated tree in Figure 3. The λ-term decorating the root node,

$$(\lambda u.u\,\mathbf{John})(\lambda x.(\lambda uv.\exists(\lambda y.\wedge(uy)(vy)))\,\mathbf{unicorn}\,(\lambda y.\mathbf{find}\,y\,x)),$$

β-reduces to the λ-term

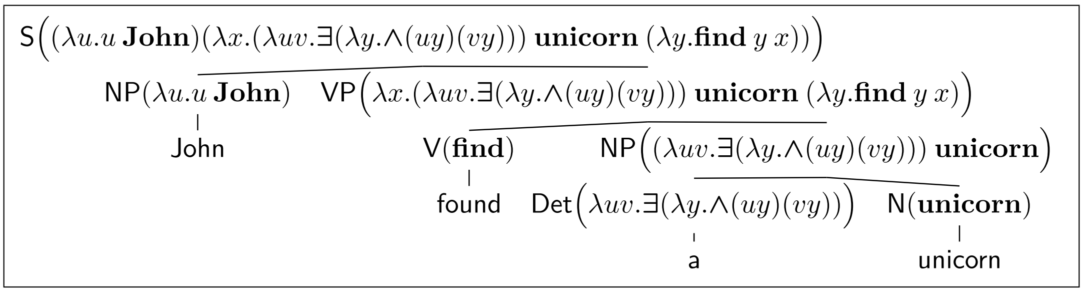
$$\exists(\lambda y.\wedge(\mathbf{unicorn}\,y)(\mathbf{find}\,y\,\mathbf{John})) \tag{8}$$

encoding the first-order logic formula representing the meaning of the sentence (i.e., its logical form):

$$\exists y(\mathbf{unicorn}(y) \wedge \mathbf{find}(\mathbf{John}, y)).$$

[4]Grammars like this one are basically *generalized phrase structure grammars* [25] without features or metarules.

[5]We follow standard notational conventions in typed λ-calculus, rather than Montague's [57]. Thus, an application $M_1 M_2 M_3$ (written without parentheses) associates to the left, $\lambda x.\lambda y.M$ is abbreviated to $\lambda xy.M$, and $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$.


$\mathsf{S}\Big((\lambda u.u\,\mathbf{John})(\lambda x.(\lambda uv.\exists(\lambda y.\wedge(uy)(vy)))\,\mathbf{unicorn}\,(\lambda y.\mathbf{find}\,y\,x))\Big)$
$\mathsf{NP}(\lambda u.u\,\mathbf{John})$
$\mathsf{VP}\Big(\lambda x.(\lambda uv.\exists(\lambda y.\wedge(uy)(vy)))\,\mathbf{unicorn}\,(\lambda y.\mathbf{find}\,y\,x)\Big)$
John
$\mathsf{V}(\mathbf{find})$
$\mathsf{NP}\Big((\lambda uv.\exists(\lambda y.\wedge(uy)(vy)))\,\mathbf{unicorn}\Big)$
found
$\mathsf{Det}\Big(\lambda uv.\exists(\lambda y.\wedge(uy)(vy))\Big)$
$\mathsf{N}(\mathbf{unicorn})$
a
unicorn
Figure 3: A decorated derivation tree of a CFG with Montague semantics.

Thus, computing the logical form(s) of a sentence—the task of *semantic interpretation*[6]—involves parsing and λ-term normalization. Conversely, to find a sentence expressing a given logical form—the task of *surface realization*—it suffices to find a derivation tree whose root node is decorated with a λ-term that β-reduces to the given logical form; the desired sentence can simply be read off from the derivation tree. At the heart of both tasks is the computation of the derivation tree(s) that yield the input. In the case of surface realization, this may be viewed as parsing the input λ-term with a "context-free" grammar that generates a set of λ-terms (in β-normal form), which is obtained from the given CFG with Montague semantics by stripping off terminal symbols:

$$\begin{aligned}
&\mathsf{S}(X_1X_2) \coloneq \mathsf{NP}(X_1), \mathsf{VP}(X_2).\\
&\mathsf{VP}(\lambda x.X_2(\lambda y.X_1yx)) \coloneq \mathsf{V}(X_1), \mathsf{NP}(X_2).\\
&\mathsf{V}(\lambda yx.\wedge^{t\to t\to t}(X_1yx)(X_2yx)) \coloneq \mathsf{V}(X_1), \mathsf{V}(X_2).\\
&\mathsf{NP}(X_1X_2) \coloneq \mathsf{Det}(X_1), \mathsf{N}(X_2).\\
&\mathsf{NP}(\lambda u.u\,\mathbf{John}^e).\\
&\mathsf{V}(\mathbf{find}^{e\to e\to t}).\\
&\mathsf{V}(\mathbf{catch}^{e\to e\to t}).\\
&\mathsf{V}(=^{e\to e\to t}).\\
&\mathsf{Det}(\lambda uv.\exists^{(e\to t)\to t}(\lambda y.\wedge^{t\to t\to t}(uy)(vy))).\\
&\mathsf{N}(\mathbf{man}^{e\to t}).\\
&\mathsf{N}(\mathbf{unicorn}^{e\to t}).
\end{aligned} \tag{9}$$

Determining whether a given logical form is surface realizable with the original grammar (7) is equivalent to recognition with the resulting *context-free λ-term grammar* (CFLG) (9). As with CFG recognition/parsing, solving the problem of recognition for CFLGs almost amounts to solving the problem of parsing; so algorithms and

[6]This is sometimes called "semantic parsing" or "parsing to logical form".

complexity results for the former translate into algorithms and complexity results for the problem of surface realization.

In a CFLG such as (9), there is a mapping f from nonterminals to their semantic types:

$$f = \left\{ \begin{aligned} \mathsf{S} &\mapsto t, \\ \mathsf{NP} &\mapsto (e \to t) \to t, \\ \mathsf{VP} &\mapsto e \to t, \\ \mathsf{V} &\mapsto e \to e \to t, \\ \mathsf{Det} &\mapsto (e \to t) \to (e \to t) \to t, \\ \mathsf{N} &\mapsto e \to t \end{aligned} \right\}.$$

A rule that has B on the left-hand side and $B_1, \ldots, B_n$ as right-hand side nonterminals has its left-hand side annotated with a well-formed λ-term M that has type $f(B)$ under the type environment $X_1 : f(B_1), \ldots, X_n : f(B_n)$, or in symbols:

$$\vdash X_1 : f(B_1), \ldots, X_n : f(B_n) \Rightarrow M : f(B).$$

For example, in the case of the third rule of (9), we have

$$\vdash X_1 : e \to e \to t, X_2 : e \to e \to t \Rightarrow \lambda yx. \wedge^{t \to t \to t} (X_1 yx)(X_2 yx) : e \to e \to t. \quad (10)$$

What we are calling a context-free λ-term grammar is nothing but an alternative notation for an *abstract categorial grammar* [17] whose abstract vocabulary is second-order, with the restriction to *linear* λ-terms removed.[7] In the linear case, Salvati [62] showed the recognition/parsing complexity to be in P, and exhibited an algorithm similar to Earley parsing for TAGs. Second-order linear ACGs are known to be expressive enough to encode well-known mildly context-sensitive grammar formalisms in a straightforward way, including TAGs and (non-deleting) multiple context-free grammars (also known as *linear context-free rewriting systems*) [18, 19].

For example, the following linear CFLG is an encoding of the TAG in Figure 2, where $f(S) = o \to o$ and $f(A) = (o \to o) \to o \to o$ (see [18] for details of this encoding):

$$\begin{aligned} & S(\lambda y. X_1(\lambda z.z) y) \text{ :- } A(X_1). \\ & A(\lambda xy. \mathtt{a}^{o \to o}(X_1(\lambda z. \mathtt{b}^{o \to o}(x(\mathtt{c}^{o \to o} z)))(\mathtt{d}^{o \to o} y))) \text{ :- } A(X_1). \\ & A(\lambda xy.xy). \end{aligned} \quad (11)$$

In encoding a string-generating grammar, a CFLG uses o as the type of string position and $o \to o$ as the type of string. Each terminal symbol is represented by a

[7]A λ-term is a *λI-term* if each occurrence of λ binds at least one occurrence of a variable. A λI-term is *linear* if no subterm contains more than one free occurrence of the same variable.

constant of type $o \to o$, and a string $a_1 \dots a_n$ is encoded by the λ-term

$$/a_1 \dots a_n/ = \lambda z.a_1^{o\to o}(\dots(a_n^{o\to o}z)\dots),$$

which has type $o \to o$.[8]

A string-generating grammar coupled with Montague semantics may be represented by a *synchronous CFLG*, a pair of CFLGs with matching rule sets, as in Figure 4.[9] The transduction between strings and logical forms in either direction consists of parsing the input λ-term with the source-side grammar and normalizing the λ-term(s) constructed in accordance with the target-side grammar from the derivation tree(s) output by parsing.

2.2 Reduction to Datalog

We can show that under a weaker condition than linearity, a CFLG can be represented by a Datalog program. The presentation in this section is informal and not fully precise; formal definitions and rigorous proof of correctness are deferred to Section 3.

We use the grammar (9) as an example, which is repeated below:

$$\begin{array}{l}
\mathsf{S}(X_1X_2) :\!-\; \mathsf{NP}(X_1), \mathsf{VP}(X_2).\\
\mathsf{VP}(\lambda x.X_2(\lambda y.X_1yx)) :\!-\; \mathsf{V}(X_1), \mathsf{NP}(X_2).\\
\mathsf{V}(\lambda yx.\wedge^{t\to t\to t}(X_1yx)(X_2yx)) :\!-\; \mathsf{V}(X_1), \mathsf{V}(X_2).\\
\mathsf{NP}(X_1X_2) :\!-\; \mathsf{Det}(X_1), \mathsf{N}(X_2).\\
\mathsf{NP}(\lambda u.u\, \mathbf{John}^e).\\
\mathsf{V}(\mathbf{find}^{e\to e\to t}).\\
\mathsf{V}(\mathbf{catch}^{e\to e\to t}).\\
\mathsf{V}(=^{e\to e\to t}).\\
\mathsf{Det}(\lambda uv.\exists^{(e\to t)\to t}(\lambda y.\wedge^{t\to t\to t}(uy)(vy))).\\
\mathsf{N}(\mathbf{man}^{e\to t}).\\
\mathsf{N}(\mathbf{unicorn}^{e\to t}).
\end{array} \tag{9}$$

Note that all λ-terms in this grammar are *almost linear* in the sense of satisfying the following conditions:

- every occurrence of λ binds at least one occurrence of a variable (i.e., they are λI terms), and

[8]It is known that the class of string languages generated by linear CFLGs under this encoding coincides with the class of multiple context-free languages [63]. The class of tree languages generated by linear CFLGs has been characterized by Kanazawa [45].

[9]The use of a pair of ACGs with a common abstract vocabulary as a synchronous grammar has already been advocated by de Groote [17].

$$
\begin{array}{l}
\mathsf{S}(\lambda z.Y_1(Y_2 z),\ X_1 X_2) \mathrel{:-} \mathsf{NP}(Y_1, X_1), \mathsf{VP}(Y_2, X_2).\\
\mathsf{VP}(\lambda z.Y_1(Y_2 z),\ \lambda x.X_2(\lambda y.X_1 y x)) \mathrel{:-} \mathsf{V}(Y_1, X_1), \mathsf{NP}(Y_2, X_2).\\
\mathsf{V}(\lambda z.Y_1(\mathsf{/and/}(Y_2 z)),\ \lambda yx.\land^{t\to t\to t}(X_1 y x)(X_2 y x)) \mathrel{:-} \mathsf{V}(Y_1, X_1), \mathsf{V}(Y_2, X_2).\\
\mathsf{NP}(\lambda z.Y_1(Y_2 z),\ X_1 X_2) \mathrel{:-} \mathsf{Det}(Y_1, X_1), \mathsf{N}(Y_2, X_2).\\
\mathsf{NP}(\mathsf{/John/},\ \lambda u.u\, \mathbf{John}^{e}).\\
\mathsf{V}(\mathsf{/found/},\ \mathbf{find}^{e\to e\to t}).\\
\mathsf{V}(\mathsf{/caught/},\ \mathbf{catch}^{e\to e\to t}).\\
\mathsf{V}(\mathsf{/is/},\ =^{e\to e\to t}).\\
\mathsf{Det}(\mathsf{/a/},\ \lambda uv.\exists^{(e\to t)\to t}(\lambda y.\land^{t\to t\to t}(uy)(vy))).\\
\mathsf{N}(\mathsf{/man/},\ \mathbf{man}^{e\to t}).\\
\mathsf{N}(\mathsf{/unicorn/},\ \mathbf{unicorn}^{e\to t}).
\end{array}
$$

$$
\begin{array}{l}
\mathsf{S}(\lambda z.Y_1(Y_2 z)) \mathrel{:-} \mathsf{NP}(Y_1), \mathsf{VP}(Y_2).\\
\mathsf{VP}(\lambda z.Y_1(Y_2 z)) \mathrel{:-} \mathsf{V}(Y_1), \mathsf{NP}(Y_2).\\
\mathsf{V}(\lambda z.Y_1(\mathsf{/and/}(Y_2 z))) \mathrel{:-} \mathsf{V}(Y_1), \mathsf{V}(Y_2).\\
\mathsf{NP}(\lambda z.Y_1(Y_2 z)) \mathrel{:-} \mathsf{Det}(Y_1), \mathsf{N}(Y_2).\\
\mathsf{NP}(\mathsf{/John/}).\\
\mathsf{V}(\mathsf{/found/}).\\
\mathsf{V}(\mathsf{/caught/}).\\
\mathsf{V}(\mathsf{/is/}).\\
\mathsf{Det}(\mathsf{/a/}).\\
\mathsf{N}(\mathsf{/man/}).\\
\mathsf{N}(\mathsf{/unicorn/}).
\end{array}
$$

$$
\begin{array}{l}
\mathsf{S}(X_1 X_2) \mathrel{:-} \mathsf{NP}(X_1), \mathsf{VP}(X_2).\\
\mathsf{VP}(\lambda x.X_2(\lambda y.X_1 y x)) \mathrel{:-} \mathsf{V}(X_1), \mathsf{NP}(X_2).\\
\mathsf{V}(\lambda yx.\land^{t\to t\to t}(X_1 y x)(X_2 y x)) \mathrel{:-} \mathsf{V}(X_1), \mathsf{V}(X_2).\\
\mathsf{NP}(X_1 X_2) \mathrel{:-} \mathsf{Det}(X_1), \mathsf{N}(X_2).\\
\mathsf{NP}(\lambda u.u\, \mathbf{John}^{e}).\\
\mathsf{V}(\mathbf{find}^{e\to e\to t}).\\
\mathsf{V}(\mathbf{catch}^{e\to e\to t}).\\
\mathsf{V}(=^{e\to e\to t}).\\
\mathsf{Det}(\lambda uv.\exists^{(e\to t)\to t}(\lambda y.\land^{t\to t\to t}(uy)(vy))).\\
\mathsf{N}(\mathbf{man}^{e\to t}).\\
\mathsf{N}(\mathbf{unicorn}^{e\to t}).
\end{array}
$$

Figure 4: The grammar in (7) expressed as a synchronous CFLG (top), with its two components separated out. The first component is a linear CFLG encoding the CFG (1), and the second component is the CFLG (9).

- for every subterm N, if a variable x occurs free more than once in N, x has an atomic type,

where the type of an occurrence of a variable is determined by the typing assigned to the λ-term by the grammar. The reduction to Datalog is guaranteed to be correct only when the grammar is *almost linear* in this sense.

The key to our construction is the *principal typing* of an almost linear λ-term. In this informal exposition, we represent principal typings graphically by means of *hypergraphs* of a certain kind. A hypergraph is a generalization of a directed graph where an edge (called a *hyperedge*) may be incident on any number of nodes, depending on its label.[10]

[10] The connection between CFLGs and hypergraphs goes beyond the present informal exposition. See [45] for the relation between linear CFLGs and *hyperedge replacement grammars*, a context-free grammar formalism generating sets of hypergraphs.

For example, take the λ-term

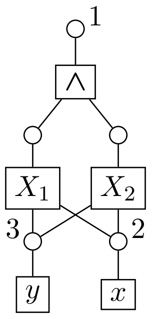
$$\lambda yx.\wedge^{t \to t \to t}(X_1 yx)(X_2 yx) \tag{12}$$

annotating the left-hand side of the third rule of the grammar (9). Recall that the function f mapping nonterminals to their types gives a typing of the λ-term annotating the left-hand side of each rule. The typing assigned to the λ-term (12) is expressed by the typing judgment (10):

$$\vdash X_1 : e \to e \to t, X_2 : e \to e \to t \Rightarrow \lambda yx.\wedge^{t \to t \to t}(X_1 yx)(X_2 yx) : e \to e \to t. \tag{10}$$

(Note that the bound variables x and y both have type e in this typing.) Given the typing judgment (10), we can build the hypergraph for the λ-term (12):

(13)

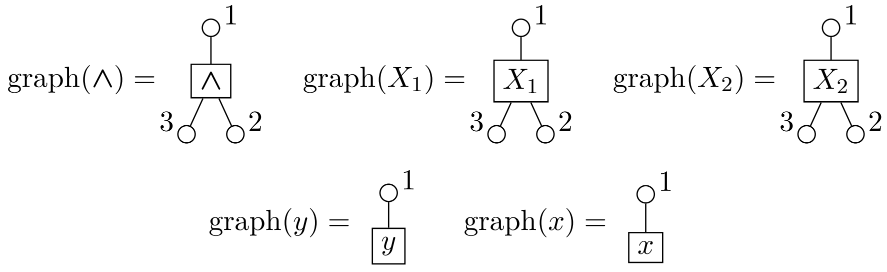


In a diagram like this, circles represent nodes, and circles with numbers attached to them are *external nodes* of the hypergraph. Each hyperedge is represented by a box with a label inside and *tentacles* connecting it to the nodes that it is incident on. The tentacles of a hyperedge are ordered; in this paper, we adopt the convention that they are ordered clockwise starting from the 12 o'clock position. Thus, the hyperedge with label X_2 in (13) has three tentacles, with the first tentacle leading to the node right above it, the second to the node right below it, and the third to the node right below the hyperedge with label X_1. We call the first node in the sequence of nodes that a hyperedge is incident on the *result node* of the hyperedge.

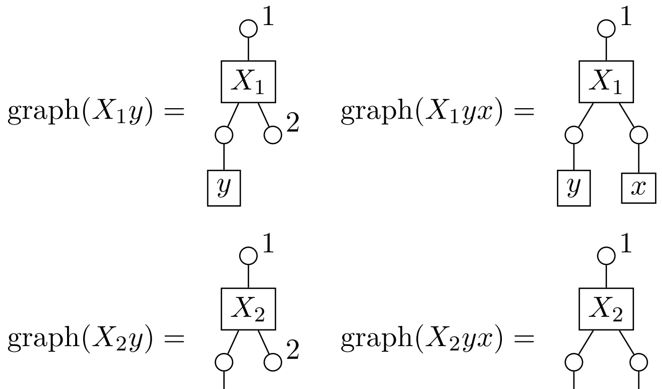
In general, the hypergraph graph(M) for a typed almost linear λ-term M is constructed by induction on the structure of M, as follows. If α is a type, let $|\alpha|$ be the number of occurrences of atomic types in α.[11]

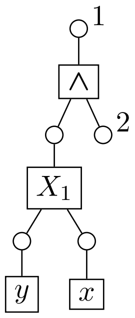
[11] In this paper, we greatly overload the notation $|\cdot|$. In addition to the use just defined, we use it to mean the number of nodes of a tree, the length of a string, and the number of components of a tuple. It should be clear from the context which meaning is intended.

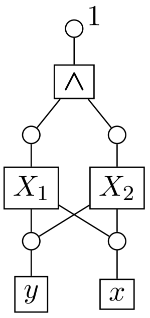
For a variable or a constant a of type α, graph(a) consists of $|\alpha|$ nodes $v_1, \ldots, v_{|\alpha|}$, all of which are external nodes, and a single hyperedge labeled by a, which is incident on $v_1, \ldots, v_{|\alpha|}$, in this order. Given the typing in (10), we have:



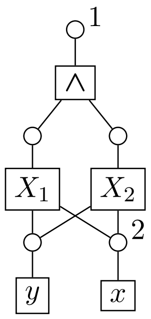
If M is an application M_1M_2, where M_1 and M_2 are of type $\alpha \to \beta$ and α, respectively, graph(M) is constructed from the union of graph(M_1) and graph(M_2) by identifying the last $|\alpha|$ external nodes of graph(M_1) with the external nodes of graph(M_2); the remaining external nodes of graph(M_1) become the external nodes of M. If M_1 and M_2 share a free variable x (which must be of atomic type since M is almost linear), then the x-labeled hyperedge in graph(M_1) and the x-labeled hyperedge in graph(M_2), as well as the nodes that they are incident on, are also identified.

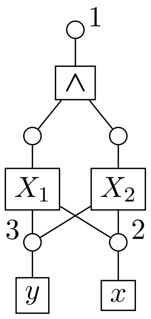


graph($\wedge(X_1yx)$) = 

graph($\wedge(X_1yx)(X_2yx)$) = 

Finally, if M is a λ-abstraction $\lambda x.M_1$, then graph(M) is obtained from graph(M_1) by appending the sequence of nodes that the x-labeled hyperedge is incident on to the sequence of external nodes.

graph($\lambda x.\wedge(X_1yx)(X_2yx)$) = 

graph($\lambda yx.\wedge(X_1yx)(X_2yx)$) =  = (13)

There are several important points to note about this construction:

- If M has type α, graph(M) has $|\alpha|$ external nodes.
- For each free variable x in M, there is exactly one hyperedge labeled by x in graph(M).

- When M is in η-long β-normal form, graph(M) is what is called a *term graph* (see [59]) with external nodes; in particular, for each node v in graph(M), there is exactly one hyperedge whose result node is v.

To convert an almost linear CFLG rule

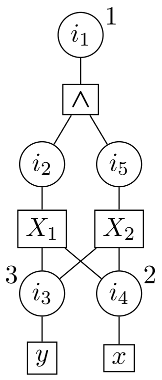
$$B(M) :- B_1(X_1), \dots, B_n(X_n)$$

into a Datalog rule, we take graph(M) and name its nodes with Datalog variables (for which we use $i_1, i_2, i_3, \dots$). In the case of the third rule of the grammar (9),

$$\mathsf{V}(\lambda yx. \wedge^{t \to t \to t}(X_1 yx)(X_2 yx)) :- \mathsf{V}(X_1), \mathsf{V}(X_2), \tag{14}$$

we get:

(15)



Then we do three things to the CFLG rule:

(i) replace the left-hand side λ-term M by the sequence of external nodes of graph(M),

(ii) replace each right-hand side variable X_i by the sequence of nodes that the X_i-labeled hyperedge is incident on in graph(M), and

(iii) for each hyperedge in graph(M) labeled by a constant b, add to the right-hand side of the rule an atom $b(\vec{v})$, where $\vec{v}$ is the sequence of nodes that the hyperedge is incident on.

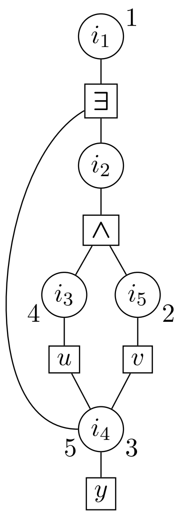
Applying this procedure to (14) produces the following result:

$$\mathsf{V}(i_1, i_4, i_3) :- \wedge(i_1, i_5, i_2), \mathsf{V}(i_2, i_4, i_3), \mathsf{V}(i_5, i_4, i_3).$$

For another example, consider the ninth rule of the CFLG in Figure 9:

$$\mathsf{Det}(\lambda uv. \exists^{(e \to t) \to t}(\lambda y. \wedge^{t \to t \to t}(uy)(vy))).$$

The hypergraph for this λ-term is



and the corresponding Datalog rule is

$$\mathsf{Det}(i_1, i_5, i_4, i_3, i_4) \mathrel{:-} \exists(i_1, i_2, i_4), \wedge(i_2, i_5, i_3).$$

Applying the same procedure to all the rules in (9), we get the following Datalog program:

$$\begin{aligned}
&\mathsf{S}(i_1) \mathrel{:-} \mathsf{NP}(i_1, i_2, i_3), \mathsf{VP}(i_2, i_3).\\
&\mathsf{VP}(i_1, i_4) \mathrel{:-} \mathsf{V}(i_2, i_4, i_3), \mathsf{NP}(i_1, i_2, i_3).\\
&\mathsf{V}(i_1, i_4, i_3) \mathrel{:-} \wedge(i_1, i_5, i_2), \mathsf{V}(i_2, i_4, i_3), \mathsf{V}(i_5, i_4, i_3).\\
&\mathsf{NP}(i_1, i_4, i_5) \mathrel{:-} \mathsf{Det}(i_1, i_4, i_5, i_2, i_3), \mathsf{N}(i_2, i_3).\\
&\mathsf{NP}(i_1, i_1, i_2) \mathrel{:-} \mathbf{John}(i_2).\\
&\mathsf{V}(i_1, i_3, i_2) \mathrel{:-} \mathbf{find}(i_1, i_3, i_2).\\
&\mathsf{V}(i_1, i_3, i_2) \mathrel{:-} \mathbf{catch}(i_1, i_3, i_2).\\
&\mathsf{Det}(i_1, i_5, i_4, i_3, i_4) \mathrel{:-} \exists(i_1, i_2, i_4), \wedge(i_2, i_5, i_3).\\
&\mathsf{N}(i_1, i_2) \mathrel{:-} \mathbf{man}(i_1, i_2).\\
&\mathsf{N}(i_1, i_2) \mathrel{:-} \mathbf{unicorn}(i_1, i_2).
\end{aligned} \tag{16}$$

The construction of the database representing the input λ-term is similar, but slightly more complex. A simple case is the λ-term (8), where each constant occurs just once:

$$\exists(\lambda y. \wedge(\mathbf{unicorn}\, y)(\mathbf{find}\, y\, \mathbf{John})) \tag{8}$$

This is an almost linear λ-term in η-long β-normal form, from which we obtain the

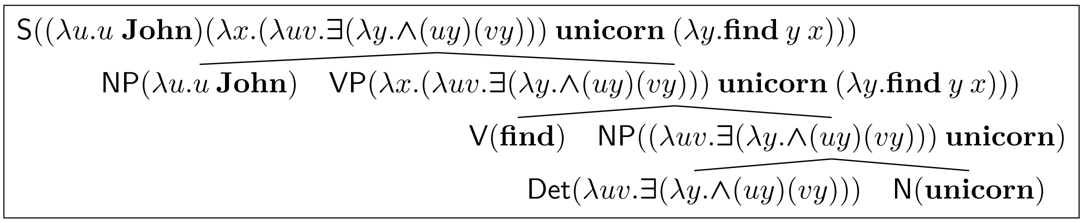
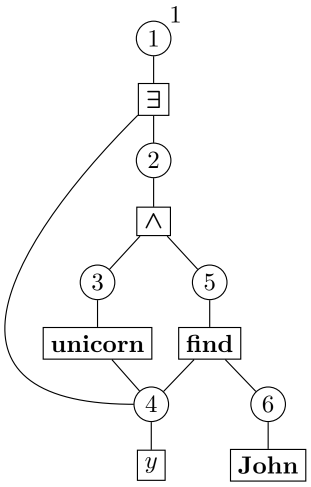

S((λu.u John)(λx.(λuv.∃(λy.∧(uy)(vy))) unicorn (λy.find y x)))
NP(λu.u John)
VP(λx.(λuv.∃(λy.∧(uy)(vy))) unicorn (λy.find y x)))
V(find)
NP((λuv.∃(λy.∧(uy)(vy))) unicorn)
Det(λuv.∃(λy.∧(uy)(vy)))
N(unicorn)

Figure 5: The CFLG derivation tree for (8)

following hypergraph:



The hyperedges of this hypergraph that are labeled by constants in the λ-term constitute the facts in the database representing the λ-term:

$$\exists(1,2,4).\ \ \wedge(2,5,3).\ \ \mathbf{unicorn}(3,4).\ \ \mathbf{find}(5,6,4).\ \ \mathbf{John}(6). \tag{17}$$

(Note that here, we are using database constants $1, 2, 3, \ldots,$ rather than Datalog variables, to name nodes.) The external nodes of the hypergraph (of which there is only one in this example) determine the query:

$$?\!-\ \mathsf{S}(1). \tag{18}$$

The λ-term (8) is in the language of the CFLG (9). Correspondingly, the answer to the query (18) against the program in (16) and the database in (17) is "yes". Figures 5 and 6 show the associated CFLG and Datalog derivation trees.

The situation becomes more complex when the input λ-term contains more than one occurrence of the same constant. Such is the case with the λ-term (19) (this is

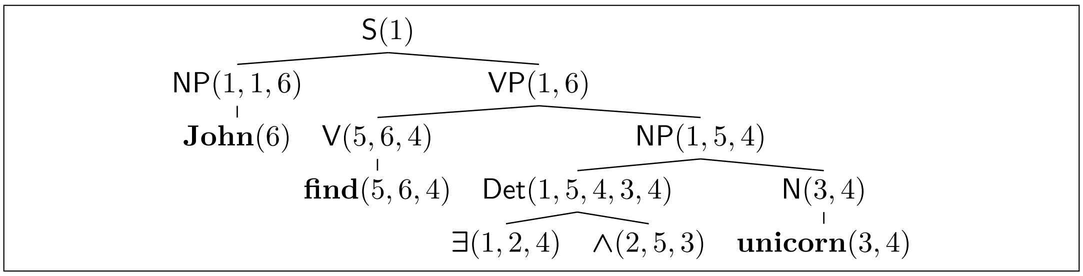


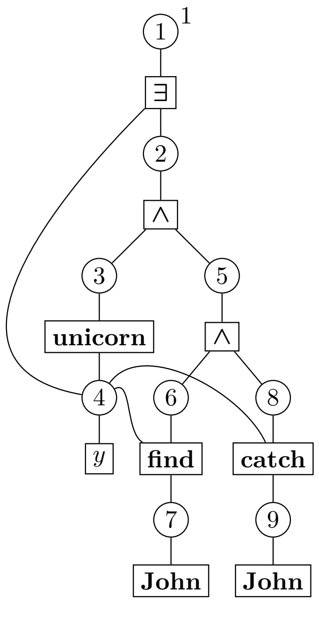
Figure 6: The Datalog derivation tree for the query (18) against the database in (17) and the program in (16).

the λ-term associated with John found and caught a unicorn by the grammar (7)):

$$\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\wedge(\mathbf{find}\, y\, \mathbf{John})(\mathbf{catch}\, y\, \mathbf{John}))). \tag{19}$$

Let us apply the same procedure to (19) as we did to (8). The hypergraph for (19) is the following:

(20)



From this hypergraph, we would get the database (21) and the query (22):

$\exists(1,2,4)$. $\wedge(2,5,3)$. $\mathbf{unicorn}(3,4)$. $\wedge(5,8,6)$. $\mathbf{find}(6,7,4)$. $\mathbf{John}(7)$. $\mathbf{catch}(8,9,4)$. $\mathbf{John}(9)$. (21)

$$?\!-\mathsf{S}(1). \tag{22}$$

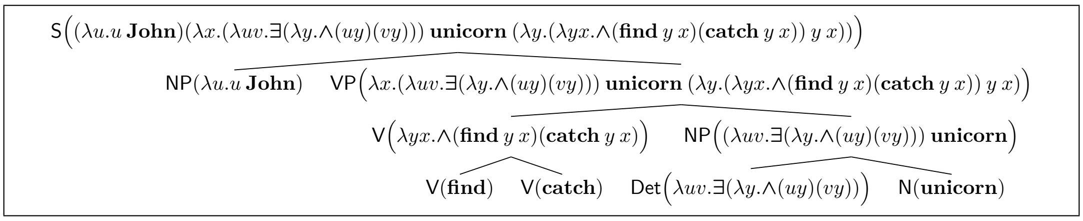
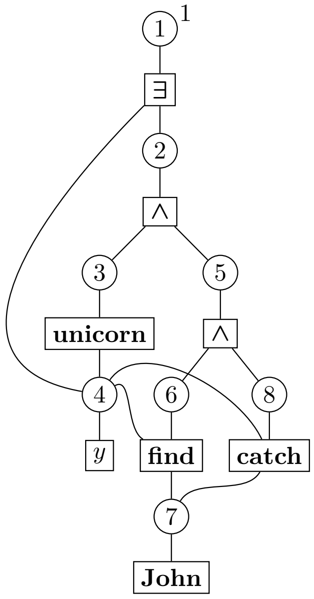

S((λu.u John)(λx.(λuv.∃(λy.∧(uy)(vy))) unicorn (λy.(λyx.∧(find y x)(catch y x)) y x)))
NP(λu.u John)
VP(λx.(λuv.∃(λy.∧(uy)(vy))) unicorn (λy.(λyx.∧(find y x)(catch y x)) y x))
V(λyx.∧(find y x)(catch y x))
NP((λuv.∃(λy.∧(uy)(vy))) unicorn)
V(find)
V(catch)
Det(λuv.∃(λy.∧(uy)(vy)))
N(unicorn)

Figure 7: The CFLG derivation tree for (19).

It turns out, however, that (21) is not the correct database corresponding to the input λ-term (19). Even though (19) is generated by the CFLG in (9) with the derivation tree in Figure 7, the answer to the query (22) against the database (21) and the program (16) is "no", as the reader can easily verify.

To obtain the desired database, we need to modify (20) by identifying the two hyperedges labeled by **John** and the nodes they are incident on, as follows:



(23)

This gives the database (24).

$\exists(1,2,4)$. $\wedge(2,5,3)$. $\wedge(5,8,6)$. **unicron**$(3,4)$. **find**$(6,7,4)$. **John**(7). **catch**$(8,7,4)$. (24)

Against this database and the program in Figure 16, the query (22) is correctly answered "yes". Figure 8 shows the associated Datalog derivation tree for this query.

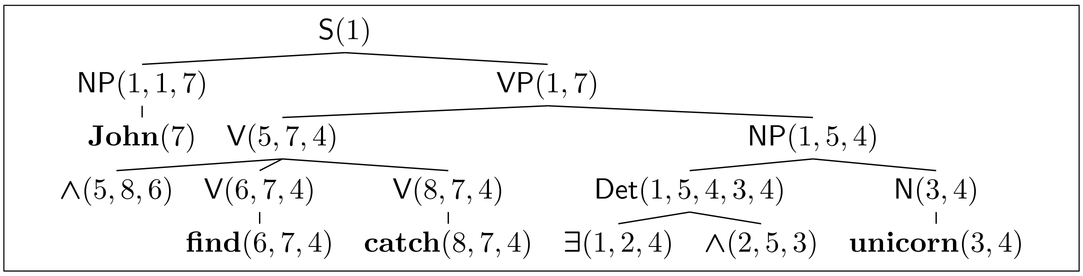


Figure 8: The Datalog derivation tree for the query (22) against the database (24) and the program in (16).

Note that the database (24) can also be obtained from the following non-β-normal λ-term, which β-reduces to (19):

$$\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)((\lambda x.\wedge(\mathbf{find}\, y\, x)(\mathbf{catch}\, y\, x))\, \mathbf{John})). \tag{25}$$

The hypergraph for (25) is identical to (23), except for the presence of an additional hyperedge labeled by x (incident on the node named "7").

The general rule is that the input λ-term should first be β-expanded to an almost linear λ-term that is the most "compact" in the sense of containing the fewest occurrences of constants, before the hypergraph and the associated database and query are extracted out of it. This explains why the two hyperedges labeled by $\wedge$ in (23) cannot be identified, because there is no almost linear λ-term with just one occurrence of $\wedge$ that β-reduces to (19). On the level of hypergraphs, the necessary operation is similar to the conversion of term graphs to their *fully collapsed* form (see [59]). This is by no means an accurate formulation, however, because the "fully collapsed form" does not always correspond to an almost linear λ-term, and there is some subtlety involved in the treatment of hyperedges labeled by bound variables.[12] A precise method of converting the input λ-term N to the desired almost linear λ-term N° will be given by Algorithm 1 in Section 3.7.[13]

Note that the way we obtain a database from an input λ-term generalizes the standard database representation of a string: from the λ-term encoding

[12]For example, the algorithm β-expands $d(b(\lambda u.a(uc)))(b(\lambda v.a(vc)))$ to $(\lambda x.dxx)(b(\lambda u.a(uc)))$, but does not β-expand $d(\lambda u.a(uc))(\lambda u.a(uc))$ to $(\lambda x.dxx)(\lambda u.a(uc))$, which would correspond to the fully collapsed form.

[13]The input λ-terms we have used as examples are both almost linear. Since the class of almost linear λ-terms is not closed under β-reduction, a β-normal λ-term generated by an almost linear CFLG is not necessarily almost linear. Thus, in general, the input λ-term has to be β-expanded to an almost linear λ-term before any hypergraph can be obtained by the method outlined above, even when no constant occurs more than once in the input λ-term.

$/a_1 \dots a_n/ = \lambda z.a_1^{o \to o}(\dots(a_n^{o \to o} z)\dots)$ of a string $a_1 \dots a_n$, we obtain the database $\{a_1(0,1), \dots, a_n(n-1,n)\}$ and the query $?\!-\ \mathsf{S}(0,n)$, as the reader may verify.

2.3 An outline of the proof of correctness

Let us give a rough idea of the proof of correctness of our reduction, presented informally in Section 2.2.

For the reader familiar with the notion of a *principal typing*, it should be clear how the hypergraph graph(M) for an almost linear λ-term M corresponds to a principal (i.e., most general) typing of M, where occurrences of constants are treated like mutually distinct free variables. For instance, corresponding to the hypergraph (20) for the almost linear λ-term (19), we have the principal typing

$$\begin{aligned}&\exists : (4 \to 2) \to 1,\ \wedge_1 : 3 \to 5 \to 2,\ \mathbf{unicorn} : 4 \to 3,\ \wedge_2 : 6 \to 8 \to 5,\\ &\qquad \mathbf{find} : 4 \to 7 \to 6,\ \mathbf{John}_1 : 7,\ \mathbf{catch} : 4 \to 9 \to 8,\ \mathbf{John}_2 : 9\ \Rightarrow\ 1. \end{aligned} \tag{26}$$

Note that distinct occurrences of $\wedge$ and of **John** in (19) are regarded as distinct free variables. In the case of the λ-term (25), which has just one occurrence of **John**, we have

$$\begin{aligned}&\exists : (4 \to 2) \to 1,\ \wedge_1 : 3 \to 5 \to 2,\ \mathbf{unicorn} : 4 \to 3,\ \wedge_2 : 6 \to 8 \to 5,\\ &\qquad \mathbf{find} : 4 \to 7 \to 6,\ \mathbf{John} : 7,\ \mathbf{catch} : 4 \to 7 \to 8\ \Rightarrow\ 1 \end{aligned} \tag{27}$$

as its principal typing, corresponding to (23).[14]

What is special about almost linear λ-terms is that when an almost linear λ-term with constants (in η-long form) is "maximally compact" in the sense that it has no β-equal almost linear λ-term with fewer occurrences of constants, its principal typing exactly characterizes the set of almost linear λ-terms (in η-long form) that are β-equal to it. More precisely, let M be such a maximally compact almost linear λ-term in η-long form and let $\Gamma \Rightarrow \alpha$ be its principal typing. Then we have the following equivalence for every almost linear λ-term M' in η-long form:

$$\begin{aligned}&M' \text{ has a typing } \Gamma' \Rightarrow \alpha \text{ for some subset } \Gamma' \text{ of } \Gamma\\ &\qquad\qquad\text{if and only if } M' \text{ is } \beta\text{-equal to } M.\end{aligned} \tag{28}$$

[14]The exact correspondence between graph(M) and a principal typing of M requires M to be in η-long form. Note that this notion of typing of a λ-term with constants is different from the notion of typing expressed by judgments like (10), where constants have fixed, pre-assigned types. In the rigorous presentation of Section 3, typings like (26) and (27) will be replaced by typings of pure λ-terms that result by replacing distinct occurrences of constants by distinct free variables.

The main ingredients of the proof of this property of almost linear λ-terms are the following:

- A principal typing of an almost linear λ-term is *negatively non-duplicated* in the sense that each atomic type has at most one negative occurrence in it (cf. [2]).
- All λ-terms that share a negatively non-duplicated typing are $\beta\eta$-equal [3]. This is a generalization of the *Coherence Theorem* (see [56]).
- The leftmost β-reduction from an almost linear λ-term is non-erasing and *almost non-duplicating* in the sense that for each β-redex $(\lambda x.P)Q$ that is contracted, x can occur free more than once in P only when the type of x is atomic.
- If there is a non-erasing, almost non-duplicating β-reduction from a pure (i.e., constant-free) λ-term M to N, every typing of N is a typing of M. This is a generalization of the *Subject Exapnsion Theorem* (see [31]).

Now let $\mathbf{P}$ be the Datalog program constructed from the given almost linear CFLG $\mathscr{G}$, and let N be the input λ-term (in η-long β-normal form). Suppose that our algorithm first β-expands N to an almost linear λ-term N°. Let $\Gamma \Rightarrow \alpha$ be a principal typing of N°, and let D and $?\!-\ S(\overline{\alpha})$ be the database and query constructed from this typing.

Suppose that there is a Datalog derivation tree T for the query $?\!-\ S(\overline{\alpha})$ against the program $\mathbf{P}$ and the database D. Given the one-one correspondence between the rules of $\mathscr{G}$ and the rules of $\mathbf{P}$, the Datalog derivation tree T determines a CFLG derivation tree T'. (See Figures 5, 6, 7, 8 for examples.) The former, however, contains more information than the latter. Each ground instance ρ of a Datalog rule used in T corresponds to a typing of the λ-term in the corresponding CFLG rule. For instance, the ground instance

$$\mathsf{V}(5,7,4) :\!-\ \mathsf{\wedge}(5,8,6), \mathsf{V}(6,7,4), \mathsf{V}(8,7,4)$$

of the third rule of (16) that is used in the Datalog derivation tree in Figure 8 gives the following typing judgment:

$$\vdash \wedge : 6 \to 8 \to 5,\ X_1 : 4 \to 7 \to 6,\ X_2 : 4 \to 7 \to 8 \Rightarrow \lambda yx.\wedge(X_1 yx)(X_2 yx) : 4 \to 7 \to 5.$$

Piecing together all these typing judgments corresponding to ground instances of rules used in T gives a typing judgment

$$\vdash \Gamma' \Rightarrow P : \alpha',$$

where P is the (non-β-normal) almost linear λ-term at the root node of T'. Since α' and Γ' correspond to the root node and the leaf nodes of T, respectively, we must have $\alpha' = \alpha$ and $\Gamma' \subseteq \Gamma$. By the special property (28) of almost linear λ-terms, it follows that P is $\beta\eta$-equal to N° and hence to N, which implies that T' is a derivation tree for N.

Let us now consider the converse direction and suppose that a derivation tree T' of $\mathscr{G}$ has its root node labeled by $S(P)$ and P β-reduces to N. By the one-one correspondence between the rules of $\mathscr{G}$ and the rules of $\mathbf{P}$, T' determines a "skeletal" Datalog derivation tree made up of non-ground instances of rules of $\mathbf{P}$, where predicates have Datalog variables as arguments, instead of database constants. The question is whether one can replace these Datalog variables with database constants from D in such a way that leaf nodes will correspond to facts in D, so that the derivation tree will become a derivation tree for $S(\overline{\alpha})$ against $\mathbf{P}$ and D. This is possible precisely when P has a typing $\Gamma' \Rightarrow \alpha$ with $\Gamma' \subseteq \Gamma$. By the special property (28) again, this must be so since P is almost linear and is $\beta\eta$-equal to N and hence to N°.

2.4 The scope of the present method

The present method of reduction to Datalog is directly applicable only to formalisms expressible in almost linear CFLGs. Almost linear λ-terms suffice to represent formulas in a logical language with quantification over individual variables only, so when the meaning representation language used in a surface realization problem is such a language, the input to the corresponding CFLG recognition problem will always be an almost linear λ-term. For instance, in the extensional subfragment of Montague's [57] fragment of English, the translations of English sentences will fall within such a language. Consequently, it is possible to extend the grammar (7) to one that covers a large portion of Montague's [57] fragment while keeping the semantic half of the grammar almost linear. However, even when almost linear λ-terms suffice to encode the target logical forms, we sometimes need grammar rules that are not almost linear.[15]

For example, suppose we add to the synchronous grammar in Figure 4 the following rules:

$\mathsf{NP}(\lambda z.Y_1(/\mathsf{and}/(Y_2 z)),\ \lambda u.\wedge^{t\to t\to t}(X_1(\lambda x.ux))(X_2(\lambda x.ux)))$:− $\mathsf{NP}(Y_1, X_1), \mathsf{NP}(Y_2, X_2)$.
$\mathsf{VP}(/\mathsf{sang}/,\ \mathbf{sing}^{e\to t})$.
$\mathsf{NP}(/\mathsf{Bill}/,\ \lambda u.u\,\mathbf{Bill}^{e})$.

[15]This is already evidenced in the grammar of Montague [57], which has a rule similar to the first of the three rules below.

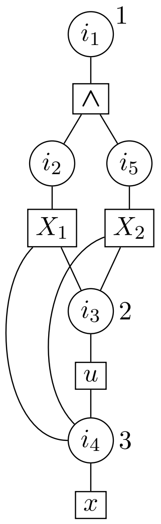
With these rules, the grammar can now generate John and Bill sang, with the logical form

$$\wedge(\textbf{sing John})(\textbf{sing Bill}). \tag{29}$$

Let us see how we might convert to Datalog the "semantic half" of the three synchronous rules above:

$$\begin{aligned}&\mathsf{NP}(\lambda u.\wedge^{t\to t\to t}(X_1(\lambda x.ux))(X_2(\lambda x.ux))) \coloneq \mathsf{NP}(X_1), \mathsf{NP}(X_2).\\ &\mathsf{VP}(\textbf{sing}^{e\to t}).\\ &\mathsf{NP}(\lambda u.u\,\textbf{Bill}^{e}).\end{aligned} \tag{30}$$

Recall that $f(\mathsf{NP}) = (e \to t) \to t$, so the type of the variables X_1 and X_2 in the first rule of (30) are $(e \to t) \to t$ and the type of u is $e \to t$. This means that the λ-term M on the left-hand side of this rule is not almost linear. The method we described was not meant to apply to a case like this, but suppose we extend it to cover this case. We would get the following hypergraph.[16]

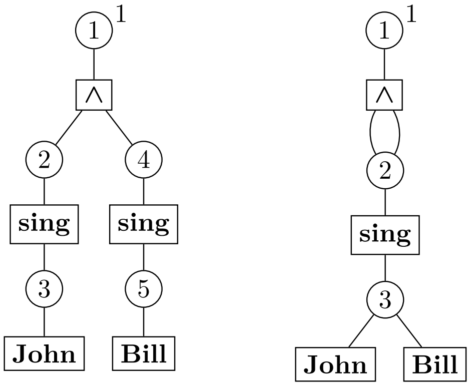


Thus, from the three CFLG rules in (30), we get the following Datalog rules:

$$\begin{aligned}&\mathsf{NP}(i_1, i_3, i_4) \coloneq \wedge(i_1, i_5, i_2), \mathsf{NP}(i_2, i_3, i_4), \mathsf{NP}(i_5, i_3, i_4).\\ &\mathsf{VP}(i_1, i_2) \coloneq \textbf{sing}(i_1, i_2).\\ &\mathsf{NP}(i_1, i_1, i_2) \coloneq \textbf{Bill}(i_2).\end{aligned} \tag{31}$$

[16]This graph corresponds to the principal typing of the λ-term M.

As for the λ-term (29), there are two conceivable hypergraphs that can be associated with it:



The first graph is what we obtain with the method described above. The second graph is the result of identifying the two edges labeled by **sing** and the nodes they are incident on. The corresponding databases are:

$$\wedge(1,4,2).\;\; \mathbf{sing}(2,3).\;\; \mathbf{sing}(4,5).\;\; \mathbf{John}(3).\;\; \mathbf{Bill}(5). \tag{32}$$

$$\wedge(1,2,2).\;\; \mathbf{sing}(2,3).\;\; \mathbf{John}(3).\;\; \mathbf{Bill}(3). \tag{33}$$

Against the database (32) and the program consisting of the rules in (16) and (31), the query

$$?\!-\, \mathsf{S}(1).$$

is answered "no". Against the database (33) and the same program, the same query is answered "yes", but there are too many Datalog derivation trees for this query. In addition to the correct one corresponding to the CFLG derivation tree for (29), there are three others, corresponding to the CFLG derivation trees for the following λ-terms:

$$\wedge(\mathbf{sing\,John})(\mathbf{sing\,John})$$
$$\wedge(\mathbf{sing\,Bill})(\mathbf{sing\,John})$$
$$\wedge(\mathbf{sing\,Bill})(\mathbf{sing\,Bill})$$

This means that if (33) is used to solve the task of finding sentences expressing meaning (29), the output obtained contains not just John and Bill sang, but also John and John sang, Bill and John sang, and Bill and Bill sang. Thus, neither (32) nor (33) gives a correct reduction of surface realization to Datalog query evaluation.

As for applications to parsing and recognition, the present method directly applies to string-generating grammars with no copying operation, like multiple context-free grammars, but not to formalisms like macro grammars [24] and parallel multiple context-free grammars [66], where derivations involve copying of strings. To represent grammar rules that duplicate strings, a CFLG must use multiple occurrences of the same variable of type $o \to o$, and so cannot be almost linear. An almost linear CFLG can represent tree grammars with copying operations, however, because trees are represented by λ-terms of atomic type o. It turns out that this provides an indirect way of applying the present method to grammars with string copying, using as input a representation of a finite set of trees that yield a given input string. This point will be elaborated in Section 4.2.

2.5 The present approach to generation

In this section, I clarify some basic assumptions I make in this work about the meaning representation language and the task of surface realization. These assumptions do not concern the formal result about the reduction of almost linear CFLGs to Datalog, but rather the kind of application of the formal result to grammars for natural language I have in mind.

In Montague's [57] work, the meaning representation language, which incorporates a form of λ-calculus, is just a convenient tool used to give a model-theoretic semantics to the object language, and can in principle be dispensed with. In contrast, this work assumes that the level of semantic representation is crucial and that grammar rules specifically refer to λ-terms as structured, "syntactic" objects. Any computation on meanings must be performed on some form of representation or other; using λ-terms as semantic representations seems to be a convenient choice.

The formalism of λ-calculus can be used in different ways for different purposes. The example grammar I have given uses λ-terms to more or less directly represent formulas of the language of some logic (subsuming at least first-order logic), using appropriately typed constants for logical and non-logical symbols of the language. Binding of a variable by a quantifier is represented by an application of the constant representing the quantifier to a λ-abstract.[17] A pleasant consequence of this is that two formulas that are related by renaming of bound variables translate into α-equivalent λ-terms and are treated as the same. However, since constants are just uninterpreted symbols, all other cases of logically equivalent pairs of formulas come out as distinct λ-terms.

This use of λ-calculus, as an alternative syntax for the language of some logic, is of course not the only way to use λ-calculus as a meaning representation language. For example, logical connectives and quantifiers may be defined in terms of equality (at different types), à la Henkin [29].[18] It is also common to represent truth values,

[17]Following Church [14], Barwise and Cooper [7], and Lloyd [53], among many others.

[18]For example, the universal quantifier (over individuals) may be defined as $\forall^{(e \to t) \to t} =$

Boolean functions, etc., with pure (i.e., constant-free) λ-terms, using $\tau \to \tau \to \tau$ as the type of truth values.[19] One can even represent finite models as λ-terms and cast sentence meanings as functions from finite models to truth values [30]. These more sophisticated uses of λ-calculus, however, almost always take us outside of the realm of almost linear λ-terms, so the main result of this paper will not be applicable.[20]

The main result of this paper applies to surface realization as understood to be the problem of finding a sentence such that the logical form associated with it by the grammar exactly matches the input logical form. This means that the question of whether or not the input logical form is surface realizable depends on the exact shape of the input. If we take our example grammar (7), the answer is different for each of the following pairs:

(34) a. $\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\mathbf{find}\, y\, \mathbf{John}))$
b. $\exists(\lambda y.\wedge(\mathbf{find}\, y\, \mathbf{John})(\mathbf{unicorn}\, y))$

(35) a. $\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\wedge(\mathbf{find}\, y\, \mathbf{John})(\mathbf{catch}\, y\, \mathbf{John})))$
b. $\exists(\lambda y.\wedge(\wedge(\mathbf{unicorn}\, y)(\mathbf{find}\, y\, \mathbf{John}))(\mathbf{catch}\, y\, \mathbf{John}))$

(36) a. $\exists(\lambda x.\wedge(\mathbf{man}\, x)(\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\mathbf{find}\, y\ x))))$
b. $\exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\exists(\lambda x.\wedge(\mathbf{man}\, x)(\mathbf{find}\, y\ x))))$

(37) a. $\exists(\lambda y.\wedge(\mathbf{man}\, y)(=\ y\, \mathbf{John}))$
b. $\mathbf{man}\, \mathbf{John}$

It is generally agreed in computational linguistics that the input to surface realization should not be informed by the particularities of the grammar and that ideally, both members of these pairs should lead to the same result, since they are obviously logically equivalent [68]. While accounting for the full range of logical equivalence is clearly intractable, capturing commutativity and associativity of conjunction is considered particularly important in machine translation applications, and partly for this reason it is popular in computational linguistics to use a "flat" and "unordered" meaning representation language where equivalences like (34) and (35) are built in (see, e.g., [15] or [51]). Another motivation for flat semantics is the need for compact "underspecified" representation of a range of different scope readings of sentences with multiple scope-taking operators. Generation from flat semantics

$\lambda u^{e\to t}.\ =^{(e\to t)\to(e\to t)\to t}\ u\,(\lambda x^{e}.\ =^{e\to e\to t}\ x\,x)$

[19]The truth values "true" and "false" are encoded by the λ-terms $\lambda xy.x$ and $\lambda xy.y$, respectively. These are known as *Church Booleans*.

[20]Surface realization in such a setting is still decidable since Salvati [65] proves that recognition is decidable for general CFLGs. It is an open question how far the class of almost linear λ-terms can be extended without making the resulting CFLG recognition problem intractable.

has been shown to be NP-hard [50], however, so adopting a flat representation language is (for all we know) incompatible with polynomial-time algorithms for surface realization.

Typed λ-terms, with "hierarchical" and "ordered" structures, do not seem to be particularly well suited to encoding of flat semantics, but it is possible to adapt to λ-calculus the idea of Koller et al. [49], who have proposed to use regular tree grammars that generate finite sets of trees as a formalism for underspecification. Trees cannot properly represent variable binding, so a reasonably compact description of a "regular" set of λ-terms will improve upon Koller et al.'s [49] proposal.[21] It turns out that in certain cases, a *database* serves as such a compact representation. In Section 4.2, I present a result extending the main result to handle certain regular sets of λ-terms as input to the recognition problem for almost linear context-free λ-term grammars. Notwithstanding this possibility of accommodating underspecification, I believe that thorough understanding of the simpler problem of "exact" surface realization should take precedence.

The underlying theme of this work is that the problem of surface realization can and should be studied in the style of formal language theory, just like parsing. For this purpose, the problem of surface realization should be formulated in abstract, general terms. The primary goals of any such study would be to identify the computational complexity class for which the problem is complete, and to provide natural, efficient algorithms (insofar as is allowed by the complexity lower bound) to solve the problem. The formalism in which the input to surface realization is encoded should be sufficiently rich to support constructs (e.g., variable binding) that are necessary to express natural language meaning, but should not be tied to one particular logical language. Typed λ-calculus seems to fit this role very well; it has a wide variety of uses, its formal properties have been extensively studied, and its use is also fairly common in computational linguistics. All other things being equal, a general, mathematically elegant, and well-understood formalism should be preferred over ad hoc, application-specific, ill-understood alternatives.

3 Formal development

3.1 Preliminaries

3.1.1 Datalog

A *database schema* is a pair $\mathcal{D} = (R, U)$, where R is a finite set of predicates, each of fixed arity, and U is a (possibly infinite) set of database constants. A *ground fact*

[21] See [64] for a definition of a regular or *recognizable* set of typed λ-terms.

over $\mathcal{D}$ is

$$p(\vec{s}),$$

where p is a predicate in R of arity k, and $\vec{s}$ is a k-tuple of constants in U, for some k. A *database* over $\mathcal{D}$ is a finite set of ground facts over $\mathcal{D}$. If D is a database, the *universe* of D, written U_D, is the finite set of constants appearing in D.

We assume that we are given a countably infinite supply of variables. A *Datalog program* over R is a finite set of *rules*, which are function-free definite clauses of the form

$$p_0(\vec{x}_0) :\!-\ p_1(\vec{x}_1), \dots, p_n(\vec{x}_n),$$

where $n \geq 0$, p_i are predicates, each of fixed arity, and $\vec{x}_i$ are tuples of variables (not necessarily distinct) of appropriate length, matching the predicate's arity. A predicate together with its arguments constitutes an *atom*. The left-hand side of a rule (the part to the left of :−) is called the *head*, and the right-hand side the *body*. The atoms that constitute the body are the *subgoals* of the rule.[22] The predicates in a program $\mathbf{P}$ are divided into the *intensional* predicates and the *extensional* predicates. A predicate is an intensional predicate if it appears in the head of some rule, and an extensional predicate otherwise. An *extensional database* for $\mathbf{P}$ is a database D for a schema $\mathcal{D} = (R, U)$ for some U, where R consists of the extensional predicates of $\mathbf{P}$. We call ground facts in an extensional database *extensional facts*. We follow the logic programming parlance and call a negative Horn clause a *query*.[23] In this paper, we are mainly interested in simple (i.e., non-conjunctive) ground queries of the form

$$?\!-\ p(\vec{s}),$$

where $\vec{s}$ is a tuple of constants from U_D (of appropriate length).

Given a Datalog program $\mathbf{P}$ and an extensional database D, a ground fact $p(\vec{s})$ is *derivable* from $\mathbf{P}$ and D, written

$$\mathbf{P} \cup D \vdash p(\vec{s}),$$

if and only if either $p(\vec{s}) \in D$ or there is a ground instance

$$p(\vec{s}) :\!-\ p_1(\vec{s}_1), \dots, p_n(\vec{s}_n)$$

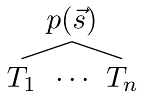
[22]In Datalog, it is often required that the variables in the head of a rule all appear in the body, but we do not assume this restriction. In particular, we allow rules with empty body (i.e., facts) in Datalog programs.

[23]In relational database theory and finite model theory, the term *query* sometimes means a function that maps a finite relational structure to a finite relational structure. A query in this sense may be expressed by a pair $(\mathbf{P}, R')$ consisting of a Datalog program $\mathbf{P}$ and a subset R' of its intensional predicates [16]. See [1] for a similar use of the term "datalog query". The logic programming parlance was used by Ullman [77] in the context of Datalog query evaluation.

of a rule in $\mathbf{P}$ such that

$$\mathbf{P} \cup D \vdash p_i(\vec{s}_i)$$

for each $i = 1, \dots, n$. A *derivation tree* is a tree whose nodes are labeled by ground facts in accordance with the above inductive definition. That is to say, a derivation tree for $p(\vec{s})$ from $\mathbf{P}$ and D is either a tree with a single node labeled by an extensional fact $p(\vec{s}) \in D$, or a tree of the form



where there exists some ground instance $p(\vec{s}) \mathrel{:-} p_1(\vec{s}_1), \dots, p_n(\vec{s}_n)$ of a rule in $\mathbf{P}$ and T_i is a derivation tree for $p_i(\vec{s}_i)$ for $i = 1, \dots, n$.

It is easy to see that for a fixed Datalog program $\mathbf{P}$, the problem of determining, given a database D and a fact q, whether $\mathbf{P} \cup D \vdash q$ holds can be solved in polynomial time in the size of (D, q). For some Datalog program, this problem is known to be P-complete (see [48] for an overview of complexity issues). Among the most basic polynomial-time algorithms for this problem are *naive* and *seminaive* bottom-up evaluation (see [1] or [76]). In these methods, derived facts that share the same predicate are grouped together into a relation, and relational algebra operations are used to expedite the iterative, bottom-up computation of the relations. In the application of Datalog to recognition and parsing, however, the number of derivable facts is usually not large, so it is not so unreasonable to process one fact at a time. Under this simplification, seminaive bottom-up evaluation can be expressed by the following pseudocode. If π is a rule, we write ground(π, U) to denote the set of ground instances of π using only constants from U.

```
1: procedure SEMINAIVE($\mathbf{P}, D$)
2:     $D^0 \leftarrow \varnothing$
3:     $D^1 \leftarrow D \cup \{\, p(\vec{s}) \mid p(\vec{s}) \in \mathrm{ground}(\pi, U_D) \text{ for some } \pi \in \mathbf{P} \,\}$
4:     $\Delta^1 \leftarrow D^1$
5:     $i \leftarrow 1$
6:     while $\Delta^i \neq \varnothing$ do
7:         $\Delta^{i+1} \leftarrow \left\{ p(\vec{s}) \;\middle|\; \begin{array}{l} \pi = p(\vec{x}) \mathrel{:-} p_1(\vec{x}_1), \dots, p_n(\vec{x}_n) \in \mathbf{P}, \\ p_1(\vec{s}_1), \dots, p_{j-1}(\vec{s}_{j-1}) \in D^i,\ p_j(\vec{s}_j) \in \Delta^i, \\ p_{j+1}(\vec{s}_{j+1}), \dots, p_n(\vec{s}_n) \in D^{i-1} \text{ for some } j \in [1, n], \\ \text{and } p(\vec{s}) \mathrel{:-} p_1(\vec{s}_1), \dots, p_n(\vec{s}_n) \in \mathrm{ground}(\pi, U_D) \end{array} \right\} - D^i$
8:         $D^{i+1} \leftarrow D^i \cup \Delta^{i+1}$
9:         $i \leftarrow i + 1$
10:    end while
11:    return $D^i$
```

```
12: end procedure
```

In this algorithm, D^i is the set of ground facts whose derivation trees have minimal height $i-1$.

Derivation trees are assembled from ground instances of rules. If, in addition to derived ground facts, we record ground instances of rules used to derive facts, we can obtain a packed representation of all derivation trees for ground facts derivable from the given program and the input database:[24]

```
 1: procedure SEMINAIVE-PARSE($\mathbf{P}, D$)
 2:     $D^0 \leftarrow \varnothing$
 3:     $D^1 \leftarrow D \cup \{\, p(\vec{s}) \mid p(\vec{s}) \in \text{ground}(\pi, U_D) \text{ for some } \pi \in \mathbf{P} \,\}$
 4:     $G^1 \leftarrow D^1$
 5:     $\Delta^1 \leftarrow D^1$
 6:     $i \leftarrow 1$
 7:     while $\Delta^i \neq \varnothing$ do
 8:         $\Delta^{i+1} \leftarrow \left\{ p(\vec{s}) \;\middle|\; \begin{array}{l} \pi = p(\vec{x}) \mathrel{:-} p_1(\vec{x}_1), \dots, p_n(\vec{x}_n) \in \mathbf{P}, \\ p_1(\vec{s}_1), \dots, p_{j-1}(\vec{s}_{j-1}) \in D^i,\ p_j(\vec{s}_j) \in \Delta^i, \\ p_{j+1}(\vec{s}_{j+1}), \dots, p_n(\vec{s}_n) \in D^{i-1} \text{ for some } j \in [1,n], \\ \text{and } p(\vec{s}) \mathrel{:-} p_1(\vec{s}_1), \dots, p_n(\vec{s}_n) \in \text{ground}(\pi, U_D) \end{array} \right\} - D^i$
 9:         $G^{i+1} \leftarrow \left\{ \pi' \;\middle|\; \begin{array}{l} \pi = p(\vec{x}) \mathrel{:-} p_1(\vec{x}_1), \dots, p_n(\vec{x}_n) \in \mathbf{P}, \\ p_1(\vec{s}_1), \dots, p_{j-1}(\vec{s}_{j-1}) \in D^i,\ p_j(\vec{s}_j) \in \Delta^i, \\ p_{j+1}(\vec{s}_{j+1}), \dots, p_n(\vec{s}_n) \in D^{i-1} \text{ for some } j \in [1,n], \\ \text{and } \pi' = p(\vec{s}) \mathrel{:-} p_1(\vec{s}_1), \dots, p_n(\vec{s}_n) \in \text{ground}(\pi, U_D) \end{array} \right\} \cup G^i$
10:         $D^{i+1} \leftarrow D^i \cup \Delta^{i+1}$
11:         $i \leftarrow i + 1$
12:     end while
13:     return $G^i$
14: end procedure
```

In the implementation of SEMINAIVE-PARSE, the operations in lines 8 and 9 should be performed simultaneously. In this algorithm, the final value of G^i records all rule instances whose subgoals are derivable facts, and constitutes a *propositional Horn clause program.*[25]

There is a natural way to associate an *alternating Turing machine* operating in logarithmic space with each Datalog program [67, 48], and this is useful for the complexity analysis of Datalog programs. Alternating Turing machines (ATMs) [13] are a generalization of non-deterministic Turing machines. The set of states of

[24]The algorithms SEMINAIVE and SEMINAIVE-PARSE can also be written in the style of *chart parsing* [69, 71]. The set Δ^i will correspond to the agenda. See Section 4.3 below.

[25]At the end of the execution of SEMINAIVE-PARSE, we have $D^i = D^{i-1}$, but not necessarily $G^i = G^{i-1}$; it would require one more iteration for G^i to stabilize.

an ATM is partitioned into *existential* and *universal* sates. If a configuration is in an existential state, at least one of the successor configurations must lead to acceptance, whereas if a configuration is in a universal sate, all of its successor configurations must lead to acceptance. A *computation tree* of an ATM $\mathscr{M}$ is a finite rooted directed tree whose nodes are configurations of $\mathscr{M}$ such that the root node is an initial configuration, each existential configuration has just one of its successor configurations as its child, and each universal configuration has all of its successor configurations as its children. An *accepting* computation tree is a computation tree whose leaves are all accepting configurations. An ATM $\mathscr{M}$ operates (simultaneously) in space $S(n)$ and tree size $Z(n)$ if on each input x of length n accepted by $\mathscr{M}$, there is an accepting computation tree of size at most $Z(n)$ in which each configuration uses at most space $S(n)$. Ruzzo [61] characterizes the complexity class LOGCFL as the class of problems for which there is an ATM operating in logarithmic space and in polynomial tree size.

A log-space-bounded ATM $\mathscr{M}_{\mathbf{P}}$ simulating a Datalog program $\mathbf{P}$ may behave as follows. The input to $\mathscr{M}_{\mathbf{P}}$ is a pair (D, q) of an extensional database D for $\mathbf{P}$ and a ground fact q; $\mathscr{M}_{\mathbf{P}}$ accepts (D, q) if and only if $\mathbf{P} \cup D \vdash q$. This ATM uses $k+1$ work tapes, where k is at least as large as the maximal arity of the predicates in $\mathbf{P}$ and the maximal number of variables in rules of $\mathbf{P}$. Each of the first k work tapes serves as a pointer to a position on the input tape where an occurrence of a constant starts. The last work tape is used to check identity of two occurrences of constants (which we assume to be coded as binary strings). Part of $\mathscr{M}_{\mathbf{P}}$'s finite control is used to store a predicate or a rule in $\mathbf{P}$. We call the combination of this part of the finite control and the first k work tapes the "storage area". The storage area of $\mathscr{M}_{\mathbf{P}}$ either stores a ground fact $p(\vec{s})$, using the work tapes to store the sequence $\vec{s}$ of constants, or a ground instance of a rule $\pi = p(\vec{x}) :\!-\; p_1(\vec{x}_1), \dots, p_n(\vec{x}_n)$, using the work tapes to store a ground substitution for the variables in π. The machine starts by copying the ground fact q on the input tape onto its storage area. Whenever $\mathscr{M}_{\mathbf{P}}$ has a ground fact q' in the storage area, it tries to verify $\mathbf{P} \cup D \vdash q'$. If q' is an extensional fact, it verifies that q' appears in the database on the input tape and accepts. If q' is an intensional fact, the machine uses existential branching and guesses a ground instance $\pi\theta$ of a rule $\pi = p(\vec{x}) :\!-\; p_1(\vec{x}_1), \dots, p_n(\vec{x}_n)$ in $\mathbf{P}$ whose head matches q', and places $\pi\theta$ in the storage area. The machine then uses universal branching and for all $i = 1, \dots, n$, places $p_i(\vec{x}_i)\theta$ in the storage area, and repeats the procedure. It should be clear that if there is a derivation tree T for $\mathbf{P} \cup D \vdash q$, then the ATM $\mathscr{M}_{\mathbf{P}}$ on input (D, q) has an accepting computation tree of size $|T| \cdot O(f(n))$, where $|T|$ is the size of T, f is a polynomial, and n is the size of the input (D, q).

Lemma 3.1. *Let* $\mathbf{P}$ *be a Datalog program and* $g(n)$ *be a polynomial. The following* *problem is in LOGCFL:*

$$\{\, (D, q, 1^m) \mid \textit{there is a derivation tree for } \mathbf{P} \cup D \vdash q \textit{ of size} \leq g(m) \,\}$$

Proof. The idea is from [26]. We modify $\mathscr{M}_{\mathbf{P}}$ by including bounds on the size of Datalog derivation trees in each configuration. The modified ATM starts by computing $g(m)$. This computation and the storage of the resulting value (in binary) can both be done within logarithmic space. When the machine is in a configuration storing an extensional fact q' and a bound b (a natural number in binary), it checks that q' appears in D and $b \geq 1$, and accepts. When the machine is in a configuration storing an intensional fact $p(\vec{s})$ and a bound b, it checks that $b > 1$ and guesses a ground instance $p(\vec{s})$:− $p_1(\vec{s}_1), \dots, p_n(\vec{s}_n)$ of some rule, together with bounds $b_1, \dots, b_n$ on the size of the derivations trees for $p_1(\vec{s}), \dots, p_n(\vec{s}_n)$, such that $b_1 + \cdots + b_n = b - 1$. It then uses universal branching to write $p_i(\vec{s}_i)$ and b_i in the storage area and try to find a derivation tree for $p_i(\vec{s}_i)$ of size $\leq b_i$. It is clear that the size of any accepting computation tree of this ATM on input of size n is bounded by some polynomial in n. □

We call a node in a derivation tree an *extensional node* if it is labeled by an extensional fact (i.e., facts from the database), and an *intensional node* otherwise. A derivation tree is called *tight* [79] if no fact occurs more than once on any of its paths. Note that whenever T is a derivation tree for $\mathbf{P} \cup D \vdash p(\vec{s})$ that is not tight, one can turn T into a tight derivation tree for $\mathbf{P} \cup D \vdash p(\vec{s})$ by deleting some nodes from T.

The following elementary lemma will be useful later.

Lemma 3.2. *Let* $\mathbf{P}$ *be a Datalog program. Then there is a polynomial* $g(n)$ *such that whenever there is a derivation tree for* $\mathbf{P} \cup D \vdash p(\vec{s})$ *with* l *extensional nodes, there is a derivation tree* $\mathbf{P} \cup D \vdash p(\vec{s})$ *with* $n \leq l$ *extensional nodes whose size does not exceed* $g(n)$.

Proof. Let k be the number of intensional predicates in $\mathbf{P}$, r be the maximal arity of intensional predicates in $\mathbf{P}$, and m be the maximal number of subgoals of rules in $\mathbf{P}$.

If p is an extensional predicate, $p(\vec{s})$ must be in D and there is a one-node derivation tree for $\mathbf{P} \cup D \vdash p(\vec{s})$. In the following, we assume that p is an intensional predicate.

We first show that there is a constant c (depending on $\mathbf{P}$) such that if $\mathbf{P} \vdash p(\vec{s})$, then there is a derivation tree for $p(\vec{s})$ with at most c nodes. (Note that $\mathbf{P} \vdash p(\vec{s})$ means that $p(\vec{s})$ is derivable without using any extensional facts.) Let T be a smallest derivation tree for $\mathbf{P} \vdash p(\vec{s})$. Without loss of generality, we can assume that all constants that appear in T appear in $p(\vec{s})$, so that there are at most r of them. This is because if T contains other constants, they can be safely replaced by constants in $\vec{s}$. Since T must be a tight derivation tree, the height of T is bounded by $kr^r - 1$. Therefore, the size of T is bounded by m^{kr^r} (if $m \geq 2$) or kr^r (if $m \leq 1$).

Now suppose $\mathbf{P} \cup D \vdash p(\vec{s})$ and let T be a smallest derivation tree for $\mathbf{P} \cup D \vdash p(\vec{s})$ with $n \leq l$ extensional nodes. As before, we can assume without loss of generality that all constants in T occur in $p(\vec{s})$ or in facts labeling extensional nodes, so that there are at most $(n+1)r$ of them. The intensional nodes of T may be divided into the following three types:

Type 0 Intensional nodes that are not ancestors of any extensional nodes.

Type 1 Intensional nodes that have just one child that is an ancestor of some extensional node.

Type 2 Intensional nodes that have two or more children that are ancestors of extensional nodes.

Since the case of $n = 0$ has already been taken care of, assume $n \geq 1$. It is easy to see that the number of intensional nodes of type 2 is at most $n - 1$.

To find a bound on the number of type 1 nodes, note first that all children of type 1 nodes are type 0 nodes, except one, which is either an extensional node, a type 1 node, or a type 2 node. We call two type 1 nodes *equivalent* if they are related by the smallest equivalence relation extending the child-of relation restricted to type 1 nodes. Each equivalence class of type 1 nodes is linearly ordered by the child-of relation, and its minimal element is the parent of an extensional node or of a type 2 node. Since T must be tight by the minimality of T, the size of each equivalence class of type 1 nodes cannot exceed $k((n+1)r)^r$. Since there are at most $2n - 1$ equivalence classes of type 1 nodes, the number of type 1 nodes is bounded by $(2n-1)k((n+1)r)^r$.

We finally turn to type 0 nodes. Note that all children of type 0 nodes are type 0 nodes. We call a type 0 node *maximal* if it is not a child of a type 0 node. Since we are assuming $n \geq 1$, any maximal type 0 node has a parent, which is either a type 1 node or a type 2 node. This implies that either there is no type 0 node or $m \geq 2$. Note that there may be up to $m - 1$ or $m - 2$ maximal type 0 nodes that share the same parent ($m - 1$ if the parent is type 1, $m - 2$ if the parent is type 2). Type 0 nodes that are not maximal are in a unique subtree rooted at a maximal type 0 node. Since we have seen that such a subtree has at most m^{kr^r} nodes, there are at most $((n-1)(m-2) + (2n-1)k((n+1)r)^r(m-1))m^{kr^r}$ type 0 nodes in total.

Therefore, the number of nodes of T is bounded by

$$2n-1+(2n-1)k((n+1)r)^r+((n-1)(m-2)+(2n-1)k((n+1)r)^r(m-1))m^{kr^r}$$

when $n \geq 1$, which is $O(n^{r+1})$. $\square$

3.1.2 Untyped λ-calculus with constants

In this and the next two sections, we review some basic concepts in λ-calculus we will need in what follows, introducing some nonstandard notions and notations along the way. For a more thorough introduction to the subject, see [6], [31], [73], or [32]. Like Sorensen and Urzyczyn [73], we make an explicit distinction between λ-terms and notations that represent them. It is important for our purposes to be completely precise about basic notions such as "subterm occurrence", "substitution", "β-reduction", "descendant", etc.

Following Statman [74], we consider a λ-term as an abstract object—namely, a binary tree equipped with some additional structure. We use a fixed scheme of naming nodes in a tree with strings of 0s and 1s. A *binary tree domain* is a finite, prefix-closed subset $\mathcal{T}$ of $\{0,1\}^*$ such that $w1 \in \mathcal{T}$ implies $w0 \in \mathcal{T}$. A node of the form wi with $i \in \{0,1\}$ is a child of the node w. A node is a leaf, a unary node, or a binary node according to whether it has 0, 1, or 2 children. We write $\mathcal{T}^{(0)}, \mathcal{T}^{(1)}, \mathcal{T}^{(2)}$, for the sets of leaves, unary nodes, and binary nodes, respectively, of $\mathcal{T}$. We write $v \leq w$ to mean v is a prefix of w, and $v < w$ to mean $v \leq w$ and $v \neq w$. The *lexicographic order* on $\{0,1\}^*$ is the strict total order $\prec$ extending $<$ such that $u0t \prec u1t'$ for every $u, t, t' \in \{0,1\}^*$. We say that v is *to the left of* w if $v \prec w$. We let $|w|$ denote the length of the string w. If $w \in \mathcal{T}$, then the *height* of w in $\mathcal{T}$ is $\max\{\, |v| \mid wv \in \mathcal{T}\,\}$. Note that $v < w$ implies that the height of v is greater than the height of w.

We assume that we are given a fixed countably infinite set $\mathcal{V} = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2, \dots\}$ of *variables*. Let C be a finite set of *constants*. A λ-*term* over C is a structure $(\mathcal{T}, f, b)$, where

- $\mathcal{T}$ is a binary tree domain,
- f is a function from a subset of $\mathcal{T}^{(0)}$ to $C \cup \mathcal{V}$,
- b is a function from $\mathcal{T}^{(0)} - \mathrm{dom}(f)$ to $\mathcal{T}^{(1)}$ such that for all $w \in \mathrm{dom}(b)$, $b(w) < w$.

Let $M = (\mathcal{T}, f, b)$ be a λ-term over C. If $c \in C$ and $f(w) = c$, we say that c *occurs* at w in M, and call the node w an *occurrence* of c in M. If $x \in \mathcal{V}$ and $f(w) = x$, then we say that x *occurs free* at w in M, and call w a *free occurrence* of x in M. For $w \in \mathrm{dom}(b)$, we call $b(w)$ the *binder* of w. The set of variables that occur free in M is written $\mathrm{FV}(M)$; its elements are the *free variables* of M. When $\mathrm{FV}(M) = \varnothing$, M is a *closed* λ-term (over C). When no constant occurs in M, M is called a *pure* λ-term.

Let $M = (\mathcal{T}_M, f_M, b_M)$ and $N = (\mathcal{T}_N, f_N, b_N)$ be λ-terms (over C). Then the *application* of M to N is the λ-term $MN = (\mathcal{T}, f, b)$ defined as follows:

$$\begin{aligned}
\mathcal{T} &= \{\epsilon\} \cup 0\mathcal{T}_M \cup 1\mathcal{T}_N,\\
f &= \{\, (0w, f_M(w)) \mid w \in \mathrm{dom}(f_M) \,\} \cup \{\, (1w, f_N(w)) \mid w \in \mathrm{dom}(f_N) \,\},\\
b &= \{(0w, 0b_M(w)) \mid w \in \mathrm{dom}(b_M) \,\} \cup \{\, (1w, 1b_N(w) \mid w \in \mathrm{dom}(b_N) \,\}.
\end{aligned}$$

It is easy to see that the map $(M, N) \mapsto MN$ is one-to-one and every λ-term whose root is a binary node is an application.

Let M be as above. For each variable $x \in \mathcal{V}$, we define the λ-term $\lambda x.M = (\mathcal{T}, f, b)$ by:

$$\begin{aligned}
\mathcal{T} &= 0\mathcal{T}_M,\\
f &= \{\, (0w, f_M(w)) \mid w \in \mathrm{dom}(f_M) \text{ and } f_M(w) \neq x \,\},\\
b &= \{\, (0w, 0b_M(w)) \mid w \in \mathrm{dom}(b_M) \,\} \cup \{\, (0w, \epsilon) \mid w \in \mathrm{dom}(f_M) \text{ and } f_M(w) = x \,\}.
\end{aligned}$$

A λ-term of the form $\lambda x.M$ is called a λ-*abstract*. Clearly, any λ-term P whose root is a unary node is a λ-abstract; indeed, given any variable $x \notin \mathrm{FV}(P)$, P can be written uniquely as $\lambda x.M$.

A λ-*expression* over C is an expression built up from variables, constants, parentheses, the dot ".", and the symbol λ by the following rules:[26]

- If $c \in C$, then c is a λ-expression over C.
- If $x \in \mathcal{V}$, then x is a λ-expression over C.
- If M, N are λ-expressions over C, then (MN) is a λ-expression over C.
- If M is a λ-expression over C and $x \in \mathcal{V}$, then $(\lambda x.M)$ is a λ-expression over C.

Then each λ-expression represents a λ-term, under the convention that a constant or variable $a \in \mathcal{C} \cup \mathcal{V}$ represents the λ-term

$$(\{\epsilon\}, \{(\epsilon, a)\}, \varnothing).$$

[26]A λ-expression is called a *pre-term* by Sorensen and Urzyczyn [73].

It is clear that a λ-expression has the same tree structure as the λ-term it represents.

If $M = (\mathcal{T}, f, b)$ is a λ-term, a *writing* of M [74] is a function $\ell\colon \mathcal{T}^{(1)} \to \mathcal{V}$ satisfying the following conditions:

- If $u, v \in \mathcal{T}^{(1)}, w \in \mathcal{T}^{(0)}, u < v < w$, and $b(w) = u$, then $\ell(u) \neq \ell(v)$.
- If $u \in \mathcal{T}^{(1)}, v \in \mathcal{T}^{(0)}, u < v$, and $v \in \mathrm{dom}(f)$, then $\ell(u) \neq f(v)$.

It is clear that every λ-term has a writing; in particular, there is always a writing ℓ of M such that ℓ is one-to-one and $\mathrm{ran}(\ell) \cap \mathrm{FV}(M) = \varnothing$.[27]

Given a λ-term $M = (\mathcal{T}, f, b)$ together with a writing ℓ, we can define a function $\mathrm{sub}_{M,\ell}$ from $\mathcal{T}$ to λ-expressions as follows:

$$\mathrm{sub}_{M,\ell}(w) = \begin{cases} f(w) & \text{if } w \in \mathrm{dom}(f), \\ \ell(b(w)) & \text{if } w \in \mathrm{dom}(b), \\ \lambda x.\, \mathrm{sub}_{M,\ell}(w0) & \text{if } w \in \mathcal{T}^{(1)} \text{ and } \ell(w) = x, \\ (\mathrm{sub}_{M,\ell}(w0)\ \mathrm{sub}_{M,\ell}(w1)) & \text{if } w \in \mathcal{T}^{(2)}. \end{cases}$$

Then it is easy to see that $\mathrm{sub}_{M,\ell}(\epsilon)$ is a λ-expression representing M. The λ-term represented by $\mathrm{sub}_{M,\ell}(w)$ is usually called the *subterm* of M occurring at w; but "subterm" is only defined relative to a writing ℓ of M.

We use usual abbreviations in writing λ-expressions. We omit the outermost parentheses from λ-expressions and write MNP for $(MN)P$, $\lambda x.MN$ for $\lambda x.(MN)$, and $\lambda x_1 x_2 \dots x_n.M$ for $\lambda x_1.(\lambda x_2.\dots(\lambda x_n.M)\dots)$.

We define the operation of *substitution* of a λ-term for a free variable in another λ-term. Let $M = (\mathcal{T}_M, f_M, b_M)$ and $N = (\mathcal{T}_N, f_N, b_N)$ be λ-terms and x be a variable in $\mathcal{V}$. Let $X = \{\, v \in \mathcal{T}_M^{(0)} \mid f_M(v) = x \,\}$. The result of *substituting* N for x in M is the λ-term $M[x := N] = (\mathcal{T}, f, b)$ defined by

$$\begin{aligned} \mathcal{T} &= \mathcal{T}_M \cup X\mathcal{T}_N, \\ f &= \{\, (w, f_M(w)) \mid w \in \mathrm{dom}(f_M) - X \,\} \cup \{\, (vw, f_N(w)) \mid v \in X, w \in \mathrm{dom}(f_N) \,\}, \\ b &= b_M \cup \{\, (vw, v b_N(w)) \mid v \in X, w \in \mathrm{dom}(b_N) \,\}. \end{aligned}$$

It follows from this definition that for all λ-terms P, Q, N, all $y \in \mathcal{V} - \{x\}$, and all $z \in \mathcal{V} - (\{x\} \cup \mathrm{FV}(N))$, we have

$$\begin{aligned} x[x := N] &= N, \\ y[x := N] &= y, \end{aligned}$$

[27]Such a writing corresponds to what Loader [54] calls a *regular* λ-term.

$$
\begin{aligned}
(PQ)[x := N] &= P[x := N]\, Q[x := N], \\
(\lambda x.P)[x := N] &= \lambda x.P, \\
(\lambda z.P)[x := N] &= \lambda z.(P[x := N]).
\end{aligned}
$$

The *simultaneous substitution* of λ-terms $N_1, \dots, N_k$ for pairwise distinct variables $x_1, \dots, x_k$ in a λ-term M is defined similarly, and is written $M[x_1{:=}N_1, \dots, x_k{:=}N_k]$. We write $M[x_1, \dots, x_k]$ to indicate that $\{x_1, \dots, x_k\} \subseteq \mathrm{FV}(M[x_1, \dots, x_k])$, and write $M[N_1, \dots, N_k]$ for $(M[x_1, \dots, x_k])[x_1 := N_1, \dots, x_k := N_k]$.

Let $M = (\mathcal{T}, f, b)$ be a λ-term. Suppose that $w \in \mathcal{T}^{(2)}$ is a binary node of M such that $w0 \in \mathcal{T}^{(1)}$. Such a node w is called a *β-redex.* Note that for every writing ℓ of M, the λ-term represented by $\mathrm{sub}_{M,\ell}(w)$ is of the form $(\lambda x.P)N$. Let $X = \{\, v \mid w0v \in \mathcal{T}^{(0)}, b(w0v) = w0 \,\}$. (The set of leaves of M whose binder is $w0$ is $w0X$.) We write

$$M \overset{w}{\to}_\beta M'$$

if $M' = (\mathcal{T}', f', b')$, where

$$
\begin{aligned}
\mathcal{T}' &= \{\, u \in \mathcal{T} \mid w \not\leq u \,\} \cup \{\, wv \mid w00v \in \mathcal{T} \,\} \cup \{\, wvu \mid v \in X, w1u \in \mathcal{T} \,\}, \\
f &= \{\, (u, f(u)) \mid u \in \mathrm{dom}(f), w \not\leq u \,\} \cup \{\, (wv, f(w00v)) \mid w00v \in \mathrm{dom}(f) \,\} \cup \\
&\qquad \{\, (wvu, f(w1u)) \mid v \in X, w1u \in \mathrm{dom}(f) \,\}, \\
b' &= \{\, (u, b(u)) \mid u \in \mathrm{dom}(b), w \not\leq u \,\} \cup \\
&\qquad \{\, (wv, b(w00v) \mid w00v \in \mathrm{dom}(b), w \not\leq b(w00v) \,\} \cup \\
&\qquad \{\, (wv, wv') \mid w00v \in \mathrm{dom}(b), b(w00v) = w00v' \,\} \cup \\
&\qquad \{\, (wvu, b(w1u)) \mid v \in X, w1u \in \mathrm{dom}(b), w \not\leq b(w1u) \,\} \cup \\
&\qquad \{\, (wvu, wvu') \mid v \in X, w1u \in \mathrm{dom}(b), b(w1u) = w1u' \,\}.
\end{aligned}
$$

See Figure 9. If ℓ is a writing of M and $(\lambda x.P)N$ is the λ-term represented by $\mathrm{sub}_{M,\ell}(w)$, then for every writing ℓ' of M' such that ℓ' agrees with ℓ on $\{\, u \in \mathcal{T}^{(1)} \mid u < w \,\}$, $\mathrm{sub}_{M',\ell'}(w)$ represents $P[x := N]$.

From here on, we will let λ-expressions denote λ-terms, rather than themselves, unless we explicitly indicate otherwise, keeping in mind that distinct λ-expressions may represent the same λ-term. For example, if $M = c(\lambda y.d((\lambda x.yxx)(yzz)))$, then the node 101 of M is a β-redex, and $M \overset{101}{\to}_\beta c(\lambda y.d(y(yzz)(yzz))) = c(\lambda y.d((yxx)[x := yzz]))$.

We write $M \to_\beta M'$ if $M \overset{w}{\to}_\beta M'$ for some β-redex w in M. We say that M *β-reduces to* M' (or M' *β-expands to* M) and write $M \twoheadrightarrow_\beta M'$ if there is a finite sequence of λ-terms $M_0, M_1, \dots, M_n$ ($n \geq 0$) such that

$$M = M_0 \to_\beta M_1 \to_\beta \cdots \to_\beta M_n = M'.$$

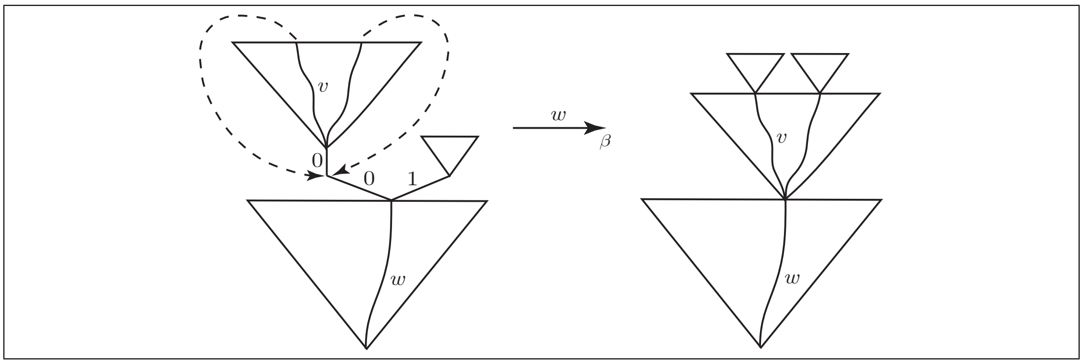


Figure 9: A one-step β-reduction. The dotted arrows represent the binding map.

If M and M' are related by the symmetric transitive closure of the relation $\twoheadrightarrow_\beta$, we say M is *β-equal* to M' and write $M =_\beta M'$.

Theorem 3.3 (Church-Rosser Theorem). *If $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$, then there exists a Q such that $N \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta Q$.*

See [6] for a proof.

A λ-term is called *β-normal* if it does not contain a β-redex. If a λ-term β-reduces to a β-normal λ-term, the latter is called the *β-normal form* of the former. By the Church-Rosser Theorem for β-reduction, any λ-term M has at most one β-normal form. If a λ-term M has a β-normal form, we denote it by $|M|_\beta$.

If $M \twoheadrightarrow_\beta M'$, each node of M' is a *descendant* of a unique node (its *ancestor*) of M. For example, in $M = (\lambda x.yxx)(zw) \twoheadrightarrow_\beta y(zw)(zw) = M'$, both occurrences of z in M' are descendants of the unique occurrence of z in M. We give the definition of the ancestor-descendant relation for one-step β-reduction as follows.[28]

Let $M = (\mathcal{T}, f, b), M' = (\mathcal{T}', f', b')$, and suppose w is a β-redex in M and $M \xrightarrow{w}_\beta M'$. We write $(M, u) \overset{w}{\blacktriangleright} (M', u')$ to mean that the node u' of M' is a descendant of the node u of M. Let $u \in \mathcal{T}$. There are four cases to consider:

Case 1. $w \not\leq u$. Then $(M, u) \overset{w}{\blacktriangleright} (M', u')$ if and only if $u' = u$.

Case 2. $u = w$ or $u = w0$. Then there is no u' such that $(M, u) \overset{w}{\blacktriangleright} (M', u')$.

Case 3. $u = w00s$. Case 3a. If $u \in \mathrm{dom}(b)$ and $b(u) = w0$, then there is no u' such that $(M, u) \overset{w}{\blacktriangleright} (M', u')$. Case 3b. Otherwise, $(M, u) \overset{w}{\blacktriangleright} (M', u')$ if and only if $u' = ws$.

Case 4. $u = w1s$. Then $(M, u) \overset{w}{\blacktriangleright} (M', u')$ if and only if $u' = wvs$ for some v such that $w00v \in \mathrm{dom}(b)$ and $b(w00v) = w0$.

[28]See [10] for a formal definition of the ancestor-descendant relation using the technique of labeling bracket pairs, originally due to Newman [58].

It is clear that each node of M' is a descendant of a unique node of M. In Cases 1 and 3b, the node u of M has just one descendant in M'. In Case 4, it has as many descendants in M' as there are leaves in M whose binder is $w0$. We write $(M,u) \overset{w}{\blacktriangleright}_k (M',u')$ to mean that the node u' of M' is the k-th among the descendants of the node u of M under the lexicographic ordering of the nodes of M'.

Here are some important properties of the ancestor-descendant relation. The proof is by straightforward inspection.

Lemma 3.4. *Let $M = (\mathcal{T}, f, b)$ and $M' = (\mathcal{T}', f', b')$, and suppose $(M,u) \overset{w}{\blacktriangleright} (M',u')$.*

(i) *$u \in \mathcal{T}^{(i)}$ if and only if $u' \in \mathcal{T}'^{(i)}$ for $i = 0, 1, 2$.*

(ii) *$u \in \mathrm{dom}(f)$ if and only if $u' \in \mathrm{dom}(f')$.*

(iii) *$u \in \mathrm{dom}(b)$ if and only if $u' \in \mathrm{dom}(b')$.*

(iv) *If $u \in \mathrm{dom}(b)$, then $(M, b(u)) \overset{w}{\blacktriangleright} (M', b'(u'))$.*

We write $(M,v) \overset{w_1,\dots,w_n}{\blacktriangleright} (M',v')$ if there are sequences $M_0, M_1, \dots, M_n$ and $v_0, v_1, \dots, v_n$ such that $(M,v) = (M_0, v_0), (M',v') = (M_n, v_n)$, and for $1 \leq i \leq n$, $(M_{i-1}, v_{i-1}) \overset{w_i}{\blacktriangleright} (M_i, v_i)$. The following theorem says that if $M \twoheadrightarrow_\beta M'$ and M' is in β-normal form, the ancestor-descendant relation between the nodes of M and the nodes of M' does not depend on the β-reduction sequence from M to M'.

Theorem 3.5. *If $(M,u) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v)$ and $(M,u') \overset{v_1,\dots,v_m}{\blacktriangleright} (|M|_\beta, v)$, then $u = u'$.*

Proof. The proof is via an equivalent definition of the ancestor-descendant relation in terms of *simply labeled λ-calculus* $\lambda_{\mathscr{A}}$ [10]. This calculus defines β-reduction on *labeled λ-terms*, where each node carries a label, and the label of a node is passed to the node's descendants. If u is the only node labeled by a in a labeled λ-term M, the set of descendants of u in $|M|_\beta$ consists of those nodes labeled by a, which is independent of the β-reduction path from M to $|M|_\beta$ because $\lambda_{\mathscr{A}}$, being an orthogonal combinatory reduction system, enjoys the Church-Rosser Property (see [10] for details). $\square$

A unary node w of $M = (\mathcal{T}, f, b)$ is an *η-redex* if $w0$ is a binary node and $w01$ is the only node whose binder is w. If ℓ is a writing of M, then the λ-term represented by $\mathrm{sub}_{M,\ell}(w)$ is of the form $\lambda x.Px$, where $x \notin \mathrm{FV}(P)$. We write

$$M \xrightarrow{w}_\eta M'$$

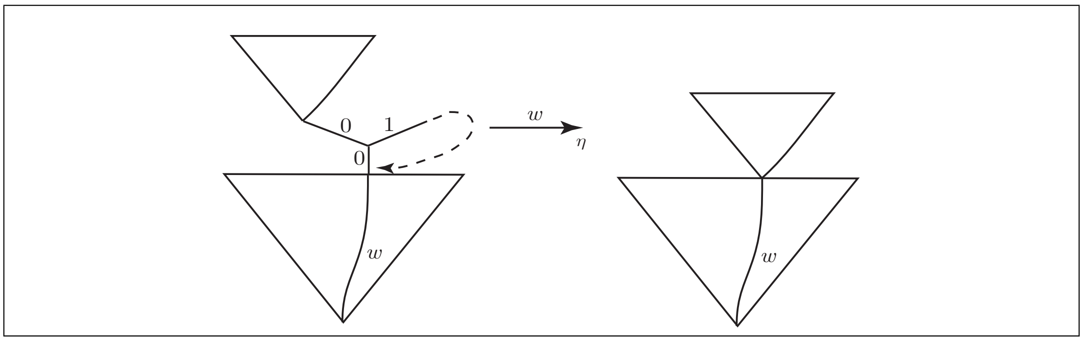


Figure 10: A one-step η-reduction. The node $w01$ is the unique node whose binder is w.

if $M' = (\mathcal{T}', f', b')$, where

$$\begin{aligned}
\mathcal{T}' &= \{\, u \in \mathcal{T} \mid w \not\leq u \,\} \cup \{\, wv \mid w00v \in \mathcal{T} \,\},\\
f' &= \{\, (u, f(u)) \mid u \in \mathrm{dom}(f), w \not\leq u \,\} \cup \{\, (wv, f(w00v) \mid w00v \in \mathrm{dom}(f) \,\},\\
b' &= \{\, (u, b(u)) \mid u \in \mathrm{dom}(b), w \not\leq u \,\} \cup\\
&\qquad \{\, (wv, b(w00v) \mid w00v \in \mathrm{dom}(b), b(w00v) < w \,\} \cup\\
&\qquad \{\, (wv, wv') \mid w00v \in \mathrm{dom}(b), b(w00v) = w00v' \,\}.
\end{aligned}$$

See Figure 10. If ℓ is a writing of M and $\lambda x.Px$ is the λ-term represented by $\mathrm{sub}_{M,\ell}(w)$, then for every writing ℓ' of M' such that ℓ' agrees with ℓ on $\{\, u \in \mathcal{T}^{(1)} \mid u < w \,\}$, the λ-term represented by $\mathrm{sub}_{M',\ell'}(w)$ is P. The notions of *η-reduction*, *η-expansion*, and *η-equality* are defined analogously to β-reduction, β-expansion, and β-equality. We write $M \twoheadrightarrow_\eta M'$ to mean M η-reduces to M' and $M =_\eta M'$ to mean M is η-equal to M'. The transitive closure of the union of $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_\eta$ is written $\twoheadrightarrow_{\beta\eta}$, and similarly for $=_{\beta\eta}$.

The following are lemmas needed to prove the Church-Rosser Theorem for $\beta\eta$-reduction (see [6] for a proof):

Lemma 3.6 (η-Postponmenet Theorem)**.** *If $M \twoheadrightarrow_\eta Q \twoheadrightarrow_\beta T$, then there exists a λ-term P such that $M \twoheadrightarrow_\beta P \twoheadrightarrow_\eta T$.*

Lemma 3.7 (Commuting Lemma)**.** *If $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\eta Q$, then there exists a λ-term T such that $P \twoheadrightarrow_\eta T$ and $Q \twoheadrightarrow_\beta T$.*

The following lemma is straightforward (see [31]):

Lemma 3.8. *If M is in β-normal form and $M \twoheadrightarrow_\eta M'$, then M' is in β-normal form.*

A λ-term M is a *λI-term* if every unary node of M binds at least one leaf. A λ-term M is *affine* if every variable occurs free in M at most once, and every unary node of M binds at most one leaf. A λ-term is *linear* if it is an affine λI-term. The class of λI-terms and the class of affine λ-terms are both closed under β-reduction and η-equality.

We introduce some nonstandard notations. The *sequence of constants in* $M = (\mathcal{T}, f, b)$, denoted $\overrightarrow{\mathrm{Con}}(M)$, is $\overrightarrow{\mathrm{Con}}(M, \epsilon)$, where $\overrightarrow{\mathrm{Con}}(M, w)$ is defined as follows:

$$\overrightarrow{\mathrm{Con}}(M,w) = \begin{cases} () & \text{if } w \in \mathrm{dom}(b), \\ () & \text{if } w \in \mathrm{dom}(f) \text{ and } f(w) \in \mathcal{V}, \\ (f(w)) & \text{if } w \in \mathrm{dom}(f) \text{ and } f(w) \in C, \\ \overrightarrow{\mathrm{Con}}(M, w0) & \text{if } w \in \mathcal{T}^{(1)}, \\ \overrightarrow{\mathrm{Con}}(M, w0)^\frown \overrightarrow{\mathrm{Con}}(M, w1) & \text{if } w \in \mathcal{T}^{(2)}, \end{cases}$$

where $^\frown$ denotes juxtaposition. The *sequence of free variables of* M, denoted $\overrightarrow{\mathrm{FV}}(M)$, is $\overrightarrow{\mathrm{FV}}(M, \epsilon)$, where $\overrightarrow{\mathrm{FV}}(M, w)$ is defined as follows:

$$\overrightarrow{\mathrm{FV}}(M,w) = \begin{cases} () & \text{if } w \in \mathrm{dom}(b), \\ (f(w)) & \text{if } w \in \mathrm{dom}(f) \text{ and } f(w) \in \mathcal{V}, \\ () & \text{if } w \in \mathrm{dom}(t) \text{ and } f(w) \in C, \\ \overrightarrow{\mathrm{FV}}(M, w0) & \text{if } w \in \mathcal{T}^{(1)}, \\ \overrightarrow{\mathrm{FV}}(M, w0)^\frown \overrightarrow{\mathrm{FV}}(M, w1) & \text{if } w \in \mathcal{T}^{(2)}, \end{cases}$$

If $\overrightarrow{\mathrm{Con}}(M) = (c_1, \dots, c_n)$ and $\{x_1, \dots, x_n\} \cap \mathrm{FV}(M) = \varnothing$ (with $x_1, \dots, x_n$ pairwise distinct), we let $\widehat{M}[x_1, \dots, x_n]$ denote the pure λ-term such that $(x_1, \dots, x_n)$ is a subsequence of $\overrightarrow{\mathrm{FV}}(\widehat{M}[x_1, \dots, x_n])$ and $M = \widehat{M}[c_1, \dots, c_n]$. For example, if $M = \lambda y.c(y(c(zd)))$, then $\overrightarrow{\mathrm{Con}}(M) = (c, c, d)$, $\overrightarrow{\mathrm{FV}}(M) = (z)$, $\widehat{M}[x_1, x_2, x_3] = \lambda y.x_1(y(x_2(zx_3)))$, and $\overrightarrow{\mathrm{FV}}(\widehat{M}[x_1, x_2, x_3]) = (x_1, x_2, z, x_3)$.

3.1.3 Simply typed λ-calculus with constants

Given a set A of *atomic types*, we let $\mathscr{T}(A)$ denote the set of *types* built up from atomic types using $\rightarrow$ as the sole type constructor. In other words, $\mathscr{T}(A)$ is the smallest set extending A such that

$$\alpha, \beta \in \mathscr{T}(A) \quad \text{implies} \quad (\alpha \rightarrow \beta) \in \mathscr{T}(A).$$

We omit the outermost parentheses in writing types, and write $\alpha \rightarrow \beta \rightarrow \gamma$ to mean $\alpha \rightarrow (\beta \rightarrow \gamma)$.

For $\alpha \in \mathscr{T}(A)$, we write $|\alpha|$ to denote the number of occurrences of atomic types in α. The notation $\overline{\alpha}$ denotes the sequence of atomic types (with repetitions) that appear in α *from right to left*, defined as follows:

$$\begin{aligned} \overline{p} &= (p) \quad \text{if } p \in A, \\ \overline{\alpha \to \beta} &= \overline{\beta}{}^\frown \overline{\alpha}. \end{aligned}$$

As before, $^\frown$ denotes juxtaposition of sequences. For example, $\overline{p \to p \to q} = (q, p, p)$. Note that the length of $\overline{\alpha}$ is $|\alpha|$.

The set of *positions* within α, denoted $\langle \alpha \rangle$, is defined as follows:

$$\begin{aligned} \langle p \rangle &= \{\epsilon\}, \\ \langle \alpha \to \beta \rangle &= \{\epsilon\} \cup 1\langle \alpha \rangle \cup 0\langle \beta \rangle. \end{aligned}$$

Then for every type α, $\langle \alpha \rangle$ is a binary tree domain that has no unary nodes. The *subtype* of α that occurs at position $w \in \langle \alpha \rangle$, $\text{subtype}(\alpha, w)$ in symbols, is defined as follows:

$$\begin{aligned} \text{subtype}(\alpha, \epsilon) &= \alpha, \\ \text{subtype}(\alpha \to \beta, 1w) &= \text{subtype}(\alpha, w), \\ \text{subtype}(\alpha \to \beta, 0w) &= \text{subtype}(\beta, w). \end{aligned}$$

The *polarity* of position w, $\text{pol}(w)$, is 1 if the number of occurrences of 1 in w is even, -1 otherwise. We say that β occurs positively (negatively) at position w in α if $\text{subtype}(\alpha, w) = \beta$ and $\text{pol}(w) = 1$ ($\text{pol}(w) = -1$).

A *type substitution* is a mapping σ from $\mathscr{T}(A)$ to $\mathscr{T}(A')$, written in postfix notation, satisfying the condition $(\alpha \to \beta)\sigma = \alpha\sigma \to \beta\sigma$. A *type relabeling* is a type substitution that sends atomic types to atomic types. Note that $\langle \alpha \rangle = \langle \beta \rangle$ if and only if there exist a type γ and type relabelings σ_1 and σ_2 such that $\alpha = \gamma\sigma_1$ and $\beta = \gamma\sigma_2$. If $|\alpha| = n$ and $q_1, \dots, q_n \in A$, then we let $\langle \alpha \rangle(q_1, \dots, q_n)$ denote the unique type β in $\mathscr{T}(A)$ such that $\overline{\beta} = (q_1, \dots, q_n)$ and $\langle \alpha \rangle = \langle \beta \rangle$. For any type β, we have $\langle \beta \rangle(\overline{\beta}) = \beta$.

A *higher-order signature* is a triple (A, C, τ), where A is a finite set of atomic types, C is a finite set of constants, and τ is a mapping from C to $\mathscr{T}(A)$. We write $\Lambda(\Sigma)$ for the set of λ-terms over C.

A *type environment* is a finite partial function from $\mathcal{V}$ to $\mathscr{T}(A)$. A type environment $\Gamma = \{(x_1, \alpha_1), \dots, (x_n, \alpha_n)\}$ is usually written as a list $x_1 : \alpha_1, \dots, x_n : \alpha_n$.

Let Γ be a type environment and $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$. A function $t\colon \mathcal{T} \to \mathscr{T}(A)$ is a *type decoration of* M *under* Γ if $\mathrm{dom}(\Gamma) = \mathrm{FV}(M)$ and

$$t(w) = \begin{cases} \Gamma(f(w)) & \text{if } w \in \mathrm{dom}(f) \text{ and } f(w) \in \mathcal{V}, \\ \tau(f(w)) & \text{if } w \in \mathrm{dom}(f) \text{ and } f(w) \in C, \\ \gamma & \text{if } w \in \mathrm{dom}(b) \text{ and } t(b(w)) = \gamma \to \delta, \\ \gamma \to \delta & \text{for some } \gamma \text{ if } w \in \mathcal{T}^{(1)} \text{ and } t(w0) = \delta, \\ \gamma \to \delta & \text{if for some } v \in \mathcal{T}^{(2)},\ w = v0,\ t(v) = \delta, \text{ and } t(v1) = \gamma. \end{cases}$$

If t is a type decoration of M (under Γ), we call (M, t) a *typed* λ*-term* over Σ (under Γ).

A typed λ-term (M, t) can be visualized in the form of a *natural deduction*: each unary and binary node w is labeled with its type $t(w)$, each node $w \in \mathrm{dom}(f)$ is labeled with $a{:}\gamma$, where $f(w) = a$ and $t(w) = \gamma$, and each node $w \in \mathrm{dom}(b)$ is labeled with $[\gamma]^v$, where $b(w) = v$ and $t(w) = \gamma$. For example, the following figure depicts a typed λ-term (M, t) under the type environment $z : p$, where $M = (\lambda y.y(yz))(\lambda x.x)$:

$$\dfrac{\dfrac{\dfrac{[p \to p]^0 \quad \dfrac{[p \to p]^0 \quad z : p}{p}}{p}}{(p \to p) \to p}\ 0 \qquad \dfrac{[p]^1}{p \to p}\ 1}{p}$$

To aid legibility, we have also placed the label v next to the horizontal line right above each unary node v.[29]

Another familiar representation of a typed λ-term is by means of a λ-expression together with a type superscript on each of its subexpression. For instance, one way of representing the above example of a typed λ-term is

$$((\lambda y^{p \to p}.(y^{p \to p}(y^{p \to p} z^p)^p)^p)^{(p \to p) \to p}(\lambda x^p.x^p)^{p \to p})^p.$$

We call an expression of the form $\Gamma \Rightarrow \alpha$, where Γ is a type environment and α is a type, a *sequent.* A sequent $\Gamma \Rightarrow \alpha$ is a *typing* of M if there is a type decoration t of M under Γ such that $t(\epsilon) = \alpha$. In this case, we write

$$\vdash_\Sigma \Gamma \Rightarrow M : \alpha.$$

[29]The resulting figure is identical to the natural deduction as defined in, e.g., [75], except that we use strings in $\{0,1\}^*$, rather than variables, as markers for closed assumptions, and we label open assumptions with variables or constants. Hindley [31] also uses node addresses as assumption markers in natural deductions, albeit in a different way.

and say that t is a type decoration for the *typing judgment* $\Gamma \Rightarrow M : \alpha$. When Γ is empty, we omit the symbol $\Rightarrow$ and write $\vdash_\Sigma M : \alpha$. Reference to Σ is dropped when M is pure.

We say that an (untyped) λ-term M is *typable* if it has a typing. It is known that every typable λ-term has a β-normal form. A sequent is said to be *inhabited* if there is a pure λ-term M (an *inhabitant*) such that $\vdash \Gamma \Rightarrow M : \alpha$. A sequent is inhabited if and only if it is a theorem of intuitionistic logic.[30]

Let $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$ and t be a type decoration of M. If ℓ is a writing of M and $w \in \mathcal{T}$, then it is clear that

$$t_w(v) = t(wv) \quad \text{for } wv \in \mathcal{T}$$

determines a type decoration t_w for $\mathrm{sub}_{M,\ell}(w)$, and we have

$$\vdash_\Sigma \{\, (x, t(wv)) \mid f(wv) = x \} \cup \{\, (\ell(b(wv)), t(wv)) \mid b(wv) < w \,\} \Rightarrow \mathrm{sub}_{M,\ell}(w) : t(w).$$

An important property of a typed λ-term in β-normal form is the so-called *subformula property*:

Theorem 3.9. *Let $M = (\mathcal{T}, f, b)$ be a pure untyped λ-term in β-normal form. If t is a type decoration for $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow M : \alpha_0$, then for every $w \in \mathcal{T}$, $t(w)$ is a subtype of α_i for some $i \in \{0, \dots, n\}$.*

Proof. The theorem is a consequence of the following statement, which is easy to see: for every $w \in \mathcal{T}$, if $w \neq \epsilon$ and $w \notin \mathrm{dom}(f)$, then there exists a $v \in \mathcal{T}$ such that $t(v) = t(w) \to \alpha$ or $t(v) = \alpha \to t(w)$ for some α. □

In general, the same typing of a λ-term may have more than one type decoration. See [31] for the proof of the following theorem:

Theorem 3.10. *If $M \in \Lambda(\Sigma)$ is a λI-term, any typing of M has a unique type decoration.*

Thus, a λI-term M together with a typing of M can be treated in the same way as a typed λ-term.

A typing $\Gamma \Rightarrow \alpha$ of M is a *principal typing of M* if for every typing $\Gamma' \Rightarrow \alpha'$ of M, there is a type substitution σ such that $\Gamma' \Rightarrow \alpha' = (\Gamma \Rightarrow \alpha)\sigma$. We call a type decoration t of M (under some type environment) a *principal type decoration of M* if for every type decoration t' of M (under some type environment), there is a type substitution σ such that $t' = \sigma \circ t$. Clearly, the typing determined by a principal type decoration is a principal typing.

[30]We use the symbol $\Rightarrow$ in the same way as Mints [56] does. This is the way Hindley [31] uses the symbol $\mapsto$. Although $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$ implies $\mathrm{dom}(\Gamma) = \mathrm{FV}(M)$, it is always possible to weaken the antecedent in the sense that $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$ implies $\vdash_\Sigma \Gamma, x : \beta \Rightarrow (\lambda y.M)x : \alpha$, where $x, y \notin \mathrm{FV}(M)$.

Theorem 3.11 (Principal Type Theorem). *If M is typable, then M has a principal typing and a principal type decoration.*

See [31] for a proof.

Let $M = (\mathcal{T}_M, f_M, b_M)$ and $N = (\mathcal{T}_N, f_N, b_N)$ be λ-terms and x be a variable in $\mathrm{FV}(M)$. Let $X = \{\, v \in \mathcal{T}_M^{(0)} \mid f_M(v) = x \,\}$. Let $M[x := N] = (\mathcal{T}, f, b)$ be the result of substituting N for x in M. The following lemmas are straightforward:

Lemma 3.12. *Suppose that t_M and t_N are type decorations for $\Gamma_1, x : \beta \Rightarrow M : \alpha$ and $\Gamma_2 \Rightarrow N : \beta$, respectively, and that Γ_1 and Γ_2 agree on $(\mathrm{FV}(M) - \{x\}) \cap \mathrm{FV}(N)$. Then we can define a type decoration t for $\Gamma_1 \cup \Gamma_2 \Rightarrow M[x := N] : \alpha$ by*

$$t(w) = \begin{cases} t_M(w) & \text{if } w \in \mathcal{T}_M, \\ t_N(v') & \text{if } w = vv' \text{ for some } v \in X \text{ and } v' \in \mathcal{T}_N. \end{cases}$$

Lemma 3.13. *Suppose that t is a type decoration for $\Gamma \Rightarrow M[x := N] : \alpha$ such that for some type β, $t(v) = \beta$ for every $v \in X$. Pick a $v \in X$. Then we can define type decorations t_M and t_N for $\Gamma_1, x : \beta \Rightarrow M : \alpha$ and $\Gamma_2 \Rightarrow N : \beta$, respectively, by*

$$\begin{aligned} t_M(w) &= t(w) \quad \text{for all } w \in \mathcal{T}_M, \\ t_N(w) &= t(vw) \quad \text{for all } w \in \mathcal{T}_N, \end{aligned}$$

where Γ_1 and Γ_2 are the restrictions of Γ to $\mathrm{FV}(M)$ and to $\mathrm{FV}(N)$, respectively.

Let $M[x_1, \dots, x_n]$ be a pure λ-term such that $\mathrm{FV}(M[x_1, \dots, x_n]) = \{x_1, \dots, x_n\}$. For any $c_1, \dots, c_n \in C$, we have

$$\vdash_\Sigma M[c_1, \dots, c_n] : \alpha \quad \text{if and only if} \quad \vdash x_1 : \tau(c_1), \dots, x_n : \tau(c_n) \Rightarrow M[x_1, \dots, x_n] : \alpha.$$

Let (M, t) be a typed λ-term. If $M \overset{w}{\to}_\beta M'$, then t, in conjunction with the ancestor-descendant relation, induces a type decoration t' of M', defined by

$$t'(v') = t(v) \quad \text{if } (M, v) \overset{w}{\blacktriangleright} (M', v').$$

This is denoted by $(M, t) \overset{w}{\to}_\beta (M', t')$. Note that even though we do not have $(M, w) \overset{w}{\blacktriangleright} (M', w)$, it is always the case that $t'(w) = t(w)$, since $t(w) = t(w00)$ and $(M, w00) \overset{w}{\blacktriangleright} (M', w)$.

Theorem 3.14 (Subject Reduction Theorem). *If $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$ and $M \twoheadrightarrow_\beta M'$, then $\vdash_\Sigma \Gamma' \Rightarrow M' : \alpha$, where Γ' is the restriction of Γ to $\mathrm{FV}(M')$.*

See, e.g., [31] for a proof.

Let $M = (\mathcal{T}, f, b)$ and suppose $M \overset{w}{\rightarrow}_\beta M'$. This β-reduction step is called *erasing* if there is no $v \in \mathcal{T}^{(0)}$ such that $b(v) = w0$, and *duplicating* if for some $v, v' \in \mathcal{T}^{(0)}$, $v \neq v'$ and $b(v) = b(v') = w0$. (The right child $w1$ of the β-redex w has no descendant in an erasing β-reduction step, and has more than one in a duplicating β-reduction step.) A β-reduction from M to M' is *non-erasing* (*non-duplicating*) if it consists entirely of non-erasing (non-duplicating) β-reduction steps.

Theorem 3.15 (Subject Expansion Theorem). *If $\vdash_\Sigma \Gamma \Rightarrow M' : \alpha$ and $M \twoheadrightarrow_\beta M'$ by non-erasing, non-duplicating β-reduction, then $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$.*

See [31]. As a special case, if M is linear and $M \twoheadrightarrow_\beta M'$, then $\vdash_\Sigma \Gamma \Rightarrow M' : \alpha$ implies $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$.

As with β-reduction, the η-reduction relation between untyped λ-terms induces the η-reduction relation between typed λ-terms. A typed λ-term (M, t), where $M = (\mathcal{T}, f, b)$, is in *η-long form* if every node $w \in \mathcal{T}$ satisfies the following condition:

- $t(w) = \beta \to \gamma$ for some β, γ implies that either $w \in \mathcal{T}^{(1)}$ or $w = v0$ for some $v \in \mathcal{T}^{(2)}$.

If (M, t) has a node w that does not satisfy this condition, there is a unique typed λ-term (M', t') such that $(M', t') \overset{w}{\rightarrow}_\eta (M, t)$. Both nodes w and $w00$ of (M', t') satisfy the condition, and $t'(w0) = \gamma$, $t'(w01) = \beta$, both of which are shorter than $\beta \to \gamma$. Thus, every typed λ-term can be converted to one in η-long form by a sequence of η-expansion steps applied to nodes that do not satisfy this condition. It is easy to see that the resulting λ-term is unique; we call it the *η-long form* of the original λ-term.

We say that an untyped λ-term $M \in \Lambda(\Sigma)$ is in *η-long form relative to* $\Gamma \Rightarrow \alpha$ if there is a type decoration t of M under Γ such that $t(\epsilon) = \alpha$ and (M, t) is in η-long form. We say that M is in η-long form if M is η-long relative to some typing (or, equivalently, relative to its principal typing).

The following lemmas are from [34]:

Lemma 3.16. *Let M and N be λ-terms and x be a variable in $\mathrm{FV}(M)$. Suppose that t_M and t_N are type decorations for $\Gamma_1, x{:}\beta \Rightarrow M{:}\alpha$ and $\Gamma_2 \Rightarrow N{:}\beta$, respectively, and that Γ_1 and Γ_2 agree on $(\mathrm{FV}(M) - \{x\}) \cap \mathrm{FV}(N)$. Let t be the type decoration for $\Gamma_1 \cup \Gamma_2 \Rightarrow M[x := N] : \alpha$ defined according to Lemma 3.12. If (M, t_M) and (N, t_N) are in η-long form, then $(M[x := N], t)$ is in η-long form.*

Lemma 3.17. *If M is in η-long form relative to $\Gamma \Rightarrow \alpha$ and $M \twoheadrightarrow_\beta M'$, then M' is in η-long form relative to $\Gamma' \Rightarrow \alpha$, where Γ' is the restriction of Γ to $\mathrm{FV}(M')$.*

Thus, the β-normal form of an η-long λ-term is η-long.

We refer to an occurrence of a type β in a sequent $x_1\colon \alpha_1, \dots, x_n\colon \alpha_n \Rightarrow \alpha_0$ or a typing judgment $x_1\colon \alpha_1, \dots, x_n\colon \alpha_n \Rightarrow M\colon \alpha_0$ by a pair (i, v), with $0 \leq i \leq n$ and $v \in \langle \alpha_i \rangle$, such that $\text{subtype}(\alpha_i, v) = \beta$. We say that an occurrence (i, v) is *positive* (resp., *negative*) and write $\text{pol}(i, v) = +1$ ($\text{pol}(i, v) = -1$) if either $i = 0$ and $\text{pol}(v) = 1$ ($\text{pol}(v) = -1$) or $i \geq 1$ and $\text{pol}(v) = -1$ ($\text{pol}(v) = 1$). For example, in $x : p, y : p \to q \Rightarrow q$, the pairs $(1, \epsilon)$ and $(2, 0)$ refer to the first occurrences of p and q, respectively, which are both negative, and the pairs $(2, 1)$ and $(0, \epsilon)$ refer to the second occurrences of p and q, respectively, which are both positive. A sequent or typing judgment is *balanced* if every atomic type has at most one positive and at most one negative occurrence in it.

Theorem 3.18 (Coherence Theorem)**.** *All inhabitants of a balanced sequent are $\beta\eta$-equal. In particular, if $\Gamma \cup \Gamma' \Rightarrow \alpha$ is a balanced sequent and both $\vdash \Gamma \Rightarrow M : \alpha$ and $\vdash \Gamma' \Rightarrow M' : \alpha$ hold, then $M =_{\beta\eta} M'$.*

See [56] for a proof.

According to Hirokawa [33], the first of the following theorems is due to Belnap [9]. See [33] for the proof of the second.

Theorem 3.19 ([9])**.** *If M is a pure affine λ-term, then the principal typing of M is balanced.*

Theorem 3.20 ([33])**.** *If a pure λ-term M in β-normal form has a balanced typing, then M is affine.*

Theorem 3.19 together with the Coherence Theorem (Theorem 3.18) implies that a pure affine λ-term is uniquely determined by its principal typing up to $\beta\eta$-equality.

3.1.4 Links in typed λ-terms

It will be convenient for our purposes to introduce a strengthening of the notion of η-long form. We say that a typed λ-term (M, t) with $M = (\mathcal{T}, f, b)$ is in *strict η-long form* if every node $w \in \mathcal{T}$ satisfies the following condition:

- if $t(w) = \beta \to \gamma$, then either (i) $w \in \mathcal{T}^{(1)}$ and $b(v) = w$ for some $v \in \mathcal{T}^{(0)}$, (ii) $w \in \mathcal{T}^{(1)}$ and β is an atomic type, or (iii) $w = v0$ for some $v \in \mathcal{T}^{(2)}$.

Note that if M is a λI-term and (M, t) is in η-long form, then (M, t) is in strict η-long form. For every typed λ-term (M, t) in η-long form, there is a typed λ-term (M', t') in strict η-long form such that both $(M', t') \twoheadrightarrow_\beta (M, t)$ and $(M', t') \twoheadrightarrow_\eta (M, t)$. Unlike η-long form, strict η-long form is not preserved under β-reduction, but we have the following:

Lemma 3.21. *Lemma 3.16 holds with "strict η-long form" in place of "η-long form".*

As with η-long form, we speak of an untyped λ-term being in strict η-long form relative to a typing.

Clearly, if $M \in \Lambda(\Sigma)$ is a closed λ-term and $\overrightarrow{\mathrm{Con}}(M) = (c_1, \dots, c_n)$, then M is in (strict) η-long form relative to α if and only if $\widehat{M}[x_1, \dots, x_n]$ is in (strict) η-long form relative to $x_1 : \tau(c_1), \dots, x_n : \tau(c_n) \Rightarrow \alpha$.

Lemma 3.22. *Let M be a pure λ-term, and suppose that t is a type decoration of M such that (M, t) is in strict η-long form. Let $\widetilde{t}$ be a principal type decoration of M. Then there is a type relabeling σ such that $t = \sigma \circ \widetilde{t}$.*

Proof. It is easy to see that if an atomic type p occurs anywhere in (M, t), then it must be that there is a node of M that is assigned type p by t, or else there is a unary node of M that is assigned a type of the form $p \to \gamma$. In both cases, the relevant node must be assigned a type of the same shape by $\widetilde{t}$. □

Lemma 3.22 implies the following:

Remark 3.23. Suppose that $M \in \Lambda(\Sigma)$ is a λ-term in strict η-long form relative to $x_1 : \gamma_1, \dots, x_n : \gamma_n \Rightarrow \gamma_0$, $\overrightarrow{\mathrm{Con}}(M) = (d_1, \dots, d_m)$, and $y_1 : \beta_1, \dots, y_m : \beta_m, x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$ is a principal typing of $\widehat{M}[y_1, \dots, y_m]$. Then $\widehat{M}[y_1, \dots, y_m]$ is in strict η-long form relative to $y_1 : \beta_1, \dots, y_m : \beta_m, x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$, and moreover, we have

$$\begin{aligned} \langle \beta_i \rangle &= \langle \tau(d_i) \rangle \quad \text{for } i = 1, \dots, m, \\ \langle \alpha_i \rangle &= \langle \gamma_i \rangle \quad \text{for } i = 0, \dots, n. \end{aligned}$$

Let (M, t) be a pure typed λ-term, where $M = (\mathcal{T}, f, b)$. We associate with (M, t) a certain directed graph $G_{(M,t)} = (V_{(M,t)}, E_{(M,t)})$.[31] The set $V_{(M,t)}$ of vertices of $G_{(M,t)}$ consists of all triples of one of the forms

$$(w, v, \uparrow) \quad \text{and} \quad (w, v, \downarrow),$$

where $w \in \mathcal{T}$ and $v \in \langle t(w) \rangle^{(0)}$. (Recall that $\langle t(w) \rangle^{(0)}$ is the set of leaves of $\langle t(w) \rangle$, that is, the set of positions where atomic types occur in $t(w)$.) Triples $(w, v, \uparrow)$ and $(w, v, \downarrow)$ correspond to the same position in $t(w)$. The existence of an edge from $(w, v, -)$ to $(w', v', -)$ (where "$-$" is to be filled by $\uparrow$ or $\downarrow$) implies that the same

[31]Our graph is essentially the natural deduction counterpart of the *logical flow graph* of Buss [12]. See [41] for an equivalent definition.

atomic type must occur at v in $t(w)$ and at v' in $t(w')$ (i.e., $\mathrm{subtype}(t(w),v) = \mathrm{subtype}(t(w'),v')$). The last component of the triples indicates the "direction of travel", which is explained below. The set $E_{(M,t)}$ of edges of $G_{(M,t)}$ is defined as follows:

$$\begin{aligned}
((w,v,\uparrow),(w',v',\uparrow)) \in E_{(M,t)} &\text{ iff either } w \in \mathcal{T}^{(1)},\ w0 = w',\text{ and } v = 0v',\text{ or}\\
&\qquad w \in \mathcal{T}^{(2)},\ w0 = w',\text{ and } 0v = v'.\\
((w,v,\downarrow),(w',v',\downarrow)) \in E_{(M,t)} &\text{ iff either } w' \in \mathcal{T}^{(1)},\ w = w'0,\text{ and } 0v = v',\text{ or}\\
&\qquad w' \in \mathcal{T}^{(2)},\ w = w'0,\text{ and } v = 0v'.\\
((w,v,\uparrow),(w',v',\downarrow)) \in E_{(M,t)} &\text{ iff either } w \in \mathcal{T}^{(1)},\ w = b(w') \text{ and } v = 1v',\text{ or}\\
&\qquad w' \in \mathcal{T}^{(1)},\ b(w) = w' \text{ and } 1v = v'.\\
((w,v,\downarrow),(w',v',\uparrow)) \in E_{(M,t)} &\text{ iff for some } u \in \mathcal{T}^{(2)},\\
&\text{either } w = u0,\ w' = u1,\text{ and } v = 1v',\text{ or}\\
&\qquad w = u1,\ w' = u0,\text{ and } 1v = v'.
\end{aligned}$$

Note that the edges in $E_{(M,t)}$ come in pairs: given an edge in $E_{(M,t)}$, one can interchange source and destination, then reverse the direction of the arrows in the third component of both vertices, and obtain another edge in $E_{(M,t)}$.

The meaning of the graph $G_{(M,t)}$ becomes easy to grasp when it is superimposed on the natural deduction representing (M,t). Each pair of edges is represented by a single curve connecting two occurrences of an atomic type; the two edges in the pair correspond to the two ways of traversing the curve, with the direction of traversal at each end point of the curve matching the direction of the arrow in the third component of the tuple $(w,v,-)$ corresponding to that point. Thus, $((w,v,\downarrow),(w',v',\downarrow))$ is an edge of the graph $G_{(M,t)}$ if there is a curve that departs downward from the atomic type occurrence at position v in the type labeling the node w of the natural deduction tree for (M,t) and reaches from above the atomic type occurrence at position v' in the type labeling the node w'; similarly for other combinations of $\uparrow$ and $\downarrow$. See Figure 11 for an example.

It is easy to see that for any pure typed λ-term (M,t), if there is a directed path from (w,v,d) to (w',v',d'), where $d,d' \in \{\uparrow,\downarrow\}$, then $\mathrm{pol}(v) = \mathrm{pol}(v')$ if and only if $d = d'$.

Note that the graph depicted in Figure 11 contains a directed cycle:

$$\begin{aligned}(0,10,\downarrow) - (1,0,\uparrow) - (10,\epsilon,\uparrow) - (1,1,\downarrow) - (0,11,\uparrow) - (000,1,\downarrow) - (001,\epsilon,\uparrow) -&\\ (0010,0,\uparrow) - (0,10,\downarrow)&\end{aligned}$$

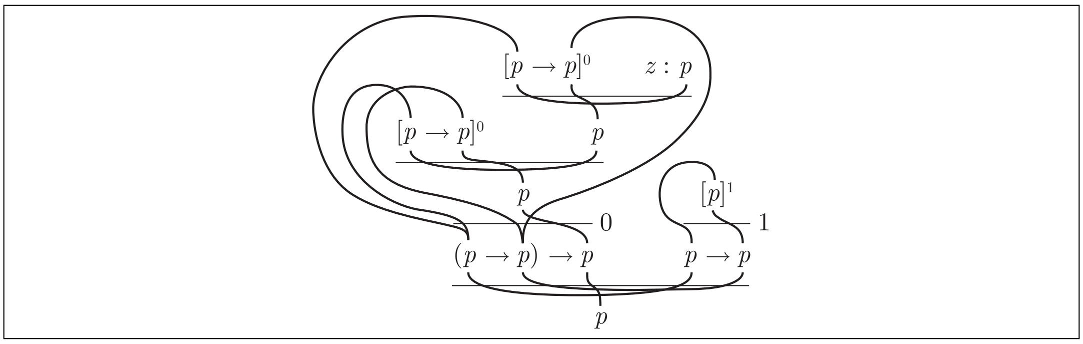


Figure 11: A natural deduction with links.

It is not hard to see that any cycle must involve the two children $w0, w1$ of a binary node w and positions u, v of $t(w1)$ such that

- $\mathrm{pol}(u) = -\,\mathrm{pol}(v)$,
- there is a directed path from $(w0, 1u, \uparrow)$ to $(w0, 1v, \downarrow)$ inside the subtree rooted at $w0$, and
- there is a directed path from $(w1, v, \uparrow)$ to $(w1, u, \downarrow)$ inside the subtree rooted at $w1$.

This implies that there exists an $n \geq 0$ such that $w0^n$ is a β-redex.

Lemma 3.24. *If (M, t) is a pure typed λ-form in β-normal form, then $G_{(M,t)}$ contains no directed cycle.*

Let $M = (\mathcal{T}, f, b)$ be a pure untyped λ-term with $\mathrm{FV}(M) = \{x_1, \dots, x_n\}$, and let t be a type decoration for $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow M : \alpha_0$. We augment the graph $G_{(M,t)}$ with the nodes of the form

$$(i, v, d)$$

where $0 \leq i \leq n$, $v \in \langle \alpha_i \rangle^{(0)}$, $d \in \{\uparrow, \downarrow\}$, and the edges

$$((i, v, \uparrow), (w, v, \downarrow)) \quad \text{and} \quad ((w, v, \uparrow), (i, v, \downarrow))$$

with $1 \leq i \leq n$ and $f(w) = x_i$, and

$$((0, v, \uparrow), (\epsilon, v, \uparrow)) \quad \text{and} \quad ((\epsilon, v, \downarrow), (0, v, \downarrow)).$$

We refer to the resulting extended graph as $\overline{G}_{(M,t)}$. Note that when $\overline{G}_{(M,t)}$ has a directed path from $(i, v, \uparrow)$ to (w, u, d) with $w \in \mathcal{T}$, we have $\mathrm{pol}(i, v) = \mathrm{pol}(u)$ if and only if $d = \uparrow$, and likewise when $\overline{G}_{(M,t)}$ has a directed path from (w, u, d) to $(i, v, \downarrow)$.

In terms of $\overline{G}_{(M,t)}$, we define two binary relations on the set

$$\{\, (i, v) \mid 0 \leq i \leq n, v \in \langle \alpha_i \rangle^{(0)} \,\}$$

of occurrences of atomic types in $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$. We say that (i, v) is *linked* to (i', v') in (M, t) if $\overline{G}_{(M,t)}$ contains a directed path from $(i, v, \uparrow)$ to $(i', v', \downarrow)$. We say that (i, v) is *connected* to (i', v') in (M, t) if $\overline{G}_{(M,t)}$ contains an *undirected* path from (i, v, d) to (i', v', d') for some $d, d' \in \{\uparrow, \downarrow\}$. Note that the relation of being linked is symmetric, but not necessarily transitive; the relation of being connected is symmetric and transitive. Clearly, if (i, v) is connected to (i', v'), then $\mathrm{subtype}(\alpha_i, v) = \mathrm{subtype}(\alpha_{i'}, v')$.

The following is clear from the definitions of $\overline{G}_{(M,t)}$ and of principal typing:

Lemma 3.25. *Let M be a pure λ-term and t be a principal type decoration of M, with the associated principal typing $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$. Then (i, v) and (i', v') are connected in (M, t) if and only if* $\mathrm{subtype}(\alpha_i, v) = \mathrm{subtype}(\alpha_{i'}, v')$.

It is clear that the graph $\overline{G}_{(M,t)}$, where $M = (\mathcal{T}, f, b)$, is completely determined by M and $\{\, (w, \langle t(w) \rangle) \mid w \in \mathcal{T} \,\}$. This means that if σ is a type relabeling, $\overline{G}_{(M,t)} = \overline{G}_{(M,\sigma \circ t)}$. Thus, Lemmas 3.22 and 3.25 give

Lemma 3.26. *Let M be a pure λ-term and t be a type decoration of M under the type environment $x : \gamma_1, \dots, x_n : \gamma_n$ such that (M, t) is in strict η-long form. Let $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$ be a principal typing of M. Then (i, v) and (i', v') are connected in (M, t) if and only if* $\mathrm{subtype}(\alpha_i, v) = \mathrm{subtype}(\alpha_{i'}, v')$.

Moreover, we have

Lemma 3.27. *Let M be a pure λI-term in β-normal form and t be a type decoration of M under the type environment $x : \gamma_1, \dots, x_n : \gamma_n$ such that (M, t) is in η-long form. Let $x_1 : \alpha_1, \dots, x_n : \alpha_n \Rightarrow \alpha_0$ be a principal typing of M. Then (i, v) and (i', v') are related by the transitive closure of the relation of being linked in (M, t) if and only if* $\mathrm{subtype}(\alpha_i, v) = \mathrm{subtype}(\alpha_{i'}, v')$.

Proof. Since $\overline{G}_{(M,t)}$ does not contain any directed cycles, the fact that M is a λI-term implies that every directed path can be extended to one that starts in a node of the form $(i, v, \uparrow)$ and ends with one that ends in a node of the form $(i', v', \downarrow)$. □

The usefulness of the notion of being linked will become clear later.

3.2 Context-free λ-term grammars

A *context-free λ-term grammar* (CFLG) is a quintuple $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$, where $\mathscr{N}$ is a finite alphabet of *nonterminals*, $\Sigma = (A, C, \tau)$ is a higher-order signature, f is a function from $\mathscr{N}$ to $\mathscr{T}(A)$, S is a distinguished member of $\mathscr{N}$, and $\mathscr{P}$ is a finite set of *rules* of the form:

$$B(M) :\!-\ B_1(X_1), \dots, B_n(X_n),$$

where $X_1, \dots, X_n$ are pairwise distinct variables and M is a λ-term in $\Lambda(\Sigma)$ that is in η-long form relative to

$$X_1 : f(B_1), \dots, X_n : f(B_n) \Rightarrow f(B).$$

It is not required that M be in β-normal form. The language of a CFLG $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ is defined in terms of the predicate $\vdash_{\mathscr{G}}$. For a nonterminal $B \in \mathscr{N}$ and a closed λ-term $P \in \Lambda(\Sigma)$,

$$\vdash_{\mathscr{G}} B(P)$$

holds if and only if there exist a rule

$$B(M) :\!-\ B_1(X_1), \dots, B_n(X_n)$$

in $\mathscr{P}$ and closed λ-terms Q_i $(i = 1, \dots, n)$ such that

$$P = M[X_1 := Q_1, \dots, X_n := Q_n],$$
$$\vdash_{\mathscr{G}} B_i(Q_i).$$

When this holds, we have a *derivation tree* for $B(P)$ of the form

$$\begin{array}{c} B(P) \\ \overbrace{T_1 \ \dots \ T_n} \end{array}$$

where T_i is a derivation tree for $B_i(Q_i)$ $(i = 1, \dots, n)$. Note that $\vdash_{\mathscr{G}} B(P)$ implies $\vdash_{\Sigma} P : f(B)$.

The language of $\mathscr{G}$ is

$$L(\mathscr{G}) = \{\, |N|_\beta \mid \vdash_{\mathscr{G}} S(N) \,\}.$$

Thus, the language of a CFLG is a set of closed β-normal λ-terms that are in η-long form relative to a certain type (namely $f(S)$) (cf. Lemmas 3.16 and 3.17).

In example grammars we have given, we have not always adhered to the condition that λ-terms in rules be in η-long form. Any rule with a non-η-long λ-term M should be understood as an abbreviation for the "official" rule that has the η-long form of M instead. The reason that we only allow λ-terms in η-long form in the language of a CFLG is that we do not wish to distinguish between λ-terms that are η-equal.

Example 3.28. The earlier example CFLG (9) in official notation is $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, \mathsf{S})$, where

$$
\begin{aligned}
\mathscr{N} &= \{\mathsf{S}, \mathsf{NP}, \mathsf{VP}, \mathsf{V}, \mathsf{Det}, \mathsf{N}\},\\
\Sigma &= (A, C, \tau),\\
A &= \{e, t\},\\
C &= \{\wedge, \mathbf{John}, \mathbf{find}, \mathbf{catch}, =, \exists, \mathbf{man}, \mathbf{unicorn}\},\\
\tau &= \left\{\begin{array}{r@{}l} \wedge &{}\mapsto t \to t \to t,\\ \mathbf{John} &{}\mapsto e,\\ \mathbf{find} &{}\mapsto e \to e \to t,\\ \mathbf{catch} &{}\mapsto e \to e \to t,\\ = &{}\mapsto e \to e \to t,\\ \exists &{}\mapsto (e \to t) \to t,\\ \mathbf{man} &{}\mapsto e \to t,\\ \mathbf{uncorn} &{}\mapsto e \to t \end{array}\right\},\\
f &= \left\{\begin{array}{r@{}l} \mathsf{S} &{}\mapsto t,\\ \mathsf{NP} &{}\mapsto (e \to t) \to t,\\ \mathsf{VP} &{}\mapsto e \to t,\\ \mathsf{V} &{}\mapsto e \to e \to t,\\ \mathsf{Det} &{}\mapsto (e \to t) \to (e \to t) \to t,\\ \mathsf{N} &{}\mapsto e \to t \end{array}\right\},
\end{aligned}
$$

and $\mathscr{P}$ consists of the following rules:

$$
\begin{aligned}
&\mathsf{S}(X_1(\lambda x.X_2 x)) \mathrel{:-} \mathsf{NP}(X_1), \mathsf{VP}(X_2).\\
&\mathsf{VP}(\lambda x.X_2(\lambda y.X_1 y x)) \mathrel{:-} \mathsf{V}(X_1), \mathsf{NP}(X_2).\\
&\mathsf{V}(\lambda y x.\wedge(X_1 y x)(X_2 y x)) \mathrel{:-} \mathsf{V}(X_1), \mathsf{V}(X_2).\\
&\mathsf{NP}(\lambda u.X_1(\lambda x.X_2 x)(\lambda x.u x)) \mathrel{:-} \mathsf{Det}(X_1), \mathsf{N}(X_2).\\
&\mathsf{NP}(\lambda u.u\,\mathbf{John}).
\end{aligned}
$$

$$\begin{aligned}&\mathsf{V}(\lambda yx.\mathbf{find}\, y\, x).\\&\mathsf{V}(\lambda yx.\mathbf{catch}\, y\, x).\\&\mathsf{V}(\lambda yx.{=}\, y\, x).\\&\mathsf{Det}(\lambda uv.\exists(\lambda y.\wedge(uy)(vy))).\\&\mathsf{N}(\lambda x.\mathbf{man}\, x).\\&\mathsf{N}(\lambda x.\mathbf{unicorn}\, x).\end{aligned}$$

3.3 Datalog programs associated with CFLGs

We associate with a CFLG $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$, where $\Sigma = (A, C, \tau)$, a Datalog program program($\mathscr{G}$), whose set of intensional predicates is $\mathscr{N}$ and whose set of extensional predicates is C. The arity of $B \in \mathscr{N}$ is $|f(B)|$, and the arity of $d \in C$ is $|\tau(d)|$.

In order to facilitate the definition of program($\mathscr{G}$) and the statement of the next lemma, we adopt the following conventions:

Convention 1. If

$$B(M) :- B_1(X_1), \dots, B_n(X_n)$$

is a rule in $\mathscr{P}$, then M is in strict η-long form relative to

$$X_1 : f(B_1), \dots, X_n : f(B_n) \Rightarrow f(B).$$

Convention 2. If

$$B(M) :- B_1(X_1), \dots, B_n(X_n)$$

is a rule in $\mathscr{P}$, then $\overrightarrow{\mathrm{FV}}(M) = (X_1, \dots, X_n)$ and all occurrences of constants in M precede the occurrences of $X_1, \dots, X_n$.

It is easy to transform a rule that does not obey these conventions into an equivalent one that does by changing M to the strict η-long form of $(\lambda X_1 \dots X_n.M)X_1 \dots X_n$, so adopting this convention does not lead to any loss of generality. It is also possible to complicate the definition of program($\mathscr{G}$) and the statement and proof of the lemma to make the following results not depend on the conventions.

We now give the definition of program($\mathscr{G}$), assuming Conventions 1 and 2. Consider a rule

$$\pi = B_0(M) :- B_1(X_1), \dots, B_n(X_n),$$

in $\mathscr{P}$. Let

$$\overrightarrow{\mathrm{Con}}(M) = (d_1, \dots, d_m),$$

and let

$$y_1 : \beta_1, \dots, y_m : \beta_m, X_1 : \alpha_1, \dots, X_n : \alpha_n \Rightarrow \widehat{M}[y_1, \dots, y_m] : \alpha_0$$

be a principal typing of $\widehat{M}[y_1, \dots, y_m]$. (Recall that $\widehat{M}[y_1, \dots, y_m]$ is a pure λ-term such that $\widehat{M}[d_1, \dots, d_m] = M$.) Note that by Convention 2, $\overrightarrow{\mathrm{FV}}(\widehat{M}[y_1, \dots, y_m]) = (y_1, \dots, y_m, X_1, \dots, X_n)$. By Convention 1 and Remark 3.23,

$$\begin{aligned} \langle \beta_i \rangle &= \langle \tau(d_i) \rangle \quad \text{for } i = 1, \dots, m, \\ \langle \alpha_i \rangle &= \langle f(B_i) \rangle \quad \text{for } i = 0, \dots, n. \end{aligned} \tag{38}$$

The Datalog rule ρ_π corresponding to π is defined as

$$B_0(\overline{\alpha_0}) :\!-\ d_1(\overline{\beta_1}), \dots, d_m(\overline{\beta_m}), B_1(\overline{\alpha_1}), \dots, B_n(\overline{\alpha_n}),$$

where atomic types in $\overline{\alpha_i}, \overline{\beta_i}$ are considered Datalog variables. Clearly, ρ_π does not depend on the choice of variables $y_1, \dots, y_m$. Also, the choice of atomic types in $\overline{\alpha_i}, \overline{\beta_i}$ is immaterial. So it does not matter which principal typing of $\widehat{M}[y_1, \dots, y_m]$ we use.[32]

The Datalog program associated with $\mathscr{G}$ is defined as

$$\mathrm{program}(\mathscr{G}) = \{\, \rho_\pi \mid \pi \in \mathscr{P} \,\}.$$

Remark 3.29.

$$B_0(\vec{s}_0) :\!-\ d_1(\vec{t}_1), \dots, d_m(\vec{t}_m), B_1(\vec{s}_1), \dots, B_n(\vec{s}_n)$$

is an instance of ρ_π if and only if

$$\begin{aligned} \vdash y_1 : \langle \tau(d_1) \rangle(\vec{t}_1), \dots, y_m : \langle \tau(d_m) \rangle(\vec{t}_m), X_1 : \langle f(B_1) \rangle(\vec{s}_1), \dots, X_n : \langle f(B_n) \rangle(\vec{s}_n) \\ \Rightarrow \widehat{M}[y_1, \dots, y_m] : \langle f(B_0) \rangle(\vec{s}_0). \end{aligned}$$

Example 3.30. For the CFLG $\mathscr{G}$ of Example 3.28, program($\mathscr{G}$) consists of the rules in (16) in Section 2.2. For example, let M_3 be the λ-term in the third rule π_3 of $\mathscr{G}$. We have $\overrightarrow{\mathrm{Con}}(M_3) = (\wedge)$, and the following is a principal typing of $\widehat{M_3}[z_1]$:

$$z_1 : i_2 \to i_5 \to i_1, X_1 : i_3 \to i_4 \to i_2, X_2 : i_3 \to i_4 \to i_5 \Rightarrow \lambda yx.z_1(X_1 yx)(X_2 yx) : i_3 \to i_4 \to i_1.$$

[32]This definition of ρ_π is applicable to arbitrary CFLG rules satisfying Conventions 1 and 2. When M is almost linear, the definition of ρ_π given here is equivalent to the definition given in Section 2.2 in terms of the hypergraph representation graph(M) of a principal typing of M.

We thus obtain

$$\rho_{\pi_3} = \mathsf{V}(i_1, i_4, i_3) \mathrel{:-} \wedge(i_3, i_5, i_2), \mathsf{V}(i_2, i_4, i_3), \mathsf{V}(i_5, i_4, i_3).$$

Note that the hypergraph representation (15) of M_3 encodes the same information as its principal typing.

The following is a key fact about program($\mathscr{G}$) that holds of any CFLG $\mathscr{G}$ satisfying Conventions 1 and 2. It basically says that under the correspondence between π and ρ_π defined above, a CFLG derivation tree plus a typing (of a certain kind) for the associated λ-term corresponds to a Datalog derivation tree, and vice versa. Its proof is quite straightforward, if rather tedious. If $\vec{u}$ is a tuple (sequence) of constants, we let $|\vec{u}|$ denote its length, i.e., the number of its components.

Lemma 3.31. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ with $\Sigma = (A, C, \tau)$ be a CFLG, and let U be some set of constants. Let $e_1, \dots, e_l \in C$, $B \in \mathscr{N}$, and $\vec{u}_1, \dots, \vec{u}_l, \vec{s}$ be sequences of constants from U such that $|\vec{u}_i| = |\tau(e_i)|$ and $|\vec{s}| = |f(B)|$. The following are equivalent:*

(i) *There exists $P \in \Lambda(\Sigma)$ such that*

$$\vdash_{\mathscr{G}} B(P),$$
$$\overrightarrow{\mathrm{Con}}(P) = (e_1, \dots, e_l),$$
$$\vdash z_1 : \langle \tau(e_1) \rangle(\vec{u}_1), \dots, z_l : \langle \tau(e_l) \rangle(\vec{u}_l) \Rightarrow \widehat{P}[z_1, \dots, z_l] : \langle f(B) \rangle(\vec{s}).$$

(ii) *There exists a derivation tree T for*

$$\mathrm{program}(\mathscr{G}) \cup \{\, e_i(\vec{u}_i) \mid 1 \le i \le l \,\} \vdash B(\vec{s})$$

such that $(e_1(\vec{u}_1), \dots, e_l(\vec{u}_l))$ lists the labels of the extensional nodes of T in the order from left to right.

Proof. (i) $\Rightarrow$ (ii). Induction on the derivation of $\vdash_{\mathscr{G}} B(P)$. Assume that $\vdash_{\mathscr{G}} B(P)$ is inferred from

$$\vdash_{\mathscr{G}} B_i(P_i) \quad (i = 1, \dots, n)$$

using a rule

$$\pi = B(M) \mathrel{:-} B_1(X_1), \dots, B_n(X_n)$$

such that

$$P = M[X_1 := P_1, \dots, X_n := P_n].$$

Let $m = |\overrightarrow{\mathrm{Con}}(M)|$ and $l_i = |\overrightarrow{\mathrm{Con}}(P_i)|$ for $i = 1, \dots, n$. Then $l = m + l_1 + \cdots + l_n$ and

$$\begin{aligned}\widehat{P}[z_1, \dots, z_l] &= \widehat{M}[z_1, \dots, z_m] \\ &\qquad [X_1 := \widehat{P_1}[z_{h(1,1)}, \dots, z_{h(1,l_1)}], \dots, X_n := \widehat{P_n}[z_{h(n,1)}, \dots, z_{h(n,l_n)}]], \\ \overrightarrow{\mathrm{Con}}(M) &= (e_1, \dots, e_m), \\ \overrightarrow{\mathrm{Con}}(P_i) &= (e_{h(i,1)}, \dots, e_{h(i,l_i)}),\end{aligned}$$

where

$$h(i,j) = m + l_1 + \cdots + l_{i-1} + j.$$

By assumption,

$$\vdash z_1 : \langle \tau(e_1) \rangle (\vec{u}_1), \dots, z_l : \langle \tau(e_l) \rangle (\vec{u}_l) \Rightarrow \widehat{P}[z_1, \dots, z_l] : \langle f(B) \rangle (\vec{s}). \tag{39}$$

By Lemma 3.13, any type decoration for (39) splits into type decorations for

$$\begin{aligned}&z_1 : \langle \tau(e_1) \rangle (\vec{u}_1), \dots, z_m : \langle \tau(e_m) \rangle (\vec{u}_m), X_1 : \alpha_1, \dots, X_n : \alpha_n \\ &\qquad\qquad \Rightarrow \widehat{M}[z_1, \dots, z_m] : \langle f(B) \rangle (\vec{s})\end{aligned}$$

and

$$\begin{aligned}&z_{h(i,1)} : \langle \tau(e_{h(i,1)}) \rangle (\vec{u}_{h(i,1)}), \dots, z_{h(i,l_i)} : \langle \tau(e_{h(i,l_i)}) \rangle (\vec{u}_{h(i,l_i)}) \\ &\qquad\qquad \Rightarrow \widehat{P_i}[z_{h(i,1)}, \dots, z_{h(i,l_i)}] : \alpha_i\end{aligned} \tag{40}$$

$(i = 1, \dots, n)$. In order to apply the induction hypothesis to (40), we need

$$\langle \alpha_i \rangle = \langle f(B_i) \rangle \quad \text{for each } i = 1, \dots, n. \tag{41}$$

If $\widehat{P}[z_1, \dots, z_l]$ is not λI, type decorations for (39) need not be unique, and indeed there may be one for which (41) fails. We show that a desirable type decoration for (39) can be obtained from a principal typing of $\widehat{P}[z_1, \dots, z_l]$ by type relabeling.

Since Convention 1 ensures that P_i is in strict η-long form relative to $f(B_i)$ and M is in strict η-long form relative to $X_1 : f(B_1), \dots, X_n : f(B_n) \Rightarrow f(B)$, it follows that $\widehat{P_i}[z_{h(i,1)}, \dots, z_{h(i,l_i)}]$ is in strict η-long form relative to $z_{h(i,1)} : \tau(e_{h(i,1)}), \dots, z_{h(i,l_i)} : \tau(e_{h(i,l_i)}) \Rightarrow f(B_i)$ and $\widehat{M}[z_1, \dots, z_m]$ is in strict η-long form relative to $z_1 : \tau(e_1), \dots, z_m : \tau(e_m), X_1 : f(B_1), \dots, X_n : f(B_n) \Rightarrow f(B)$. By Lemma 3.21, there is a type decoration $t_{\widehat{P}[z_1, \dots, z_l]}$ for

$$z_1 : \tau(e_1), \dots, z_l : \tau(e_l) \Rightarrow \widehat{P}[z_1, \dots, z_l] : f(B)$$

that is obtained by combining the type decoration for

$$z_1 : \tau(e_1), \dots, z_m : \tau(e_m), X_1 : f(B_1), \dots, X_n : f(B_n) \Rightarrow \widehat{M}[z_1, \dots, z_m] : f(B)$$

and the type decoration for

$$z_{h(i,1)} : \tau(e_{h(i,1)}), \dots, z_{h(i,l_i)} : \tau(e_{h(i,l_i)}) \Rightarrow \widehat{P_i}[z_{h(i,1)}, \dots, z_{h(i,l_i)}] : f(B_i)$$

for $i = 1, \dots, n$, such that $(\widehat{P}[z_1, \dots, z_l], t_{\widehat{P}[z_1,\dots,z_l]})$ is in strict η-long form.

Let $\widetilde{t}_{\widehat{P}[z_1,\dots,z_l]}$ be a principal type decoration for $\widehat{P}[z_1, \dots, z_l]$ with the associated principal typing

$$z_1 : \delta_1, \dots, z_l : \delta_l \Rightarrow \gamma.$$

By Lemma 3.13, $\widetilde{t}_{\widehat{P}[z_1,\dots,z_l]}$ splits into type decorations for

$$z_1 : \delta_1, \dots, z_m : \delta_m, X_1 : \gamma_1, \dots, X_n : \gamma_n \Rightarrow \widehat{M}[z_1, \dots, z_m] : \gamma \tag{42}$$

and

$$z_{h(i,1)} : \delta_{h(i,1)}, \dots, z_{h(i,l_i)} : \delta_{h(i,l_i)} \Rightarrow \widehat{P_i}[z_{h(i,1)}, \dots, z_{h(i,l_i)}] : \gamma_i \quad (i = 1, \dots, n). \tag{43}$$

By Lemma 3.22, we must have

$$\begin{aligned} \langle \delta_i \rangle &= \langle \tau(e_i) \rangle \quad \text{for } i = 1, \dots, l, \\ \langle \gamma \rangle &= \langle f(B) \rangle, \\ \langle \gamma_i \rangle &= \langle f(B_i) \rangle \quad \text{for } i = 1, \dots, n. \end{aligned}$$

By (39), there is a type substitution σ such that

$$\begin{aligned} \delta_i \sigma &= \langle \tau(e_i) \rangle (\vec{u}_i), \\ \gamma \sigma &= \langle f(B) \rangle (\vec{s}) \end{aligned}$$

that leaves atomic types that do not appear in $\delta_1, \dots, \delta_l, \gamma$ unchanged. Then σ is a type relabeling, and there are sequences $\vec{s}_1, \dots, \vec{s}_n$ of atomic types such that

$$\gamma_i \sigma = \langle f(B_i) \rangle (\vec{s}_i) \quad \text{for } i = 1, \dots, n.$$

Without loss of generality, we may assume that $\vec{s}_1, \dots, \vec{s}_n$ are sequences of constants from U. (Otherwise we may replace any constants not in U by constants in U.)

Applying σ to (42) and (43), we get

$$\vdash z_1 : \langle \tau(e_1) \rangle (\vec{u}_1), \dots, z_m : \langle \tau(e_m) \rangle (\vec{u}_m), X_1 : \langle f(B_1) \rangle (\vec{s}_1), \dots, X_n : \langle f(B_n) \rangle (\vec{s}_n) \\ \Rightarrow \widehat{M}[z_1, \dots, z_l] : \langle f(B) \rangle (\vec{s}), \quad (44)$$

and

$$\vdash z_{h(i,1)} : \langle \tau(e_{h(i,1)}) \rangle (\vec{u}_{h(i,1)}), \dots, z_{h(i,l_i)} : \langle \tau(e_{h(i,l_i)}) \rangle (\vec{u}_{h(i,l_i)}) \\ \Rightarrow \widehat{P_i}[z_{h(i,1)}, \dots, z_{h(i,l_i)}] \langle f(B_i) \rangle (\vec{s}_i) \quad (45)$$

for $i = 1, \dots, n$.

By (45), the induction hypothesis applies to P_i, giving a Datalog derivation tree T_i for

$$\text{program}(\mathscr{G}) \cup \{\, e_{h(i,j)}(\vec{u}_{h(i,j)}) \mid 1 \le j \le l_i \,\} \vdash B_i(\vec{s}_i) \quad (46)$$

such that $(e_{h(i,1)}(\vec{u}_{h(i,1)}), \dots, e_{h(i,l_i)}(\vec{u}_{h(i,l_i)}))$ lists the labels of the extensional nodes of T_i from left to right.

By (44) and Remark 3.29,

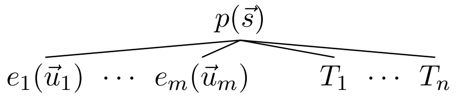
$$B(\vec{s}) :\!-\; e_1(\vec{u}_1), \dots, e_m(\vec{u}_m), B_1(\vec{s}_1), \dots, B_n(\vec{s}_n) \quad (47)$$

is an instance of ρ_π. Combining (46) and (47), we obtain a Datalog derivation tree T for

$$\text{program}(\mathscr{G}) \cup \{\, e_i(\vec{u}_i) \mid 1 \le i \le l \,\} \vdash B(\vec{s}),$$

such that $(e_1(\vec{u}_1), \dots, e_l(\vec{u}_l))$ lists the labels of the extensional nodes of T from left to right.

(ii) ⇒ (i). Induction on T. Assume that T is of the form



and the root node of T is obtained by an application of an instance

$$B(\vec{s}) :\!-\; e_1(\vec{u}_1), \dots, e_m(\vec{u}_m), B_1(\vec{s}_1), \dots, B_n(\vec{s}_n)$$

of some ρ_π, where $m \le l$ and

$$\pi = B(M) :\!-\; B_1(X_1), \dots, B_n(X_n), \quad (48)$$

$$\overrightarrow{\text{Con}}(M) = (e_1, \dots, e_m). \quad (49)$$

Let l_i be the number of extensional nodes of T_i. Then $l = m + l_1 + \cdots + l_n$, and for $i = 1, \ldots, n$, T_i is a derivation tree for

$$\text{program}(\mathscr{G}) \cup \{e_{h(i,1)}(\vec{u}_{h(i,1)}), \ldots, e_{h(i,l_i)}(\vec{u}_{h(i,l_i)})\} \vdash B_i(\vec{s}_i),$$

where $h(i,j) = m + l_1 + \cdots + l_{i-1} + j$, and $(e_{h(i,1)}(\vec{u}_{h(i,1)}), \ldots, e_{h(i,l_i)}(\vec{u}_{h(i,l_i)}))$ lists the labels of the extensional nodes of T_i from left to right.

By Remark 3.29, we have

$$\begin{aligned} &\vdash z_1 : \langle \tau(e_1) \rangle(\vec{u}_1), \ldots, z_m : \langle \tau(e_m) \rangle(\vec{u}_m), X_1 : \langle f(B_1) \rangle(\vec{s}_1), \ldots, X_n : \langle f(B_n) \rangle(\vec{s}_n) \\ &\qquad\qquad\qquad\qquad \Rightarrow \widehat{M}[z_1, \ldots, z_m] : \langle f(B) \rangle(\vec{s}). \qquad (50) \end{aligned}$$

By induction hypothesis, for $i = 1, \ldots, n$, there exists $P_i \in \Lambda(\Sigma)$ such that

$$\vdash_{\mathscr{G}} B_i(P_i), \qquad (51)$$

$$\overrightarrow{\text{Con}}(P_i) = (e_{h(i,1)}, \ldots, e_{h(i,l_i)}), \qquad (52)$$

and

$$\begin{aligned} &\vdash z_{h(i,1)} : \langle \tau(e_{h(i,1)}) \rangle(\vec{u}_{h(i,1)}), \ldots, z_{h(i,l_i)} : \langle \tau(e_{h(i,l_i)}) \rangle(\vec{u}_{h(i,l_i)}) \\ &\qquad\qquad\qquad\qquad \Rightarrow \widehat{P_i}[z_{h(i,1)}, \ldots, z_{h(i,l_i)}] : \langle f(B_i) \rangle(\vec{s}_i). \qquad (53) \end{aligned}$$

Let

$$P = M[X_1 := P_1, \ldots, X_n := P_n].$$

Then by (48), (51), (49), and (52),

$$\begin{aligned} &\vdash_{\mathscr{G}} B(P), \\ \overrightarrow{\text{Con}}(P) &= (e_1, \ldots, e_m, e_{h(1,1)}, \ldots, e_{h(1,l_1)}, \ldots, e_{h(n,1)}, \ldots, e_{h(n,l_n)}) \\ &= (e_1, \ldots, e_l). \end{aligned}$$

We have

$$\begin{aligned} &\widehat{P}[z_1, \ldots, z_l] = \\ &\quad \widehat{M}[z_1, \ldots, z_m][X_1 := \widehat{P_1}[z_{h(1,1)}, \ldots, z_{h(1,l_1)}], \ldots, X_n := \widehat{P_n}[z_{h(n,1)}, \ldots, z_{h(n,l_n)}]]. \end{aligned}$$

By Lemma 3.12 applied to (50) and (53), we get

$$\vdash z_1 : \langle \tau(e_1) \rangle(\vec{u}_1), \ldots, z_m : \langle \tau(e_m) \rangle(\vec{u}_m),$$

$$
\begin{aligned}
&z_{h(1,1)} : \langle\tau(e_{h(1,1)})\rangle(\vec{u}_{h(1,1)}), \dots, z_{h(1,l_1)} : \langle\tau(e_{h(1,l_1)})\rangle(\vec{u}_{h(1,l_1)}),\\
&\qquad\qquad\qquad\qquad\vdots\\
&z_{h(n,1)} : \langle\tau(e_{h(n,1)})\rangle(\vec{u}_{h(n,1)}), \dots, z_{h(n,l_n)} : \langle\tau(e_{h(n,l_n)})\rangle(\vec{u}_{h(n,l_n)})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \widehat{P}[z_1, \dots, z_l] : \langle f(B)\rangle(\vec{s}).
\end{aligned}
$$

Hence P satisfies the required properties. □

Example 3.32. Let $\mathscr{G}$ be the CFLG of Example 3.28. Let

$$
\begin{aligned}
P = \; &(\lambda u.u\,\mathbf{John})(\lambda x.\\
&\quad(\lambda x.\\
&\qquad(\lambda u.\\
&\qquad\quad(\lambda uv.\exists(\lambda y.\wedge(uy)(vy)))\\
&\qquad(\lambda x.\\
&\qquad\quad(\lambda x.\mathbf{unicorn}\,x)\\
&\qquad x)(\lambda x.ux))\\
&\quad(\lambda y.\\
&\qquad(\lambda yx.\mathbf{find}\,y\,x)\\
&\quad y\,x))\\
&x).
\end{aligned}
$$

Then $\vdash_{\mathscr{G}} \mathsf{S}(P)$. The derivation tree for $\mathsf{S}(P)$ (in abbreviated notation) was shown in Figure 5 in Section 2.2. We have $\overrightarrow{\mathrm{Con}}(P) = (\mathbf{John}, \exists, \wedge, \mathbf{unicorn}, \mathbf{find})$ and

$$
\begin{aligned}
\widehat{P}[z_1, z_2, z_3, z_4, z_5] = \; &(\lambda u.uz_1)(\lambda x.\\
&\quad(\lambda x.\\
&\qquad(\lambda u.\\
&\qquad\quad(\lambda uv.z_2(\lambda y.z_3(uy)(vy)))\\
&\qquad(\lambda x.\\
&\qquad\quad(\lambda x.z_4x)\\
&\qquad x)(\lambda x.ux))\\
&\quad(\lambda y.\\
&\qquad(\lambda yx.z_5yx)\\
&\quad yx))\\
&x).
\end{aligned}
$$

By one direction of Lemma 3.31, whenever we have

$$\begin{aligned}&\vdash z_1 : u_{1,1}, z_2 : (u_{2,3} \to u_{2,2}) \to u_{2,1},\\ &\qquad z_3 : u_{3,3} \to u_{3,2} \to u_{3,1}, z_4 : u_{4,2} \to u_{4,1}, z_5 : u_{5,3} \to u_{5,2} \to u_{5,1}\\ &\qquad\qquad \Rightarrow \widehat{P}[z_1, z_2, z_3, z_4, z_5] : s, \end{aligned} \tag{54}$$

we must have

$$\begin{aligned}&\text{program}(\mathscr{G}) \cup \{\mathbf{John}(u_{1,1}), \exists(u_{2,1}, u_{2,2}, u_{2,3}), \wedge(u_{3,1}, u_{3,2}, u_{3,3}),\\ &\qquad \mathbf{unicorn}(u_{4,1}, u_{4,2}), \mathbf{find}(u_{5,1}, u_{5,2}, u_{5,3})\} \vdash \mathsf{S}(s). \end{aligned} \tag{55}$$

The Datalog derivation tree for (55) will have the same shape as the one in Figure 6 in Section 2.2. Conversely, whenever (55) has a derivation tree of this shape, we must have (54), by (the proof of) the other direction of Lemma 3.31

Let $\Sigma = (A, C, \tau)$ be a higher-order signature and U be some set of database constants. We write $\mathcal{D}_{\Sigma,U}$ for the database schema (C, U), where each $d \in C$ has arity $|\tau(d)|$. Let D be a database over $\mathcal{D}_{\Sigma,U}$ and $\alpha \in \mathscr{T}(A)$. We define a set $\Lambda(D, \alpha)$ of closed λ-terms over Σ as follows:

$$\Lambda(D,\alpha) = \left\{ M \in \Lambda(\Sigma) \;\middle|\; \begin{array}{l} \mathrm{FV}(M) = \varnothing, \overrightarrow{\mathrm{Con}}(M) = (d_1, \dots, d_n), \{d_1(\vec{s}_1), \dots, d_n(\vec{s}_n)\} \subseteq D, \\ \vdash z_1 : \langle \tau(d_1) \rangle(\vec{s}_1), \dots, z_n : \langle \tau(d_n) \rangle(\vec{s}_n) \Rightarrow \widehat{M}[z_1, \dots, z_n] : \alpha \end{array} \right\}.$$

Example 3.33. Let Σ' be the extension of the higher-order signature Σ in Example 3.28 with an additional constant $\neg$ of type $t \to t$. Let $U = \{a, b, 0, 1\}$, and consider the following database D over $\mathcal{D}_{\Sigma',U}$:

$$\begin{gathered}\mathbf{man}(1,a), \quad \mathbf{man}(0,b), \quad \mathbf{unicorn}(0,a), \quad \mathbf{unicorn}(1,b),\\ \wedge(1,1,1), \quad \wedge(0,1,0), \quad \wedge(0,0,1), \quad \wedge(0,0,0), \quad \neg(0,1), \quad \neg(1,0),\\ \exists(1,1,a), \quad \exists(1,1,b).\end{gathered}$$

The set $\Lambda(D, 1)$ contains, e.g.,

$$\begin{gathered}\wedge(\exists(\lambda x.\mathbf{man}\, x))(\exists(\lambda y.\mathbf{unicorn}\, y)),\\ \exists(\lambda x.\wedge(\mathbf{man}\, x)(\neg(\mathbf{unicorn}\, x))),\end{gathered}$$

but not, e.g.,

$$\exists(\lambda x.\wedge(\mathbf{man}\, x)(\mathbf{unicorn}\, x)).$$

Lemma 3.34. *Let $M, M' \in \Lambda(\Sigma)$.*

(i) *If $M \twoheadrightarrow_\beta M'$, then $M \in \Lambda(D, \alpha)$ implies $M' \in \Lambda(D, \alpha)$.*

(ii) *If $M' \twoheadrightarrow_\beta M$ by non-erasing non-duplicating β-reduction, then $M \in \Lambda(D, \alpha)$ implies $M' \in \Lambda(D, \alpha)$.*

Proof. Let $M \in \Lambda(D, \alpha)$ and $\overrightarrow{\mathrm{Con}}(M) = (d_1, \dots, d_n)$. By the definition of $\Lambda(D, \alpha)$, for some $\vec{s}_1, \dots, \vec{s}_n$ such that $\{d_1(\vec{s}_1), \dots, d_n(\vec{s}_n)\} \subseteq D$,

$$\vdash z_1 : \langle \tau(d_1) \rangle(\vec{s}_1), \dots, z_1 : \langle \tau(d_n) \rangle(\vec{s}_n) \Rightarrow \widehat{M}[z_1, \dots, z_n] : \alpha.$$

(i). Suppose $M \twoheadrightarrow_\beta M'$. Let $m = |\overrightarrow{\mathrm{Con}}(M')|$ and $g \colon \{1, \dots, m\} \to \{1, \dots, n\}$ be the function such that the ith occurrence of a constant in M' is a descendant of the $g(i)$th occurrence of a constant in M. Then $\overrightarrow{\mathrm{Con}}(M') = (d_{g(1)}, \dots, d_{g(m)})$ and

$$\widehat{M}[z_1, \dots, z_n] \twoheadrightarrow_\beta \widehat{M'}[z_{g(1)}, \dots, z_{g(m)}].$$

By the Subject Reduction Theorem (Theorem 3.14),

$$\vdash \{\, z_{g(i)} : \langle \tau(d_{g(i)}) \rangle(\vec{s}_{g(i)}) \mid 1 \le i \le m \,\} \Rightarrow \widehat{M'}[z_{g(1)}, \dots, z_{g(m)}] : \alpha,$$

and thus

$$\vdash y_1 : \langle \tau(d_{g(1)}) \rangle(\vec{s}_{g(1)}), \dots, y_m : \langle \tau(d_{g(m)}) \rangle(\vec{s}_{g(m)}) \Rightarrow \widehat{M'}[y_1, \dots, y_m] : \alpha.$$

This shows $M' \in \Lambda(D, \alpha)$.

(ii). Suppose $M' \twoheadrightarrow_\beta M$ by non-erasing non-duplicating β-reduction. Then $|\overrightarrow{\mathrm{Con}}(M')| = n$ and there is a permutation g of $\{1, \dots, n\}$ such that the ith occurrence of a constant in M' is the ancestor of the $g(i)$th occurrence of a constant in M. We have $\overrightarrow{\mathrm{Con}}(M') = (d_{g(1)}, \dots, d_{g(n)})$ and $\widehat{M'}[z_{g(1)}, \dots, z_{g(n)}] \twoheadrightarrow_\beta \widehat{M}[z_1, \dots, z_n]$ by non-erasing non-duplicating β-reduction. By the Subject Expansion Theorem (Theorem 3.15),

$$\vdash z_1 : \langle \tau(d_1) \rangle(\vec{s}_1), \dots, z_n : \langle \tau(d_n) \rangle(\vec{s}_n) \Rightarrow \widehat{M'}[z_{g(1)}, \dots, z_{g(n)}] : \alpha.$$

Therefore, $M' \in \Lambda(D, \alpha)$. $\square$

The next lemma is an immediate consequence of Lemma 3.31:

Lemma 3.35. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be a CFLG. Let U be some set of database constants, D be a database over $\mathcal{D}_{\Sigma,U}$, and $\vec{s}$ be a sequence of constants from U such that $|\vec{s}| = |f(S)|$. The following are equivalent:*

$$\boxed{\begin{array}{ccc} \{\, P \in \Lambda(\Sigma) \mid \vdash_{\mathscr{G}} S(P) \,\} \cap \Lambda(D, \langle f(S) \rangle(\vec{s})) \neq \varnothing & \Longleftrightarrow & \text{program}(\mathscr{G}) \cup D \vdash S(\vec{s}) \\ \Downarrow & & \\ L(\mathscr{G}) \cap \Lambda(D, \langle f(S) \rangle(\vec{s})) \neq \varnothing & & \end{array}}$$

Figure 12: A general property of program($\mathscr{G}$).

(i) *There exists some* $P \in \Lambda(D, \langle f(S) \rangle(\vec{s}))$ *such that* $\vdash_{\mathscr{G}} S(P)$.

(ii) $\text{program}(\mathscr{G}) \cup D \vdash S(\vec{s})$.

Lemma 3.36. *Let* $\mathscr{G}, U, D, \vec{s}$ *be as in Lemma 3.35. If* $\text{program}(\mathscr{G}) \cup D \vdash S(\vec{s})$, *then* $L(\mathscr{G}) \cap \Lambda(D, \langle f(S) \rangle(\vec{s})) \neq \varnothing$.

Proof. By Lemma 3.34, part (i) and Lemma 3.35. □

See Figure 12. The converse of Lemma 3.36 does not hold in general, but we shall see below some special cases where it does hold (Theorems 3.40, 3.65, and 4.3).

3.4 Databases determined by λ-terms

Let $M \in \Lambda(\Sigma)$ be a closed λ-term in strict η-long form relative to γ such that $\overrightarrow{\text{Con}}(M) = (d_1, \dots, d_m)$. Define

$$\begin{aligned} \text{database}(M) &= \{\, d_i(\overline{\beta_i}) \mid 1 \leq i \leq m \,\}, \\ \text{tuple}(M) &= \overline{\alpha} \end{aligned}$$

where

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \widehat{M}[y_1, \dots, y_m] : \alpha$$

is a principal typing of $\widehat{M}[y_1, \dots, y_m]$.[33] Here, atomic types that occur in $\beta_1, \dots, \beta_m, \alpha$ are regarded as database constants. Note that by Remark 3.23, $\langle \gamma \rangle(\text{tuple}(M)) = \alpha$ and $\Lambda(\text{database}(M), \langle \gamma \rangle(\text{tuple}(M)))$ is well-defined. The following is obvious from the above definition:

Lemma 3.37. *If* $M \in \Lambda(\Sigma)$ *is a closed* λ*-term in strict* η*-long form relative to* γ, *then* $M \in \Lambda(\text{database}(M), \langle \gamma \rangle(\text{tuple}(M)))$.

[33]When M is almost linear, the definition of $(\text{database}(M), \text{tuple}(M))$ here is equivalent to the definition in terms of graph(M) given in Section 2.2.

Note that if M is in strict η-long form relative to γ, then $|M|_\beta$ is in η-long form relative to γ and belongs to $\Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ (Lemma 3.34). We shall see below that in some special cases $|M|_\beta$ is the only η-long β-normal λ-term in $\Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ (Lemmas 3.41 and 3.54).

Example 3.38. Consider the λ-term (8) from Sections 2.1–2.2:

$$M = \exists(\lambda y.\wedge(\mathbf{unicorn}\, y)(\mathbf{find}\, y\, \mathbf{John})).$$

Using the principle typing

$$z_1 : (4 \to 2) \to 1, z_2 : 3 \to 5 \to 2, z_3 : 4 \to 3, z_4 : 4 \to 6 \to 5, z_5 : 6 \Rightarrow z_1(\lambda y.z_2(z_3 y)(z_4 y z_5)) : 1$$

of $\widehat{M}[z_1, z_2, z_3, z_4, z_5]$, we obtain

$$\begin{aligned} \text{database}(M) &= \{\exists(1,2,4), \wedge(2,5,3), \mathbf{unicorn}(3,4), \mathbf{find}(5,6,4), \mathbf{John}(6)\}, \\ \text{tuple}(M) &= (1). \end{aligned}$$

Lemma 3.54 below implies that M is the only λ-term in $\Lambda(\text{database}(M), 1)$ that is in η-long β-normal form relative to the type t.

3.5 From CFLGs to Datalog: The case of linear CFLGs

We first treat the special case of linear CFLGs because the reduction to Datalog as well as the proof of its correctness can be made much simpler in this case than in the more general case of almost linear CFLGs.

The crucial property is the following:

Lemma 3.39. *Let $\Sigma = (A, C, \tau)$ be a higher-order signature, U be a set of database constants, D be a database over $\mathcal{D}_{\Sigma,U}$, and $\alpha \in \mathscr{T}(A)$. For every linear closed λ-term $M \in \Lambda(\Sigma)$, $M \in \Lambda(D, \alpha)$ if and only if $|M|_\beta \in \Lambda(D, \alpha)$.*

Proof. The "only if" direction is by Lemma 3.34, part (i). Since the β-reduction $M \twoheadrightarrow_\beta |M|_\beta$ must be non-erasing and non-duplicating, the "if" direction follows by part (ii) of the same lemma. ☐

Theorem 3.40. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be a linear CFLG. Let U be some set of database constants, D be a database over $\mathcal{D}_{\Sigma,U}$, and $\vec{s}$ be a sequence of constants from U such that $|\vec{s}| = |f(S)|$. The following are equivalent:*

(i) $L(\mathscr{G}) \cap \Lambda(D, \langle f(S)\rangle(\vec{s})) \neq \varnothing$.

(ii) $\text{program}(\mathscr{G}) \cup D \vdash S(\vec{s})$.

Proof. In view of Lemmas 3.35 and 3.36, it suffices to show that $\vdash_{\mathscr{G}} S(P)$ and $|P|_\beta \in \Lambda(D, \langle f(S)\rangle(\vec{s}))$ imply $P \in \Lambda(D, \langle f(S)\rangle(\vec{s}))$. But this is immediate from Lemma 3.39, since $\vdash_{\mathscr{G}} S(P)$ implies that P is linear. □

Lemma 3.41. *If M is an affine λ-term in strict η-long form relative to γ, then $|M|_\beta$ is the only λ-term in $\Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ that is in η-long β-normal form relative to γ.*

Proof. Let $\overrightarrow{\text{Con}}(M) = (d_1, \dots, d_m)$, and let

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \alpha \tag{56}$$

be a principal typing of $\widehat{M}[y_1, \dots, y_m]$. Then database$(M) = \{\, d_i(\overline{\beta_i}) \mid 1 \le i \le m \,\}$. Note that $\widehat{M}[y_1, \dots, y_m]$ is a pure affine λ-term. By Theorem 3.19, (56) is a balanced typing.

We know from Lemmas 3.17, 3.34 and 3.37 that $|M|_\beta$ is in η-long form relative to γ and that $|M|_\beta \in \Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$. Suppose $M' \in \Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ and $|\overrightarrow{\text{Con}}(M')| = n$. Then there is a function $g : \{1, \dots, n\} \to \{1, \dots, m\}$ such that $\overrightarrow{\text{Con}}(M') = (d_{g(1)}, \dots, d_{g(n)})$ and

$$\vdash z_1 : \beta_{g(1)}, \dots, z_n : \beta_{g(n)} \Rightarrow \widehat{M'}[z_1, \dots, z_n] : \alpha.$$

Substituting $y_{g(i)}$ for z_i, we get

$$\vdash \{\, y_{g(i)} : \beta_{g(i)} \mid 1 \le i \le n \,\} \Rightarrow \widehat{M'}[y_{g(1)}, \dots, y_{g(n)}] : \alpha.$$

By the Coherence Theorem (Theorem 3.18), $\widehat{M'}[y_{g(1)}, \dots, y_{g(n)}] =_{\beta\eta} \widehat{M}[y_1, \dots, y_n]$, and so $M' =_{\beta\eta} M$. It follows that if M' is in η-long β-normal form relative to γ, then $M' = |M|_\beta$. □

Theorem 3.42. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be a linear CFLG. Suppose that $N \in \Lambda(\Sigma)$ is a linear λ-term in η-long β-normal form relative to $f(S)$. Then the following are equivalent:*

(i) $N \in L(\mathscr{G})$.

(ii) program$(\mathscr{G}) \cup$ database$(N) \vdash S(\text{tuple}(N))$.

Proof. Immediate from Lemma 3.41 and Theorem 3.40. □

Let us analyze the computational complexity of this reduction.

Lemma 3.43. *Given a linear λ-term $N \in \Lambda(\Sigma)$ in η-long β-normal form relative to $f(S)$, the pair* $(\text{database}(N), S(\text{tuple}(N)))$ *can be computed by a deterministic log-space-bounded Turing machine.*

Proof (sketch). We sketch how a deterministic log-space-bounded Turing machine $\mathscr{M}$ that has multiple heads on the input tape can compute $(\text{database}(N), S(\text{tuple}(N)))$, relying on the fact that an extra head can be simulated by a log-space-bounded work tape. We assume that the input λ-term N is given in the form of a λ-expression; the λ symbol, parentheses, and constants in N are represented by individual symbols of the input alphabet of $\mathscr{M}$, and variables are represented by strings of the form $\mathtt{v}l$, where $\mathtt{v}$ is a special symbol and l is a natural number written in binary (i.e., a string over $\{\mathtt{0},\mathtt{1}\}$).

Let $\overrightarrow{\text{Con}}(N) = (c_1, \dots, c_n)$. The output of $\mathscr{M}$ will be of the form

$$c_1(k_{1,1}, \dots, k_{1,r_1}), \dots, c_n(k_{n,1}, \dots, k_{n,r_n}), S(k_{0,1}, \dots, k_{0,r_0}),$$

where $r_i = |\tau(c_i)|$ for $i = 1, \dots, n$ and $r_0 = |f(S)|$, and each $k_{i,j}$ is a natural number in binary. Let $v_{i,j}$ be the jth leaf (counting from the right) of $\langle \tau(c_i) \rangle$ if $1 \le i \le n$ and $1 \le j \le r_i$, and let $v_{0,j}$ be the jth leaf (from right) of $\langle f(S) \rangle$ for $1 \le j \le r_0$. If either $1 \le i \le n$ and $\text{pol}(v_{i,j}) = 1$ or $i = 0$ and $\text{pol}(v_{i,j}) = -1$, then for some $p \le \sum_{i=0}^{r_i} r_i$, the pair $(i, v_{i,j})$ represents the pth negative atomic type occurrence in

$$z_1 : \tau(c_1), \dots, z_n : \tau(c_n) \Rightarrow \widehat{N}[z_1, \dots, z_n] : f(S). \tag{57}$$

In this case, $k_{i,j}$ will be the binary representation of p (which can be computed in logarithmic space). If either $1 \le i \le n$ and $\text{pol}(v_{i,j}) = -1$ or $i = 0$ and $\text{pol}(v_{i,j}) = 1$, then the pair $(i, v_{i,j})$ represents a positive atomic type occurrence in (57). In this case, $k_{i,j}$ will be $k_{i',j'}$, where (i', j') is the unique pair such that $(i, v_{i,j})$ is linked to $(i', v_{i',j'})$ in $(\widehat{N}[z_1, \dots, z_n], t)$, where t is the type decoration that is determined by the typing (57). (Uniqueness is guaranteed by the linearity of N.)

For each pair (i, j) for which $(i, v_{i,j})$ is positive, the machine $\mathscr{M}$ computes the corresponding pair (i', j') by starting from $(i, v_{i,j}, \uparrow)$ and following edges of $\overline{G}_{(\widehat{N}[z_1,\dots,z_n],t)}$. The machine does this without explicitly computing the type decoration t. In order to represent a vertex (w, v, d) of $G_{(\widehat{N}[z_1,\dots,z_n],t)}$, the machine $\mathscr{M}$ can place one of its heads at the beginning of the subexpression of N occurring at node w, and store (v, d) in its finite control. This is possible because the fact that $\widehat{N}[z_1, \dots, z_n]$ is β-normal implies that for all nodes w of $\widehat{N}$, $t(w)$ is a subtype of some type in $\{\tau(c_1), \dots, \tau(c_n), f(S)\} \subseteq \{\, \tau(c) \mid c \in C \,\} \cup \{f(S)\}$ by the subformula property, and there are only finitely many possible values of v. Traversal of edges in $G_{(\widehat{N}[z_1,\dots,z_n],t)}$, which is deterministic because $\widehat{N}[z_1, \dots, z_n]$ is linear, can easily be done using an extra head. For example, suppose $\mathscr{M}$ is in a configuration representing $(w, 1v, \uparrow)$, where w is a unary node (λ-abstract) and $t(w) = \gamma \to \delta$. The machine's head is at the first symbol of a string of the form $(\lambda \mathtt{v}l.P)$. In order to switch to a configuration representing $(w', v, \downarrow)$, where $b(w') = w$, $\mathscr{M}$ can use an extra head to locate the occurrence of $\mathtt{v}l$ bound by the lambda. For another example, suppose $\mathscr{M}$ is in a configuration representing $(w1, v, \downarrow)$, where w is a binary node (application) and $t(w0) = \gamma \to \delta$. The machine's head is at the first symbol of Q in a string of the form (PQ). In order to switch to a configuration representing $(w0, 1v, \uparrow)$, the machine can move the head to the first symbol of P by counting in binary unmatched closing parentheses encountered along the way, which requires no more than logarithmic space. □

Thus, for every linear CFLG $\mathscr{G}$, the set $L(\mathscr{G})$ is log-space-reducible to $\{\, (D, q) \mid \text{program}(\mathscr{G}) \cup D \vdash q \,\}$. Since for every Datalog program $\mathbf{P}$, the language $\{\, (D, q) \mid \mathbf{P} \cup D \vdash q \,\}$ is in P, it immediately follows that $L(\mathscr{G})$ is in P for every linear CFLG $\mathscr{G}$, a fact first proved by Salvati [62]. A more careful analysis gives a tight complexity upper bound:

Theorem 3.44. *For every linear CFLG $\mathscr{G}$, $L(\mathscr{G})$ belongs to LOGCFL.*

Proof. Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$, and let $g(n)$ be the polynomial associated with program($\mathscr{G}$) by Lemma 3.2. We show that whenever $N \in L(\mathscr{G})$, there is a derivation tree for $\text{program}(\mathscr{G}) \cup \text{database}(N) \vdash S(\text{tuple}(N))$ of size $\leq g(|\overrightarrow{\text{Con}}(N)|)$. The proof of this claim is by a more careful use of Lemma 3.31 than in the proof of Theorem 3.42.

Let $N \in \Lambda(\Sigma)$ be a linear λ-term in η-long β-normal form relative to $f(S)$. Assume $N \in L(\mathscr{G})$. Let $\overrightarrow{\text{Con}}(N) = (d_1, \dots, d_m)$ and let

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \alpha$$

be a principal typing of $\widehat{N}[y_1, \dots, y_m]$. Then

$$\begin{aligned} \text{database}(N) &= \{\, d_i(\overline{\beta_i}) \mid 1 \leq i \leq m \,\}, \\ \text{tuple}(N) &= \overline{\alpha}. \end{aligned}$$

Since $\mathscr{G}$ is linear, there exists some linear λ-term $P \in \Lambda(\Sigma)$ such that $\vdash_{\mathscr{G}} S(P)$ and $P \twoheadrightarrow_\beta N$. Since the β-reduction from P to N must be non-erasing and non-duplicating, $|\overrightarrow{\text{Con}}(P)| = m$, and $\widehat{P}[y_{h(1)}, \dots, y_{h(m)}] \twoheadrightarrow_\beta \widehat{N}[y_1, \dots, y_m]$ for some permutation h on $\{1, \dots, m\}$. This means

$$\vdash y_1 : \langle \tau(d_1) \rangle (\overline{\beta_1}), \dots, y_m : \langle \tau(d_m) \rangle (\overline{\beta_m}) \Rightarrow \widehat{P}[y_{h(1)}, \dots, y_{h(m)}] : \langle f(S) \rangle (\overline{\alpha}).$$

By Lemma 3.31, there is a derivation tree for program($\mathscr{G}$) $\cup$ database(N) $\vdash$ $S(\overline{\alpha})$ with m extensional nodes. By Lemma 3.2, it follows that there is a derivation tree for program($\mathscr{G}$) $\cup$ database(N) $\vdash$ $S(\overline{\alpha})$ of size at most $g(m)$. Therefore, $(\text{database}(N), S(\text{tuple}(N)), \mathtt{1}^{|\overrightarrow{\text{Con}}(N)|})$ belongs to the set

$$\{\, (D, q, \mathtt{1}^n) \mid \text{there is a derivation tree for program}(\mathscr{G}) \cup D \vdash q \text{ of size} \leq g(n) \,\}. \tag{58}$$

Now assume $N \notin L(\mathscr{G})$. Then by Theorem 3.42, it is not the case that program($\mathscr{G}$) $\cup$ database(N) $\vdash$ $S(\text{tuple}(N))$, and $(\text{database}(N), S(\text{tuple}(N)), \mathtt{1}^{|\overrightarrow{\text{Con}}(N)|})$ does not belong to (58).

By Lemmas 3.1 and 3.43, we conclude that $L(\mathscr{G})$ is log-space reducible to a problem in LOGCFL. Since the class of functions computable in logarithmic space is closed under composition, $L(\mathscr{G})$ itself is in LOGCFL. □

3.6 Almost affine λ-terms

A typed λ-term (M, t), where $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$, is *almost affine* if for every $w, w' \in \mathcal{T}$ such that $w \neq w'$, $f(w) = f(w') \in \mathcal{V}$ or $b(w) = b(w')$ implies that $t(w) = t(w')$ is an atomic type. An untyped λ-term M is *almost affine relative to* $\Gamma \Rightarrow \alpha$ if there is a type decoration t for $\Gamma \Rightarrow M : \alpha$ such that (M, t) is almost affine. We say that a typable λ-term is almost affine if it is almost affine relative to some typing, or equivalently, relative to its principal typing.

If a typed λ-term is almost affine, then so is its η-long form. The class of almost affine untyped λ-terms is closed under η-reduction, but not under β-reduction. For example, a pure λ-term $M = (\lambda x.yxx)(zw)$ is almost affine relative to $y : o, z : o \to o, w : o \Rightarrow o$, but $|M|_\beta = y(zw)(zw)$ is not (relative to any typing).

We say that a λ-term $M \in \Lambda(\Sigma)$ is *almost linear* if M is an almost affine λI-term.

A sequent is *negatively non-duplicated* if no atomic type has more than one negative occurrence in it. The following result generalizes the Coherence Theorem (Theorem 3.18):

Theorem 3.45 (Aoto and Ono [3]). *All inhabitants of a negatively non-duplicated sequent are $\beta\eta$-equal.*[34]

The following is a slight generalization of a result by Aoto [2]:

Theorem 3.46. *If $\Gamma \Rightarrow \alpha$ is a principal typing of an almost affine pure λ-term M, then $\Gamma \Rightarrow \alpha$ is negatively non-duplicated.*

[34]This theorem can be stated in the same style as the Coherence Theorem: If $\Gamma \cup \Gamma' \Rightarrow \alpha$ is a negatively non-duplicated sequent and both $\vdash \Gamma \Rightarrow M : \alpha$ and $\vdash \Gamma' \Rightarrow M' : \alpha$ hold, then $M =_{\beta\eta} M'$. See [46].

Proof. Let $\Gamma = x_1 : \alpha_1, \dots, x_n : \alpha_n$ and let t be a principal type decoration for M associated with the typing $\Gamma \Rightarrow \alpha$. Suppose that $(i, v), (i', v')$ are two distinct negative occurrences of the same atomic type p in $\Gamma \Rightarrow \alpha$. By Lemma 3.25, (i, v) is connected to (i', v') in (M, t). This means that $\overline{G}_{(M,t)}$ contains an undirected path from (i, v, d) to (i', v', d') for some $d, d' \in \{\uparrow, \downarrow\}$. Since both (i, v) and (i', v') are negative, by the property of $\overline{G}_{(M,t)}$ mentioned above immediately after its definition, we must have $d = d'$. We may assume $d = d' = \uparrow$. Since there cannot be a directed path from $(i, v, \uparrow)$ to $(i', v', \uparrow)$, this implies that there are three nodes ν_1, ν_2, ν_3 of $\overline{G}_{(M,t)}$ such that

- $\nu_1 \neq \nu_3$,
- there is a directed path from $(i, v, \uparrow)$ to ν_1,
- (ν_1, ν_2) and (ν_3, ν_2) are edges of $\overline{G}_{(M,t)}$, and
- there is an undirected path from $(i', v', \uparrow)$ to ν_3.

The first and third conditions can obtain only in two cases:

- $\nu_2 = (j, u, \downarrow)$ for some $j \in \{1, \dots, n\}$ and $\nu_1 = (w_1, u, \uparrow), \nu_3 = (w_3, u, \uparrow)$, where $f(w_1) = f(w_3) = x_j$.
- $\nu_2 = (w_2, 1u, \downarrow)$, $\nu_1 = (w_1, u, \uparrow), \nu_3 = (w_3, u, \uparrow)$, and $b(w_1) = b(w_3) = w_2$.

In both cases, since (M, t) is almost affine, it must be the case that $t(w_1) = t(w_3) = p$ and $u = \epsilon$. However, since $\mathrm{pol}(i, v) = -1$ and $\mathrm{pol}(\epsilon) = 1$, there cannot be a directed path from $(i, v, \uparrow)$ to $\nu_1 = (w_1, \epsilon, \uparrow)$, a contradiction. $\square$

Theorems 3.45 and 3.46 show that a principal typing of an almost affine pure λ-term uniquely characterizes it up to $\beta\eta$-equality.

Although we do not need it in establishing the results to follow, we note that the converse of Theorem 3.46 also holds [46]:

Theorem 3.47. *Suppose $\Gamma \Rightarrow \alpha$ is a negatively non-duplicated sequent. For every pure λ-term M such that $\vdash \Gamma \Rightarrow M{:}\alpha$, there exists a λ-term M' such that $M' =_{\beta\eta} M$ and M' is almost affine relative to $\Gamma \Rightarrow \alpha$.*

Let $M \in \Lambda(\Sigma)$ be a typable λ-term, and let t be a principal typing for M. A β-reduction step $M \overset{w}{\to}_\beta M'$ is *almost non-duplicating* if either it is non-duplicating or $\mathrm{subtype}(t(w0), 1) = t(w1)$ is atomic. A β-reduction $M \twoheadrightarrow_\beta M'$ is almost non-duplicating if it consists entirely of almost non-duplicating β-reduction steps.

Example 3.48. Let $M = (\lambda x.(\lambda z.yzz)(xz))(\lambda z.uz)$. Then

$$\begin{aligned} M &\xrightarrow{\epsilon}_\beta (\lambda z.yzz)((\lambda z.uz)z) \\ &\xrightarrow{\epsilon}_\beta y((\lambda z.uz)z)((\lambda z.uz)z) \end{aligned}$$

is almost non-duplicating, whereas

$$\begin{aligned} M &\xrightarrow{00}_\beta (\lambda x.y(xz)(xz))(\lambda z.uz) \\ &\xrightarrow{\epsilon}_\beta y((\lambda z.uz)z)((\lambda z.uz)z) \end{aligned}$$

is not, because the second step duplicates the subterm $(\lambda z.uz)$, whose type must be non-atomic.

A β-reduction $M_0 \xrightarrow{w_1}_\beta M_1 \xrightarrow{w_2}_\beta \cdots \xrightarrow{w_n}_\beta M_n$ is called *leftmost* if for $i = 1, \dots, n$, w_i is the leftmost β-redex of M_{i-1}, i.e., w_i is the first β-redex of M_{i-1} under the lexicographic ordering $\prec$ of the nodes of M_{i-1}.

Lemma 3.49. *If $M \in \Lambda(\Sigma)$ is almost affine, then the leftmost β-reduction from M to $|M|_\beta$ is almost non-duplicating.*

Proof. Let $M = M_0 \xrightarrow{w_1}_\beta M_1 \xrightarrow{w_2}_\beta \cdots \xrightarrow{w_n}_\beta M_n = |M|_\beta$ by leftmost β-reduction, and let $M_i = (\mathcal{T}_i, f_i, b_i)$. We show that each step of this reduction is almost non-duplicating. Let t be a principal type decoration of M, and for $i = 0, \dots, n$, let t_i be the type decoration for M_i such that

$$(M, t) = (M_0, t_0) \xrightarrow{w_1}_\beta (M_1, t_1) \xrightarrow{w_2}_\beta \cdots \xrightarrow{w_n}_\beta (M_n, t_n).$$

To prove the lemma, it suffices to show that for every i and every unary node $w \in \mathcal{T}_i^{(1)}$, either

(i) w is to the left of any β-redex in M_i,

(ii) $\mathrm{subtype}(t_i(w), 1)$ is an atomic type, or

(iii) there is at most one $w' \in \mathcal{T}_i$ such that $b_i(w') = w$.

The condition holds of (M_0, t_0) by the assumption that M is an almost affine λ-term. Assume that (M_i, t_i) satisfies the condition, and let v be a unary node of $\mathcal{T}_{i+1}$. Then v is a descendant of a unary node w of $\mathcal{T}_i$ distinct from w_i0.

Suppose that (i) holds of w. Then w is to the left of w_i. Clearly, v must be to the left of any β-redex in M_{i+1}, satisfying (i).

Suppose that (ii) holds of w. Since $t_{i+1}(v) = t_i(w)$, it holds that $\mathrm{subtype}(t_{i+1}(v), 1)$ is an atomic type. So v satisfies (ii).

Suppose that (iii) holds of w. Assume that v does not satisfy (iii), i.e., there are $v', v'' \in \mathcal{T}_{i+1}$ such that $v' \neq v''$ and $b_{i+1}(v') = b_{i+1}(v'') = v$. Then v' and v'' must be descendants of the unique node w' of $\mathcal{T}_i$ such that $b_i(w') = w$. For this to hold, it must be the case that $w < w_i$ and $w_i 1 \leq w'$. Then $v = w$. Since w_i is the leftmost β-redex of M_i, there is no β-redex in M_i to the left of w, and it follows that there is no β-redex in M_{i+1} to the left of v. □

We now define a certain equivalence relation between nodes in a λ-term. Let $M = (\mathcal{T}, f, b)$, and let $w, w' \in \mathcal{T}$. We say that w and w' are *congruent* in M and write $w \cong_M w'$, if the following conditions hold:

- $\{\, v \mid wv \in \mathcal{T} \,\} = \{\, v \mid w'v \in \mathcal{T} \,\}$,
- for all v such that $wv \in \mathcal{T}^{(0)}$, either
  - $wv, w'v \in \mathrm{dom}(f)$ and $f(wv) = f(w'v)$,
  - $wv, w'v \in \mathrm{dom}(b)$ and $b(wv) = b(w'v)$, or
  - $wv, w'v \in \mathrm{dom}(b)$ and $b(wv) = wu$ and $b(w'v) = w'u$ for some $u < v$.

It is clear that if $w \cong_M w'$, then for every writing ℓ of M, the λ-expressions $\mathrm{sub}_{M,\ell}(w)$ and $\mathrm{sub}_{M,\ell}(w')$ represent the same λ-term.

The following is clear from the definition of the ancestor-descendant relation for one-step β-reduction.

Lemma 3.50. *Let $M \overset{w}{\to}_\beta M'$ be a duplicating β-reduction step. If $(M, w1) \overset{w}{\blacktriangleright} (M', v_1)$ and $(M, w1) \overset{w}{\blacktriangleright} (M', v_2)$, then $v_1 \cong_{M'} v_2$.*

Let $M = (\mathcal{T}_M, f_M, b_M)$ be a λ-term. Suppose that $v_1, \dots, v_k$ are nodes in $\mathcal{T}_M$ such that $v_1 \cong_M \ldots \cong_M v_k$. Let w be a node in $\mathcal{T}_M$ such that for all i, $w < v_i$, and $b_M(v_i u) < w$ holds whenever $b_M(v_i u) < v_i$. It is clear that there must be such a w. Define $\mathrm{expand}(M, w, \{v_1, \dots, v_k\}) = (\mathcal{T}, f, b)$ as follows:

$$\begin{aligned}
\mathcal{T} &= \{\, v \in \mathcal{T}_M \mid w \not< v \,\} \cup \{w0\} \cup \{\, w00v \mid wv \in \mathcal{T}_M, \neg\exists i (v_i < wv) \,\} \cup \\
&\qquad \{\, w1u \mid v_1 u \in \mathcal{T}_M \,\}, \\
f &= \{\, (v, f_M(v)) \mid w \not< v, v \in \mathrm{dom}(f_M) \,\} \cup \\
&\qquad \{\, (w00v, f_M(wv)) \mid wv \in \mathrm{dom}(f_M), \neg\exists i (v_i < wv) \,\} \cup \\
&\qquad \{\, (w1u, f_M(v_1 u)) \mid v_1 u \in \mathrm{dom}(f_M) \,\}
\end{aligned}$$

$$\begin{aligned} b = {} & \{\, (v, b_M(v)) \mid w \not< v, v \in \mathrm{dom}(b_M) \,\} \cup \\ & \{\, (w00v, b_M(wv)) \mid b_M(wv) < w, \neg\exists i(v_i \leq wv) \,\} \cup \\ & \{\, (w00v, w00u) \mid b_M(wv) = wu, \neg\exists i(v_i \leq wv) \,\} \cup \\ & \{\, (w00v_i, w0) \mid 1 \leq i \leq k \,\} \cup \\ & \{\, (w1u, b_M(v_1u)) \mid b_M(v_1u) < w \,\} \cup \{\, (w1u, w1s) \mid b_M(v_1u) = v_1s \,\}. \end{aligned}$$

It is clear that $\mathrm{expand}(M, w, \{v_1, \dots, v_k\})$ is a λ-term, and we have $\mathrm{expand}(M, w, \{v_1, \dots, v_k\}) \overset{w}{\to}_\beta M$ and $(\mathrm{expand}(M, w, \{v_1, \dots, v_k\}), w1) \overset{w}{\blacktriangleright} (M, v_i)$ for $i = 1, \dots, k$.

Lemma 3.51. *Let $M \in \Lambda(\Sigma)$, where $\Sigma = (A, C, \tau)$, and let t be a type decoration of M. Suppose that w, w' are two nodes of M such that $w \cong_M w'$. If $t(w)$ is an atomic type, then $t(w) = t(w')$.*

Proof. Let ℓ be a writing of M and let $N = \mathrm{sub}_{M,\ell}(w) = \mathrm{sub}_{M,\ell}(w')$. Let

$$\begin{aligned} \Gamma_w &= \{\, (x, t(wv)) \mid x = f(wv) \in \mathcal{V} \,\} \cup \{\, (\ell(b(wv)), t(wv)) \mid b(wv) \leq w \,\} \\ \Gamma_{w'} &= \{\, (x, t(w'v)) \mid x = f(w'v) \in \mathcal{V} \,\} \cup \{\, (\ell(b(w'v)), t(w'v)) \mid b(w'v) \leq w' \,\}. \end{aligned}$$

Then we have

$$\begin{gathered} \vdash_\Sigma \Gamma_w \Rightarrow N : t(w), \\ \vdash_\Sigma \Gamma_{w'} \Rightarrow N : t(w'). \end{gathered}$$

Since $w \cong_M w'$, we must have $\Gamma_w = \Gamma_{w'}$. By the Subject Reduction Theorem (Theorem 3.14),

$$\begin{gathered} \vdash_\Sigma \Gamma' \Rightarrow |N|_\beta : t(w), \\ \vdash_\Sigma \Gamma' \Rightarrow |N|_\beta : t(w') \end{gathered}$$

where $\Gamma' = \Gamma_w \restriction \mathrm{FV}(|N|_\beta)$. By assumption, $t(w)$ is some atomic p, so $|N|_\beta$ must be of the form

$$yP_1 \dots P_l$$

for some variable y, or else of the form

$$cP_1 \dots P_l$$

for some constant c. In the former case, $y : \gamma_1 \to \dots \to \gamma_l \to p$ is in Γ', and in the latter case, $\tau(c) = \gamma_1 \to \dots \to \gamma_l \to p$ for some types $\gamma_1, \dots, \gamma_l$. In either case, we must have $t(w') = p$. □

The following lemma generalizes the Subject Expansion Theorem (Theorem 3.15):

Lemma 3.52. *Let $M, M' \in \Lambda(\Sigma)$ be typable λ-terms. Suppose $M \twoheadrightarrow_\beta M'$ by non-erasing, almost non-duplicating β-reduction. If $\vdash_\Sigma \Gamma \Rightarrow M' : \alpha$, then $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$.*

Proof. Clearly, it suffices to consider the case where the β-reduction consists of just one step and $\Gamma \Rightarrow \alpha$ is a principal typing of M'. Let $M = (\mathcal{T}, f, b), M' = (\mathcal{T}', f', b')$, and $M \overset{w}{\rightarrow}_\beta M'$. Let t be a principal type decoration of M, t' be the type decoration of M' induced by t (i.e., $(M,t) \overset{w}{\rightarrow}_\beta (M',t')$), and $\tilde{t}$ be a principal type decoration of M' (with the associated typing $\Gamma \Rightarrow \alpha$). If the β-reduction step $M \overset{w}{\rightarrow}_\beta M'$ is non-erasing and non-duplicating, then $\vdash_\Sigma \Gamma \Rightarrow M : \alpha$ by the Subject Expansion Theorem. So suppose that this β-reduction step is duplicating. Let

$$\{\, v \mid b(w00v) = w0 \,\} = \{v_1, \dots, v_k\},$$

where $k \geq 2$. Since the β-reduction step is almost non-duplicating, we have $t(w1) = p$ for some atomic type p. For each $i \in \{1, \dots, k\}$, we have

$$(M, w1) \overset{w}{\blacktriangleright} (M', wv_i)$$

and $t'(wv_i) = p$. Since $\tilde{t}$ is a principal type decoration of M', there is a type substitution σ such that $t' = \sigma \circ \tilde{t}$. It follows that for each $i = 1, \dots, k$, there is an atomic type q_i such that $\tilde{t}(wv_i) = q_i$. By Lemma 3.50, we have

$$wv_1 \cong_{M'} \dots \cong_{M'} wv_k,$$

and by Lemma 3.51, it follows that

$$q_1 = \dots = q_k.$$

Define a function $\tilde{t}_1 \colon \mathcal{T} \to \mathscr{T}(A)$ as follows:

$$\tilde{t}_1(v) = \begin{cases} \tilde{t}(v) & \text{if } w \not\leq v, \\ \tilde{t}(w) & \text{if } v = w, \\ q_1 \to \tilde{t}(w) & \text{if } v = w0, \\ \tilde{t}(wu) & \text{if } v = w00u, \\ \tilde{t}(wv_1u) & \text{if } v = w1u. \end{cases}$$

It is clear that $\tilde{t}_1$ is a type decoration of M. Although $(M, \tilde{t}_1) \overset{w}{\rightarrow}_\beta (M', \tilde{t})$ does not necessarily hold, it is easy to see that $\tilde{t}_1$ is a type decoration for $\Gamma \Rightarrow M : \alpha$. □

Lemma 3.53. *If M is an almost linear λ-term, M and $|M|_\beta$ have the same principal typing.*

Proof. Since M is almost affine, by Lemma 3.49, the leftmost β-reduction from M to $|M|_\beta$ is almost non-duplicating. Since M is a λI-term, this β-reduction must also be non-erasing. By the Subject Reduction Theorem (Theorem 3.14) and Lemma 3.52, any typing of M is a typing of $|M|_\beta$ and vice versa. □

3.7 From CFLGs to Datalog: The case of almost linear CFLGs

Given Aoto and Ono's [3] generalization of the Coherence Theorem (Theorem 3.45), we easily obtain a generalization of Lemma 3.41 to almost affine λ-terms.

Lemma 3.54. *Let M be an almost affine λ-term in strict η-long form relative to γ. Then $|M|_\beta$ is the only λ-term in $\Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ that is in η-long β-normal form relative to γ.*

Proof. The proof parallels that of Lemma 3.41. Let $\overrightarrow{\text{Con}}(M) = (d_1, \dots, d_m)$, and let

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \alpha \tag{59}$$

be a principal typing of $\widehat{M}[y_1, \dots, y_m]$. Then $\text{database}(M) = \{\, d_i(\overline{\beta_i}) \mid 1 \le i \le m \,\}$. Note that $\widehat{M}[y_1, \dots, y_m]$ is a pure almost affine λ-term in strict η-long form. By Theorem 3.46, (59) is negatively non-duplicated.

We know from Lemmas 3.17, 3.34, and 3.37 that $|M|_\beta$ is in η-long β-normal form relative to γ and that $|M|_\beta \in \Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$. Suppose $N \in \Lambda(\text{database}(M), \langle\gamma\rangle(\text{tuple}(M)))$ and $|\overrightarrow{\text{Con}}(N)| = n$. Then there is a function $g : \{1, \dots, n\} \to \{1, \dots, m\}$ such that $\overrightarrow{\text{Con}}(N) = (d_{g(1)}, \dots, d_{g(n)})$ and

$$\vdash z_1 : \beta_{g(1)}, \dots, z_n : \beta_{g(n)} \Rightarrow \widehat{N}[z_1, \dots, z_n] : \alpha.$$

Substituting $y_{g(i)}$ for z_i, we get

$$\vdash \{\, y_{g(i)} : \beta_{g(i)} \mid 1 \le i \le n \,\} \Rightarrow \widehat{N}[y_{g(1)}, \dots, y_{g(n)}] : \alpha.$$

By Theorem 3.45, $\widehat{N}[y_{g(1)}, \dots, y_{g(n)}] =_{\beta\eta} \widehat{M}[y_1, \dots, y_n]$, and so $N =_{\beta\eta} M$. It follows that if N is in η-long β-normal form relative to γ, then $N = |M|_\beta$. □

A CFLG $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ is *almost linear* if for every $\pi \in \mathscr{P}$, the λ-term on the left-hand side of π is almost linear. An example of an almost linear CFLG is the grammar in Example 3.28. Almost linear CFLGs can encode IO context-free tree grammars [21] in a straightforward way, similarly to de Groote and Pogodalla's [19] encoding of linear context-free tree grammars.[35]

Lemma 3.39 and Theorem 3.40 do not generalize to the almost linear case, and there is no simple analogue of Theorem 3.42 for almost linear CFLGs. The reason is that $\Lambda(D, \alpha)$ need not be closed under the converse of non-erasing almost non-duplicating β-reduction (in contrast to part (ii) of Lemma 3.34), despite the fact that the Subject Expansion Theorem generalizes to such β-reduction (Lemma 3.52). This is so even when $D = \text{database}(N)$ and $\alpha = \langle\gamma\rangle(\text{tuple}(N))$ for an almost linear λ-term $N \in \Lambda(\Sigma)$ in η-long β-normal form relative to γ.

Example 3.55. Consider the λ-term (19) from Section 2.2:

$$N = \exists(\lambda y. \wedge(\mathbf{unicorn}\, y)(\wedge(\mathbf{find}\, y\, \mathbf{John})(\mathbf{catch}\, y\, \mathbf{John}))),$$

where the types of the constants $\exists, \wedge, \mathbf{unicorn}, \mathbf{find}, \mathbf{John}, \mathbf{catch}$ are as follows:

$$\begin{aligned} \exists &: (e \to t) \to t, \\ \wedge &: t \to t \to t, \\ \mathbf{unicorn} &: e \to t, \\ \mathbf{find} &: e \to e \to t, \\ \mathbf{John} &: e, \\ \mathbf{catch} &: e \to e \to t. \end{aligned}$$

We have $\overrightarrow{\text{Con}}(N) = (\exists, \wedge, \mathbf{unicorn}, \wedge, \mathbf{find}, \mathbf{John}, \mathbf{catch}, \mathbf{John})$. A principal typing of

$$\widehat{N}[z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8] = z_1(\lambda y. z_2(z_3 y)(z_4(z_5 y z_6)(z_7 y z_8)))$$

is

$$z_1{:}(4{\to}2){\to}1, z_2{:}3{\to}5{\to}2, z_3{:}4{\to}3, z_4{:}6{\to}8{\to}5, z_5{:}4{\to}7{\to}6, z_6{:}7, z_7{:}4{\to}9{\to}8, z_8{:}9 \Rightarrow 1,$$

which gives rise to

$$\begin{aligned} \text{database}(N) = \{&\exists(1,2,4), \wedge(2,5,3), \mathbf{unicorn}(3,4), \wedge(5,8,6), \mathbf{find}(6,7,4), \mathbf{John}(7), \\ &\mathbf{catch}(8,9,4), \mathbf{John}(9)\}, \end{aligned}$$

[35]With respect to string languages, it is easy to see that almost linear CFLGs generating λ-terms representing strings are no more powerful than linear CFLGs (see footnote 41). What this means is that encodings of "non-linear" grammars like IO macro grammars [24] and parallel multiple context-free grammars [66] in terms of CFLGs cannot be almost linear. However, our reduction to Datalog applies to these cases indirectly if we take almost linear CFLGs encoding tree analogues of these grammars (i.e., IO context-free tree grammars and what one might call "parallel multiple regular tree grammars") and use regular sets of trees as input. See Section 4.2 below.

tuple$(N) = (1)$.

Now consider the λ-term (25):

$$N^\circ = \exists(\lambda y. \wedge(\mathbf{unicorn}\, y)((\lambda x. \wedge(\mathbf{find}\, y\, x)(\mathbf{catch}\, y\, x))\, \mathbf{John})).$$

Although $N^\circ \twoheadrightarrow_\beta N$ by non-erasing almost non-duplicating β-reduction, it is easy to see that $N^\circ \notin \Lambda(\text{database}(N), 1)$. This does not contradict Lemma 3.52, because

$$\widehat{N^\circ}[y_1, y_2, y_3, y_4, y_5, y_6, y_7] \not\twoheadrightarrow_\beta \widehat{N}[z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8],$$

no matter how one picks the variables $y_1, y_2, y_3, y_4, y_5, y_6, y_7$.

Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be an almost linear CFLG, and suppose that $N \in \Lambda(\Sigma)$ is in η-long β-normal form relative to $f(S)$. In order to find a database D and a tuple $\vec{s}$ such that $N \in L(\mathscr{G})$ if and only if program$(\mathscr{G}) \cup D \vdash S(\vec{s})$, what we do is to β-expand N to a 'most compact' almost linear λ-term N° such that $N^\circ \twoheadrightarrow_\beta N$ in the sense that for any almost linear $P \in \Lambda(\Sigma)$, if $P \twoheadrightarrow_\beta N$ and two occurrences of the same constant in N have a common ancestor in P, then they have a common ancestor in N°. Thus, for every constant $c \in C$, N° contains the fewest occurrences of c among all almost linear λ-terms that β-reduce to N. We have $P \in \Lambda(\text{database}(N^\circ), \text{tuple}(N^\circ))$ if and only if $P \twoheadrightarrow_\beta N$ for all almost linear $P \in \Lambda(\Sigma)$ in η-long form,[36] and the desired equivalence of $N \in L(\mathscr{G})$ and program$(\mathscr{G}) \cup \text{database}(N^\circ) \vdash S(\text{tuple}(N^\circ))$ follows. We show that such N° can be computed efficiently.

Let us call a node v of a λ-term $M = (\mathcal{T}, f, b)$ a *pivot* if (i) $v \in \mathcal{T}^{(0)} \cup \mathcal{T}^{(2)}$, and (ii) $v = v'0$ implies $v' \in \mathcal{T}^{(1)}$. A pivot v is *duplicated* if $v \notin \text{dom}(b)$ and there is another pivot $v' \in \mathcal{T}$ such that $v \cong_M v'$. If some type decoration of M assigns a node v an atomic type, then v must be a pivot. If M is in η-long form and v is a pivot of M, then the principal type decoration of M assigns v an atomic type.

Algorithm 1.

```
1: procedure Collapse($M$)
2:     $M^\circ \leftarrow M$
3:     while there is a duplicated pivot in $M^\circ$ do
4:         Let $V$ be the set of duplicated pivots of maximal height in $M^\circ$
5:         Let $v_1$ be the leftmost (i.e., lexicographically first) node in $V$
6:         Let $\{v_2, \dots, v_k\} = \{ v \mid v$ is a pivot, $v \neq v_1$, and $v_1 \cong_{M^\circ} v \}$
7:         Let $w$ be the pivot of minimal height such that $w < v_i$ for $i = 1, \dots, k$
```

[36]This corresponds to the special property (28) mentioned in Section 2.3.

```
 8:         $M^\circ \leftarrow \text{expand}(M^\circ, w, \{v_1, \dots, v_k\})$
 9:     end while
10:     return $M^\circ$
11: end procedure
```

It is clear that the node w picked in line 7 is such that $\text{expand}(M^\circ, w, \{v_1, \dots, v_k\})$ in line 8 is defined.

Lemma 3.56. *Let M be a typable λ-term in η-long form, and consider the execution of Algorithm 1 on input M. If w_i is the node w that is picked in line 7 during the ith iteration of the while loop, then the following conditions hold after the ith iteration of the while loop.*

(i) *M° is a typable λ-term in η-long form.*

(ii) *$M^\circ \twoheadrightarrow_\beta M$ by non-erasing, almost non-duplicating β-reduction.*

(iii) *If u_1 and u_2 are pivots of M such that $u_1 \cong_M u_2$ and $(M^\circ, u_1') \overset{w_i,\dots,w_1}{\blacktriangleright} (M, u_1)$ and $(M^\circ, u_2') \overset{w_i,\dots,w_1}{\blacktriangleright} (M, u_2)$, then u_1' and u_2' are also pivots and $u_1' \cong_{M_i} u_2'$.*

Proof. Let $M_0 = M$, and let M_i be the value of M° after the ith iteration of the while loop. We show that the conditions (i), (ii), and (iii) hold by induction on i, on the understanding that $u_1' = u_1$ and $u_2' = u_2$ when $i = 0$.

The three conditions clearly hold when $i = 0$. Assume that the three conditions hold of M_i and let $v_1, v_2, \dots, v_k$ be the nodes that the algorithm picks in lines 5–6 during the $(i+1)$st iteration of the while loop. Since M_i is typable and in η-long form, the principal type decoration t_i of M_i assigns v_1 an atomic type p, and by Lemma 3.51, $t_i(v_2) = \cdots = t_i(v_k) = p$. As in the proof of Lemma 3.52, this allows us to define a type decoration for $M_{i+1} = \text{expand}(M_i, w_{i+1}, \{v_1, \dots, v_k\})$ that assigns p to the node $w_{i+1}1$. It follows that the β-reduction step $M_{i+1} \overset{w_{i+1}}{\rightarrow}_\beta M_i$ is almost non-duplicating. It is also easy to see that M_{i+1} is in η-long form. So (i) and (ii) hold of M_{i+1}. To see that (iii) holds of M_{i+1}, let s_1 and s_2 be pivots of M_i such that $s_1 \cong_{M_i} s_2$, and let s_1' and s_2' be the nodes such that $(M_{i+1}, s_1') \overset{w_{i+1}}{\blacktriangleright} (M_i, s_1)$ and $(M_{i+1}, s_2') \overset{w_{i+1}}{\blacktriangleright} (M_i, s_2)$. It is easy to see that s_1' and s_2' are also pivots, so it suffices to show $s_1' \cong_{M_{i+1}} s_2'$. If $s_1 = s_2$, then clearly $s_1' = s_2'$. If $s_1 \neq s_2$, since v_1 is a duplicated pivot of maximal height in M_i and $v_1, v_2, \dots, v_k$ are all of the same height, it cannot be the case that $s_1 < v_i$ or $s_2 < v_i$ for any i. This ensures that $\{\, s \mid s_1 s \text{ is a node of } M_i \,\} = \{\, s \mid s_1' s \text{ is a node of } M_{i+1} \,\}$, $(M_{i+1}, s_1' s) \overset{w_{i+1}}{\blacktriangleright} (M_i, s_1 s)$, and likewise for s_2 and s_2'. By Lemma 3.4, it is easy to check that the remaining conditions for $s_1' \cong_{M_{i+1}} s_2'$ are satisfied. □

Lemma 3.57. *Algorithm 1 always terminates.*

Proof. One can prove by induction that the following condition holds at each stage of the algorithm: every pivot v in $M^\circ = (\mathcal{T}_{M^\circ}, f_{M^\circ}, b_{M^\circ})$ that is not in $\mathrm{dom}(b_{M^\circ})$ is either an ancestor of a pivot in M or else a β-redex such that $v00$ and $v1$ are pivots not in $\mathrm{dom}(b_{M^\circ})$. It follows that every pivot in M° that is not in $\mathrm{dom}(b_{M^\circ})$ contains an ancestor of a node in M, and the number of nodes in M° that are ancestors of nodes in M strictly decreases at each iteration of the while loop. □

We now prove two lemmas (Lemmas 3.60 and 3.61) needed to show that Collapse(M) is more compact than any almost affine λ-term that β-reduces to M. These lemmas require us to introduce two new binary relations on nodes. The following lemma is needed to prove the first of these lemmas:

Lemma 3.58. *Let $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$, and $v, v' \in \mathcal{T}$ be two nodes such that $v \cong_M v'$. Suppose $M = M_0 \overset{w_1}{\to}_\beta M_1 \overset{w_2}{\to}_\beta \cdots \overset{w_n}{\to}_\beta M_n = |M|_\beta$. Let $v_0 = v, v'_0 = v'$, and for $1 \le i \le n$, let v_i and v'_i be nodes of M_i that satisfy one of the following conditions:*

(i) *$(M_{i-1}, v_{i-1}) \overset{w_i}{\blacktriangleright}_k (M_i, v_i)$ and $(M_{i-1}, v'_{i-1}) \overset{w_i}{\blacktriangleright}_{k'} (M_i, v'_i)$, where $k = k'$ if both $w_i 1 \le v_{i-1}$ and $w_i 1 \le v'_{i-1}$ hold.*

(ii) *$v_{i-1} = v_i = w_i$ and $(M_{i-1}, v'_{i-1}) \overset{w_i}{\blacktriangleright} (M_i, v'_i)$.*

(iii) *$(M_{i-1}, v_{i-1}) \overset{w_i}{\blacktriangleright} (M_i, v_i)$ and $v'_{i-1} = v'_i = w_i$.*

Then $v_n \cong_{|M|_\beta} v'_n$. Moreover, $(M, vu) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v_n s)$ implies $(M, v'u) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v'_n s)$.

Proof. Let $M_i = (\mathcal{T}_i, f_i, b_i)$ for $i = 0, 1, \dots, n$. For $i = 1, \dots, n$, define $\hat{w}_i$ by:

$$\hat{w}_i = \begin{cases} v'_i r & \text{if } w_i = v_i r, \\ v_i r & \text{if } w_i = v'_i r, \\ w_i & \text{if } v_i \not\le w_i \text{ and } v'_i \not\le w_i. \end{cases}$$

Then it is not hard to see that there are λ-terms $\hat{M}_i = (\hat{\mathcal{T}}_i, \hat{f}_i, \hat{b}_i)$ for $i = 0, 1, \dots, n$ such that

$$v_i, v'_i \in \hat{\mathcal{T}}_i,$$
$$\{\, u \mid u \in \mathcal{T}_i, v_i \not< u, v'_i \not< u \,\} = \{\, u \mid u \in \hat{\mathcal{T}}_i, v_i \not< u, v'_i \not< u \,\},$$

$$\{\, u \mid v_i u \in \mathcal{T}_i \,\} = \{\, u \mid v_i' u \in \hat{\mathcal{T}}_i \,\},$$
$$\{\, u \mid v_i' u \in \mathcal{T}_i \,\} = \{\, u \mid v_i u \in \hat{\mathcal{T}}_i \,\},$$

and

$$M = \hat{M}_0 \xrightarrow{\hat{w}_1}_\beta \hat{M}_1 \xrightarrow{\hat{w}_2}_\beta \cdots \xrightarrow{\hat{w}_n}_\beta \hat{M}_n = |M|_\beta.$$

Moreover, we can check that the following conditions hold for $i = 0, 1, \dots, n$ by induction:

- $v_i u \in \mathrm{dom}(f_i)$ if and only if $v_i' u \in \mathrm{dom}(\hat{f}_i)$,
- if $v_i u \in \mathrm{dom}(f_i)$, then $f_i(v_i u) = \hat{f}_i(v_i' u)$,
- $v_i u \in \mathrm{dom}(b_i)$ if and only if $v_i' u \in \mathrm{dom}(\hat{b}_i)$,
- if $v_i u \in \mathrm{dom}(b_i)$, then $v_i \le b_i(v_i u)$ if and only if $v_i' \le \hat{b}_i(v_i' u)$,
- if $v_i u \in \mathrm{dom}(b_i)$ and $v_i \le b_i(v_i u)$, then for some s, $b_i(v_i u) = v_i s$ and $\hat{b}_i(v_i' u) = v_i' s$,
- if $v_i u \in \mathrm{dom}(b_i)$ and $b_i(v_i u) < v_i$, then $b_i(v_i u) = \hat{b}_i(v_i' u)$,
- $i \ge 1$ and $(M_{i-1}, v_{i-1} u) \overset{w_i}{\blacktriangleright} (M_i, v_i s)$ imply $(\hat{M}_{i-1}, v_{i-1}' u) \overset{\hat{w}_i}{\blacktriangleright} (\hat{M}_i, v_i' s)$.

From these conditions, we can see that $v_n \cong_{|M|_\beta} v_n'$ and that $(M, vu) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v_n s)$ implies $(M, v'u) \overset{\hat{w}_1,\dots,\hat{w}_n}{\blacktriangleright} (|M|_\beta, v_n' s)$. By Theorem 3.5, we can conclude that $(M, vu) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v_n s)$ implies $(M, v'u) \overset{w_1,\dots,w_n}{\blacktriangleright} (|M|_\beta, v_n' s)$. $\square$

Let $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$. We call two nodes w, w' of M *homologous* and write $w \approx_M w'$ if there are v, v', u satisfying the following conditions:

- $w = vu, w' = v'u$,
- $v \cong_M v'$, and
- v and v' are pivots.

The relation $\approx_M$ is symmetric, but not transitive. We call two nodes w, w' of M *similar* if $w \approx_M^* w'$ (i.e., if they stand in the reflexive transitive closure of the relation of being homologous).

Example 3.59. Let

$$M = \lambda yzvw.w(v(\lambda x.z(yx)(yx)))(v(\lambda x.z(yx)(yx))).$$

This is a λ-term in η-long β-normal form, and the following is a natural deduction representation of (M, t), where t is a principal type decoration of M:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[2\to2\to1]^{000}\quad \dfrac{[(5\to3)\to2]^{00}\quad \dfrac{\dfrac{\dfrac{[4\to4\to3]^{0}\quad \dfrac{[5\to4]^{\epsilon}\quad [5]^{0000011}}{4}}{4\to3}\quad \dfrac{[5\to4]^{\epsilon}\quad [5]^{0000011}}{4}}{3}}{5\to3}\,0000011}{2}}{2\to1}\quad \dfrac{[(5\to3)\to2]^{00}\quad \dfrac{\dfrac{\dfrac{[4\to4\to3]^{0}\quad \dfrac{[5\to4]^{\epsilon}\quad [5]^{000011}}{4}}{4\to3}\quad \dfrac{[5\to4]^{\epsilon}\quad [5]^{000011}}{4}}{3}}{5\to3}\,000011}{2}}{1}}{(2\to2\to1)\to1}\,000}{((5\to3)\to2)\to(2\to2\to1)\to1}\,00}{(4\to4\to3)\to((5\to3)\to2)\to(2\to2\to1)\to1}\,0}{(5\to4)\to(4\to4\to3)\to((5\to3)\to2)\to(2\to2\to1)\to1}\,\epsilon$$

Note that a node v of M is a pivot if and only if $t(v)$ is an atomic type. We have

- 000001 $\cong_M$ 00001 (the two occurrences of $v(\lambda x.z(yx)(yx))$ (with type 2) are congruent)
- 0000011001 $\cong_M$ 000001101 (the first and second occurrences of yx (with type 4) are congruent)
- 000011001 $\cong_M$ 00001101 (the third and fourth occurrences of yx (with type 4) are congruent)

Consequently, we have

- 00000110010 $\approx_M$ 0000110010 (the first and third occurrences of y are homologous),
- 0000011010 $\approx_M$ 000011010 (the second and fourth occurrences of y are homologous),
- 00000110010 $\approx_M$ 0000011010 (the first and second occurrences of y are homologous),
- 0000110010 $\approx_M$ 000011010 (the third and fourth occurrences of y are homologous),

and all four occurrences of y in (the above λ-expression for) M are similar. Note that M β-expands to an almost linear λ-term

$$M' = \lambda yzvw.(\lambda x_1.wx_1x_1)(v(\lambda x.(\lambda x_2.zx_2x_2)(yx)))),$$

in which all four occurrences of y in M have a common ancestor.

Lemma 3.60. *Let $M \in \Lambda(\Sigma)$ be a typable λ-term and suppose $M \twoheadrightarrow_\beta |M|_\beta$ by almost non-duplicating β-reduction. If two distinct nodes of $|M|_\beta$ share a common ancestor in M, then they are similar.*

Proof. Consider two distinct nodes v, v' of $|M|_\beta$ that share a common ancestor in M. Let $M = M_0 \overset{w_1}{\to}_\beta M_1 \overset{w_2}{\to}_\beta \cdots \overset{w_n}{\to}_\beta M_n = |M|_\beta$ be an almost non-duplicating β-reduction, and let t_i be a principal type decoration of M_i. Let $v_n = v$, $v'_n = v'$, and for $i = 1, \dots, n$, let v_{i-1} and v'_{i-1} be the nodes of M_{i-1} such that $(M_{i-1}, v_{i-1}) \overset{w_i}{\blacktriangleright}_{k_i} (M_i, v_i)$ and $(M_{i-1}, v'_{i-1}) \overset{w_i}{\blacktriangleright}_{k'_i} (M_i, v'_i)$. By assumption, $v_0 = v'_0$. We first prove the following:

Claim. For some distinct nodes u, u' of $|M|_\beta$, it holds that $u \le v$, $u' \le v'$, $u \cong_{|M|_\beta} u'$, and u and u' are pivots.

Since v and v' are distinct, there is an $i \ge 1$ such that $v_{i-1} = v'_{i-1}$ and $v_i \ne v'_i$. We must have $w_i1 \le v_{i-1}$, $w_i1 \le v'_{i-1}$, and $k_i \ne k'_i$. Let $m = \max\{\, i \mid 1 \le i \le n, w_i1 \le v_{i-1}, w_i1 \le v'_{i-1}, k_i \ne k'_i \,\}$. There must be nodes u_m, u'_m of M_m such that $(M_{m-1}, w_m1) \overset{w_m}{\blacktriangleright}_{k_m} (M_m, u_m)$, $(M_{m-1}, w_m1) \overset{w_m}{\blacktriangleright}_{k'_m} (M_m, u'_m)$, $u_m \le v_m$, and $u'_m \le v'_m$. By Lemma 3.50, we have $u_m \cong_{M_m} u'_m$. Since by assumption the β-reduction step $M_{m-1} \overset{w_m}{\to}_\beta M_m$ is almost non-duplicating, $t_{m-1}(w_m1)$ must be an atomic type. It follows that u_m and u'_m are pivots; in particular, neither u_m nor u'_m is a unary node.

For $i = m+1, \dots, n$, define u_i and u'_i as follows:

$$u_i = \begin{cases} w_i & \text{if } u_{i-1} = w_i, \\ \text{the node such that } (M_{i-1}, u_{i-1}) \overset{w_i}{\blacktriangleright}_{k_i} (M_i, u_i) & \text{if } w_i1 \le u_{i-1}, \\ \text{the node such that } (M_{i-1}, u_{i-1}) \overset{w_i}{\blacktriangleright}_{1} (M_i, u_i) & \text{otherwise.} \end{cases}$$

$$u'_i = \begin{cases} w_i & \text{if } u'_{i-1} = w_i, \\ \text{the node such that } (M_{i-1}, u'_{i-1}) \overset{w_i}{\blacktriangleright}_{k'_i} (M_i, u'_i) & \text{if } w_i1 \le u'_{i-1}, \\ \text{the node such that } (M_{i-1}, u'_{i-1}) \overset{w_i}{\blacktriangleright}_{1} (M_i, u'_i) & \text{otherwise.} \end{cases}$$

It is easy to see by induction that such u_i and u'_i always exist, and it holds that $u_i \leq v_i$ and $u'_i \leq v'_i$. By the assumption about m, we have that for $i \in \{m+1, \dots, n\}$, if $w_i 1 \leq u_{i-1}$ and $w_i 1 \leq u'_{i-1}$, then $k_i = k'_i$. By Lemma 3.58, it follows that $u_n \cong_{|M|_\beta} u'_n$.

For $i = m-1, \dots, n$, define a type decoration t'_i for M_i by

$$t'_{m-1} = t_{m-1},$$
$$(M_{i-1}, t'_{i-1}) \xrightarrow{w_i}_\beta (M_i, t'_i) \quad \text{for } i = m, \dots, n.$$

Then it is easy to see $t_{m-1}(w_m 1) = t'_i(u_i) = t'_i(u'_i)$ for all $i = m, \dots, n$. Since $t_{m-1}(w_m 1)$ is an atomic type, it follows that u_n and u'_n are pivots. So we have proved the above claim, with $u = u_n$ and $u' = u'_n$.

Now we show that v and v' are similar by induction on $|v| - |u| + |v'| - |u'|$. Let s, s' be such that $v = us$ and $v' = u's'$. If $s = s'$, then v and v' are homologous and hence similar. Suppose $s \neq s'$. By Lemma 3.58, the nodes $v = us$ and us' of $|M|_\beta$ share a common ancestor in M. By the above claim applied to us, us' in place of v, v', we must have $s = s_1 s_2, s' = s'_1 s'_2$, $s_1 \neq s'_1$, $us_1 \cong_{|M|_\beta} us'_1$, and us_1 and us'_1 are pivots. Since $|us| - |us_1| + |us'| - |us'_1| = |s_2| + |s'_2| < |s| + |s'| = |v| - |u| + |v'| - |u'|$, the induction hypothesis applies; hence us and us' are similar. Since us' and $u's'$ are homologous, it follows that us and $u's'$ are similar. □

Lemma 3.61. *Let $M = (\mathcal{T}, f, b) \in \Lambda(\Sigma)$ be a closed typable λ-term in η-long form, and let $M^\circ = \textsc{Collapse}(M)$. Suppose that u_1 and u_2 are distinct nodes of M such that $u_1 \approx^*_M u_2$. Unless u_1 is a pivot and $u_1 \in \mathrm{dom}(b)$, u_1 and u_2 share a common ancestor in M°.*

Proof. Clearly, it suffices to consider the case where $u_1 \approx_M u_2$. There must be a pair of pivots s_1, s_2 such that $s_1 \cong_M s_2$ and for some u, $u_1 = s_1 u$ and $u_2 = s_2 u$. By the assumption about u_1, u_2, we have $s_1, s_2 \notin \mathrm{dom}(b)$. At each stage of the execution of Algorithm 1, let u'_1, u'_2, s'_1, s'_2 be the ancestors of u_1, u_2, s_1, s_2, respectively, in M°. By Lemma 3.56, s'_1 and s'_2 are pivots and $s'_1 \cong_{M^\circ} s'_2$. We must have $s'_1 = s'_2$ at the end of the execution of Algorithm 1. Since the nodes $v_1, v_2, \dots, v_k$ picked in lines 5–6 of the algorithm are duplicated pivots of maximal height in M°, we cannot have $s'_1 < v_i$ or $s'_2 < v_i$ for some $i \in \{1, 2, \dots, k\}$ until $s'_1 = s'_2$. Hence, until $s'_1 = s'_2$, we have $u'_1 = s'_1 u$ and $u'_2 = s'_2 u$. This means that at the first stage where $s'_1 = s'_2$ holds, we have $u'_1 = u'_2$. Therefore, u_1 and u_2 have the same ancestor. □

Lemma 3.62. *Let $M \in \Lambda(\Sigma)$ be a closed λ-term in η-long β-normal form and $M^\circ = \textsc{Collapse}(M)$. Suppose $M' \twoheadrightarrow_\beta M$ by almost non-duplicating β-reduction. Let $m = |\overrightarrow{\mathrm{Con}}(M^\circ)|$ and $n = |\overrightarrow{\mathrm{Con}}(M')|$. The following hold:*

(i) $|\widehat{M^\circ}[y_1,\dots,y_m]|_\beta = |\widehat{M'}[y_{g(1)},\dots,y_{g(n)}]|_\beta$ *for some* $g\colon \{1,\dots,n\} \to \{1,\dots,m\}$.

(ii) *If M' is almost affine, then so is M°.*

Proof. (i) Consider two occurrences v_1, v_2 of a free variable z_i in $|\widehat{M'}[z_1,\dots,z_n]|_\beta$. Since each free variable occurs just once in $\widehat{M'}[z_1,\dots,z_n]$, the unique occurrence u of z_i in $\widehat{M'}[z_1,\dots,z_n]$ is the common ancestor of the nodes v_1 and v_2 of $|\widehat{M'}[z_1,\dots,z_n]|_\beta$. This means that the node u of M' is the common ancestor of the nodes v_1 and v_2 of M. By Lemma 3.60, we have $v_1 \approx_M^* v_2$. Since some constant occurs at v_1 and v_2 in M, Lemma 3.61 implies that v_1 and v_2 have the same ancestor in M°. This means that the same free variable occurs at v_1 and v_2 in $|\widehat{M^\circ}[y_1,\dots,y_m]|_\beta$. Therefore, there is a function $g\colon \{\, i \mid z_i \in \mathrm{FV}(|\widehat{M'}[z_1,\dots,z_n]|_\beta)\,\} \to \{1,\dots,m\}$ such that if v is an occurrence of z_i in $|\widehat{M'}[z_1,\dots,z_n]|_\beta$, then v is an occurrence of $y_{g(i)}$ in $|\widehat{M^\circ}[y_1,\dots,y_m]|_\beta$. Some z_i may not occur in $|\widehat{M'}[z_1,\dots,z_n]|_\beta$, but by extending g to a function from $\{1,\dots,n\} \to \{1,\dots,m\}$ in an arbitrary way, we have $|\widehat{M^\circ}[y_1,\dots,y_m]|_\beta = |\widehat{M'}[y_{g(1)},\dots,y_{g(n)}]|_\beta$.

(ii) Let t_{M° be a principal type decoration of $M^\circ = (\mathcal{T}_{M^\circ}, f_{M^\circ}, b_{M^\circ})$. Suppose that M° is not almost affine. Then there are two distinct leaves $v_1, v_2 \in \mathcal{T}_{M^\circ}^{(0)}$ such that $b_{M^\circ}(v_1) = b_{M^\circ}(v_2)$ and $t_{M^\circ}(v_1) = t_{M^\circ}(v_2)$ is a non-atomic type. Since M° is in η-long form, we have $v_1 = u_1 0$ and $v_2 = u_2 0$ for some $u_1, u_2 \in \mathcal{T}_{M^\circ}^{(2)}$. Since the β-reduction from M° to $M = (\mathcal{T}_M, f_M, b_M)$ is non-erasing and almost non-duplicating, it is easy to see that by taking the leftmost (i.e., lexicographically first) descendants at each step, we can arrive at $u_1', u_2' \in \mathcal{T}_M^{(2)}$ such that $u_1'0, u_2'0$ are descendants of v_1 and v_2, respectively, and $b_M(u_1'0) = b_M(u_2'0)$. Now let v_1' and v_2' be the ancestors of $u_1'0$ and $u_2'0$, respectively, in $M' = (\mathcal{T}_{M'}, f_{M'}, b_{M'})$. By Lemma 3.4, part (iv), we see that $b_{M'}(v_1') = b_{M'}(v_2')$. Let $t_{M'}$ be a principal type decoration of M'. Since M' is almost affine, either $v_1' = v_2'$ or $t_{M'}(v_1') = t_{M'}(v_2')$ is an atomic type q. Let t_M be a type decoration of M such that $(M', t_{M'}) \twoheadrightarrow_\beta (M, t_M)$. If $t_{M'}(v_1') = t_{M'}(v_2') = q$, then $t_M(u_1'0) = t_M(u_2'0) = q$, contradicting $u_1', u_2' \in \mathcal{T}_M^{(2)}$. Hence $v_1' = v_2'$. By Lemma 3.60, $u_1'0 \approx_M^* u_2'0$. By Lemma 3.61, $v_1 = v_2$, a contradiction. This contradiction shows that M° is almost affine. □

Lemma 3.63. *Let $M \in \Lambda(\Sigma)$ be a closed λI-term in η-long β-normal form relative to γ, and let $M^\circ = \textsc{Collapse}(M)$. The following hold:*

(i) *M° is a λI-term in η-long form.*

(ii) *If $M' \twoheadrightarrow_\beta M$ by non-erasing almost non-duplicating β-reduction, then $M' \in \Lambda(\mathrm{database}(M^\circ), \langle\gamma\rangle(\mathrm{tuple}(M^\circ)))$.*

Proof. Part (i) is by Lemma 3.56, parts (i) and (ii).

For part (ii), let $\overrightarrow{\mathrm{Con}}(M^\circ) = (d_1, \dots, d_m)$ and let

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \alpha$$

be a principal typing of $\widehat{M^\circ}[y_1, \dots, y_m]$. Then $d_i(\overline{\beta_i}) \in \text{database}(M^\circ)$ for $i = 1, \dots, m$, and $\langle\gamma\rangle(\text{tuple}(M^\circ)) = \alpha$. Let $n = |\overrightarrow{\mathrm{Con}}(M')|$. By Lemma 3.62, there is a function $g\colon \{1, \dots, n\} \to \{1, \dots, m\}$ such that

$$|\widehat{M^\circ}[y_1, \dots, y_m]|_\beta = |\widehat{M'}[y_{g(1)}, \dots, y_{g(n)}]|_\beta$$

Since $M' \twoheadrightarrow_\beta M$ by non-erasing almost non-duplicating β-reduction, we also have $\widehat{M'}[y_{g(1)}, \dots, y_{g(n)}] \twoheadrightarrow_\beta |\widehat{M'}[y_{g(1)}, \dots, y_{g(n)}]|_\beta$ by non-erasing almost non-duplicating β-reduction. Then by the Subject Reduction Theorem (Theorem 3.14) and Lemma 3.52, we have

$$\vdash \{\, y_{g(i)} : \beta_{g(i)} \mid 1 \le i \le n \,\} \Rightarrow \widehat{M'}[y_{g(1)}, \dots, y_{g(n)}] : \alpha.$$

This means that $M' \in \Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$. □

Lemma 3.63 does not say that $\Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$ is closed under non-erasing almost non-duplicating β-expansion, but together with Lemma 3.54 implies the following, which corresponds to the special property (28) highlighted in the rough proof sketch given in Section 2.3.

Lemma 3.64. *Let $M \in \Lambda(\Sigma)$ be a closed λ-term in η-long β-normal form relative to γ, and suppose that $M^\circ = \textsc{Collapse}(M)$ is almost linear. Then for every almost linear closed λ-term $M' \in \Lambda(\Sigma)$ in η-long form relative to γ, $M' \twoheadrightarrow_\beta M$ if and only if $M' \in \Lambda(\text{database}(M^\circ, \langle\gamma\rangle(\text{tuple}(M^\circ)))$.*

Proof. First note that $M \in \Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$ by Lemma 3.37 and part (i) of Lemma 3.34.

Suppose $M' \twoheadrightarrow_\beta M$. By Lemma 3.49, the leftmost β-reduction from M' to $M = |M'|_\beta$ is non-erasing and almost non-duplicating. Lemma 3.63 then implies $M' \in \Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$.

Conversely, suppose $M' \in \Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$. By part (i) of Lemma 3.34 again, $|M'|_\beta \in \Lambda(\text{database}(M^\circ), \langle\gamma\rangle(\text{tuple}(M^\circ)))$. Since by Lemma 3.17 $|M'|_\beta$ must be in η-long form relative to γ, Lemma 3.54 implies $|M'|_\beta = M$. □

The following theorem is the main result of the paper.

Theorem 3.65. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be an almost linear CFLG. Suppose that $N \in \Lambda(\Sigma)$ is a λ-term in η-long β-normal form relative to $f(S)$. Then the following are equivalent:*

(i) *$N \in L(\mathscr{G})$.*

(ii) *$N^\circ = \textsc{Collapse}(N)$ is almost linear and* $\text{program}(\mathscr{G}) \cup \text{database}(N^\circ) \vdash S(\text{tuple}(N^\circ))$.

Proof. (i) $\Rightarrow$ (ii). Suppose $N \in L(\mathscr{G})$. Then there is a closed λ-term $P \in \Lambda(\Sigma)$ in η-long form such that $\vdash_{\mathscr{G}} S(P)$ and $P \twoheadrightarrow_\beta N$. Since $\mathscr{G}$ is almost linear, P is almost linear. By part (ii) of Lemma 3.62 and part (i) of Lemma 3.63, $N^\circ = \textsc{Collapse}(N)$ is almost linear. By Lemma 3.64, $P \in \Lambda(\text{database}(N^\circ), \langle f(S)\rangle(\text{tuple}(N^\circ)))$. Lemma 3.35 then implies $\text{program}(\mathscr{G}) \cup \text{database}(N^\circ) \vdash S(\text{tuple}(N^\circ))$.

(ii) $\Rightarrow$ (i). By Lemma 3.35, there is an almost linear λ-term $P \in \Lambda(\Sigma)$ in η-long form such that $\vdash_{\mathscr{G}} S(P)$ and $P \in \Lambda(\text{database}(N^\circ), \langle f(S)\rangle(\text{tuple}(N^\circ)))$. Since N° is almost linear, Lemma 3.64 implies that $P \twoheadrightarrow_\beta N$. Therefore, $N \in L(\mathscr{G})$. □

Notice that when $\mathscr{G}$ is a linear CFLG and N is a linear λ-term, Theorems 3.42 and 3.65 both hold of $\mathscr{G}$ and N, even though $N^\circ \neq N$ in general.

Let us turn to the complexity analysis of the reduction. Since it is easy to see that Algorithm 1 runs in polynomial time, an immediate corollary to Theorem 3.65 is that the language of every almost linear CFLG belongs to the complexity class P. As in the linear case, we can obtain a tight complexity upper bound. Recall that $\prec$ is the lexicographic order on $\{0,1\}^*$. We write $\approx_M \cap \prec$ for the intersection of the two relations $\approx_M$ and $\prec$, thought of as sets of ordered pairs.

Lemma 3.66. *Let $M = (\mathcal{T}, f, b)$ be a λ-term, and suppose that u_1 $(\approx_M \cap \prec)$ v and u_2 $(\approx_M \cap \prec)$ v. Then there exists a v' such that*

(i) *either v' $(\approx_M \cap \prec)$ u_1 or $v' = u_1$, and*

(ii) *either v' $(\approx_M \cap \prec)$ u_2 or $v' = u_2$.*

Proof. There are $\hat{u}_1, \hat{u}_2$ and pivots w_1, w, s, s_2 such that

$$\begin{array}{llll} w_1 \cong_M w, & w_1 \prec w, & u_1 = w_1\hat{u}_1, & v = w\hat{u}_1, \\ s_2 \cong_M s, & s_2 \prec s, & u_2 = s_2\hat{u}_2, & v = s\hat{u}_2. \end{array}$$

Since w and s are prefixes of v, either $w \leq s$ or $s \leq w$. We may assume $w \leq s$. We have

$$s = w\hat{s}, \quad \hat{u}_1 = \hat{s}\hat{u}_2$$

for some $\hat{s}$.

Case 1. There is an s' such that $w \leq b(ss') < s$. Since $s_2 \cong_M s$, it must be the case that $w \leq b(ss') = b(s_2 s') < s_2$. So there is an $\hat{s}_2$ such that

$$s_2 = w\hat{s}_2, \quad \hat{s}_2 \prec \hat{s}.$$

Since $w\hat{s}_2 = s_2 \cong_M s = w\hat{s}$ and $w_1 \cong_M w$, the definition of the congruence relation $\cong_M$ implies that $w_1\hat{s}_2$ and $w_1\hat{s}$ are pivots and

$$w_1\hat{s}_2 \cong_M w_1\hat{s}.$$

Therefore,

$$u_1 = w_1\hat{u}_1 = w_1\hat{s}\hat{u}_2 \approx_M w_1\hat{s}_2\hat{u}_2 \approx_M w\hat{s}_2\hat{u}_2 = s_2\hat{u}_2 = u_2.$$

Let $v' = w_1\hat{s}_2\hat{u}_2$. Since $\hat{s}_2 \prec \hat{s}$ and $w_1 \prec w$, we have $v' \prec u_1$ and $v' \prec u_2$.

Case 2. There is no s' such that $w \leq b(ss') < s$. Since $w_1 \cong_M w$, we must have

$$s_2 \cong_M s = w\hat{s} \cong_M w_1\hat{s},$$

and $w_1\hat{s}$ is a pivot. Therefore,

$$u_2 = s_2\hat{u}_2 \approx_M w_1\hat{s}\hat{u}_2 = w_1\hat{u}_1 = u_1,$$

and the conclusion clearly holds with either $v' = u_1$ or $v' = u_2$. □

The next lemma easily follows from Lemma 3.66.

Lemma 3.67. *If $u \approx_M^* v$, then there exists a w such that $w\ (\approx_M \cap \prec)^*\ u$ and $w\ (\approx_M \cap \prec)^*\ v$.*

Lemma 3.68. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be an almost linear CFLG. There is a log-space-bounded deterministic Turing machine that takes as input a λ-term $N \in \Lambda(\Sigma)$ in η-long β-normal form relative to $f(S)$ and decides whether Algorithm 1 returns an almost linear N°, and if so, computes* $(\text{database}(N^\circ), S(\text{tuple}(N^\circ)))$.

Proof (sketch). Let $N = (\mathcal{T}, f, b)$. We assume that N is given as a λ-expression as before. We must avoid computing the output $N^\circ = \textsc{Collapse}(N)$ of Algorithm 1 explicitly. By Lemmas 3.60 and 3.61, N° is almost affine if and only if for every pair of nodes $v, v' \in \mathcal{T}^{(0)}$ such that $b(v) = b(v')$, it holds that $v \approx_N^* v'$. By Lemma 3.67, this is so if and only if $w\ (\approx_N \cap \prec)^*\ v'$, where w is the leftmost node such that $w\ (\approx_N \cap \prec)^*\ v$.[37] Checking whether two nodes are homologous clearly requires no

[37]In fact, by refining the proof of Lemma 3.60, it is not hard to see that it suffices to take the leftmost w such that $b(w) = b(v) = b(v')$ and check $w\ (\approx_N \cap \prec)^*\ v$ and $w\ (\approx_N \cap \prec)^*\ v'$.

more than logarithmic space, so the relation $(\approx_N \cap \prec)^*$ can be decided in logarithmic space as well. It follows that the similarity of v and v' can also be checked in logarithmic space. It is also easy to see that N° is a λI-term if and only if N is, and clearly this can be checked in logarithmic space.

Now suppose that N is λI and N° is almost linear. Let $\overrightarrow{\mathrm{Con}}(N) = (d_1, \dots, d_m)$, and let

$$y_1 : \beta_1, \dots, y_m : \beta_m \Rightarrow \beta_0$$
$$z_1 : \gamma_1, \dots, z_n : \gamma_n \Rightarrow \gamma_0$$

be principal typings of $\widehat{N}[y_1, \dots, y_m]$ and $\widehat{N^\circ}[z_1, \dots, z_n]$, respectively. Let $\{w_1, \dots, w_m\} = \mathrm{dom}(f)$, where $w_1 \prec \dots \prec w_m$. (We have $f(w_i) = d_i$.) Let

$$I = \{\, i \in \{1, \dots, m\} \mid \text{there is no } k < i \text{ such that } w_k \approx_N^* w_i \,\}.$$

Let $g\colon \{1, \dots, m\} \to I$ be the function such that $w_{g(i)} \approx_N^* w_i$ for $i \in \{1, \dots, m\}$. By Lemmas 3.60 and 3.61, we must have a bijection $h\colon \{1, \dots, n\} \to I$ such that $\widehat{N^\circ}[y_{h(1)}, \dots, y_{h(n)}] \twoheadrightarrow_\beta \widehat{N}[y_{g(1)}, \dots, y_{g(m)}]$. Let σ be a most general unifier of

$$\{\, (\beta_i, \beta_j) \mid i \in I, w_i \approx_N^* w_j \,\}$$

Then

$$\{\, y_i : \beta_i\sigma \mid i \in I \,\} \Rightarrow \beta_0\sigma \tag{60}$$

is a principal typing of $\widehat{N}[y_{g(1)}, \dots, y_{g(m)}]$. By Lemma 3.53, (60) is a principal typing of $\widehat{N^\circ}[y_{h(1)}, \dots, y_{h(n)}]$ as well, and we have

$$\mathrm{database}(N^\circ) = \{\, d_i(\overline{\beta_i\sigma}) \mid i \in I \,\},$$
$$\mathrm{tuple}(N^\circ) = \overline{\beta_0\sigma}.$$

By Theorem 3.46, (60) is negatively non-duplicated. For every $i \in \{0, \dots, m\}$ and $v \in \langle\beta_i\rangle^{(0)}$, let $p_{i,v} = \mathrm{subtype}(\beta_i, v)$. Then if $(i_1, v_1), (i_2, v_2)$ are distinct negative occurrences and $i_1, i_2 \in \{0\} \cup I$, then $p_{i_1,v_1}\sigma \neq p_{i_2,v_2}\sigma$. Now consider a positive occurrence (i, v) of $p_{i,v}$ such that $i \in \{0\} \cup I$. The fact that $\widehat{N}[y_1, \dots, y_m]$ is a λI-term implies that in $(\widehat{N}[y_1, \dots, y_m], \hat{t})$, where $\hat{t}$ is a principal type decoration of $\widehat{N}[y_1, \dots, y_m]$, the occurrence (i, v) is linked to some negative occurrence (i', v') of $p_{i,v} = p_{i',v'}$. If $i' \in \{1, \dots, m\}$, let $j = g(i')$; otherwise let $j = i' = 0$. Then it must be that $p_{i,v}\sigma = p_{i',v'}\sigma = p_{j,v'}\sigma$. Note that although (i, v) may be linked to more than one (i', v') in $(\widehat{N}[y_1, \dots, y_m], \hat{t})$, the pair (j, v') is uniquely determined independently of the choice of (i', v') because (60) is negatively non-duplicated.

As in the proof of Lemma 3.43, a deterministic log-space-bounded Turing machine can compute a negative occurrence (i', v') linked to a positive occurrence (i, v) by following edges of $\overline{G}_{(\widehat{N}[y_1,\dots,y_m],\hat{t})} = \overline{G}_{(\widehat{N}[y_1,\dots,y_m],t)}$, where t is the type decoration for $y_1 : \tau(d_1), \dots, y_m : \tau(d_m) \Rightarrow \hat{N}[y_1, \dots, y_m] : f(S)$. Again, the type decoration t is not explicitly computed. There may be more than one maximal directed path starting from (i, v), but any such path will do, so the machine simply picks the first relevant edge that it can find at each point. Once the machine reaches a configuration representing $(w_{i'}, v', \uparrow)$, it can then find the least j such that $w_j \ (\approx_N \cap \prec)^* \ w_{i'}$ using no more than logarithmic space. □

Theorem 3.69. *For every almost linear CFLG $\mathscr{G}$, $L(\mathscr{G})$ belongs to LOGCFL.*

Proof. The proof is similar to that of Theorem 3.44. Note that if $P \twoheadrightarrow_\beta N$ by non-erasing β-reduction, then $|\overrightarrow{\mathrm{Con}}(P)| \leq |\overrightarrow{\mathrm{Con}}(N)|$. This implies that whenever $\mathrm{program}(\mathscr{G}) \cup \mathrm{database}(N^\circ) \vdash S(\mathrm{tuple}(N^\circ))$ holds, there is a derivation tree for it whose size is bounded by a polynomial in the number of occurrences of constants in N. □

4 Some consequences and extensions

4.1 Further complexity-theoretic consequences

We have seen that the problem of recognition for a fixed almost linear CFLG is in LOGCFL. Since there is a context-free language that is LOGCFL-complete [27], it follows that LOGCFL is a tight upper bound on the computational complexity of fixed almost linear CFLG recognition.

Let us sketch some further complexity-theoretic consequences of this work. These concern three different types of problems: (i) the problem of uniform recognition for subclasses of almost linear CFLGs, (ii) the problem of parsing for a fixed almost linear CFLG, and (iii) the problem of finding one target λ-term from an input λ-term for a fixed almost linear synchronous CFLG.

4.1.1 Uniform recognition

If the grammar is not fixed and is part of the input, the recognition problem (known as *uniform recognition*) is known to be P-complete for general context-free grammars, and PSPACE-complete for non-deleting multiple context-free grammars [38, 39]. Since it is easy to translate non-deleting multiple context-free grammars into linear CFLGs, the latter gives a lower bound on the complexity of uniform recognition for almost linear CFLGs. The EXPTIME-completeness of the program complexity of general Datalog query evaluation [16] provides an upper bound; currently I do not know whether either of these bounds is tight, however.

A lower complexity bound for uniform recognition can be obtained for restricted subclasses of almost linear CFLGs. We call a Datalog program $\mathbf{P}$ *k-bounded* if k is at least as large as the maximal arity of predicates in $\mathbf{P}$ and the number of variables in any rule of $\mathbf{P}$. For a k-bounded Datalog program $\mathbf{P}$, the number of work tapes needed in the "storage area" in the log-space-bounded ATM $\mathscr{M}_{\mathbf{P}}$ simulating $\mathbf{P}$ does not exceed k. (The description of $\mathscr{M}_{\mathbf{P}}$ was given in Section 3.1.1.) With additional work tapes to serve as pointers to rules and predicates in the Datalog program, the program can be moved from the finite control of the ATM to part of the input. The resulting log-space-bounded ATM can decide, given input $(\mathbf{P}, D, q)$ with $\mathbf{P}$ k-bounded, whether $\mathbf{P} \cup D \vdash q$. Now consider the class of *k-bounded* almost linear CFLGs, i.e., almost linear CFLGs $\mathscr{G}$ such that program($\mathscr{G}$) is k-bounded. As in the proof of Lemma 3.68, it is clear that the translation from $\mathscr{G}$ to program($\mathscr{G}$) can be done in logarithmic space. This means that there is a log-space reduction from the uniform recognition problem for k-bounded almost linear CFLGs to a problem in ALOGSPACE = P. Since uniform recognition for CFGs whose rules are all of the form $A \to BC$ or $A \to \epsilon$ is already P-complete [36], it follows that the uniform recognition problem for k-bounded almost linear CFLGs is P-complete.

It is folklore [55] that the uniform recognition problem for the class of context-free grammars without ϵ-productions is in LOGCFL. What corresponds to an ϵ-production in the case of CFLGs is a rule of the form

$$B(M)$$

(with an empty right-hand side) where M is a pure λ-term. We can eliminate all such *ϵ-rules* from an almost linear CFLG by the same method that Kanazawa and Yoshinaka [47] used for linear CFLGs, so the uniform recognition problem for the class of almost linear CFLGs without ϵ-rules is of interest. If $\mathscr{G}$ is such a CFLG, then all leaves of Datalog derivation trees for program($\mathscr{G}$) are extensional nodes. By the analysis in the proof of Lemma 3.2, we can show that the uniform recognition problem for the class of ϵ-free k-bounded almost linear CFLGs is in LOGCFL.

4.1.2 Parsing

It is also interesting to ask the computational complexity of parsing, as opposed to recognition. *Functional LOGCFL* (written $\mathrm{FL}^{\mathrm{LOGCFL}}$) is the class of solution search problems that can be solved by a deterministic log-space-bounded Turing machine with a LOGCFL oracle [26]. It is a natural functional analogue of LOGCFL. Gottlob et al. [26] show that given an ATM $\mathscr{M}$ with simultaneous log-space and poly-size bounds, the problem of finding a first accepting computation tree of $\mathscr{M}$ on input w (within a given polynomial size bound) is in functional LOGCFL. In the course of proving this result, they also show that the set of all accepting computation trees (within a given polynomial size bound), in the form of a 'shared forest', can be computed by a log-space-bounded Turing machine with a LOGCFL oracle. We can use this result to show that the problem of parsing for a fixed almost linear CFLG is in functional LOGCFL, but here we opt to give the following direct proof, which is straightforward and more informative.

Let $\mathbf{P}$ be a Datalog program, D an extensional database for $\mathbf{P}$, and q a ground fact. The 'shared forest' representation of the set of all derivation trees for $\mathbf{P} \cup D \vdash q$ is just the set F all ground instances

$$p(\vec{s}) :\!-\; p_1(\vec{s}_1), \dots, p_l(\vec{s}_l)$$

of rules $p(\vec{x}) :\!-\; p_1(\vec{x}_1), \dots, p_l(\vec{x}_l) \in \mathbf{P}$ that can appear in some derivation tree for $\mathbf{P} \cup D \vdash q$ which use only constants from $D \cup \{q\}$.[38] Suppose that the number of extensional nodes in any derivation tree for $\mathbf{P} \cup D \vdash q$ is bounded by a number k, depending only on D. In order to see whether $p(\vec{s}) :\!-\; p_1(\vec{s}_1), \dots, p_l(\vec{s}_l)$ is in F, one need only check whether there are derivation trees (with no more than k extensional nodes) for

$$\mathbf{P} \cup D \vdash p_i(\vec{s}_i) \quad (i = 1, \dots, l)$$

and one for

$$\mathbf{P} \cup \{p(\vec{s})\} \cup D \vdash q$$

in which $p(\vec{s})$ appears on exactly one of its leaves. Let $g(n)$ be the polynomial that Lemma 3.2 associates with $\mathbf{P}$. Then derivation trees for $\mathbf{P} \cup D \vdash p_i(\vec{s}_i)$ $(i = 1, \dots, l)$ can be found from among those with at most $g(k)$ nodes, if there are any. It is not hard to see that the same reasoning as in the proof of Lemma 3.2 shows that the minimal size of the required kind of derivation tree for $\mathbf{P} \cup \{p(\vec{s})\} \cup D \vdash q$ can be bounded by $g(k + 1)$. Thus, answers to these questions can be obtained through oracle queries to two sets

$$\{\, (D, q_1, \mathtt{1}^m) \mid \text{there is a derivation tree for } \mathbf{P} \cup D \vdash q_1 \text{ of size} \leq g(m) \,\},$$
$$\{\, (\{q_2\} \cup D, q_1, \mathtt{1}^m) \mid \text{there is a derivation tree for } \mathbf{P} \cup \{q_2\} \cup D \vdash q_1 \text{ of size} \leq g(m) \text{ with } q_2 \text{ on exactly one leaf} \,\}.$$

[38]It would be more appropriate to call the set F the "reduced" shared forest, since a shared parse forest in general may contain useless elements.

The former is in LOGCFL by Lemma 3.1. A slight modification of its proof shows that the latter is in LOGCFL, too, and it is easy to combine the two into a single LOGCFL oracle. Thus, if $(D, q, \mathtt{1}^k)$ is given as input, the set F can be computed in logarithmic space with a LOGCFL oracle by cycling through all ground instances of all rules in $\mathbf{P}$.

Let $\mathbf{P} = \text{program}(\mathscr{G})$ for some almost linear CFLG, and suppose (D, q) is obtained from a λ-term N as in Theorems 3.65 and 3.69. Then we can take the number k to be $|\overrightarrow{\text{Con}}(N)|$, and the set F can be computed in logarithmic space with a LOGCFL oracle.

With the one-one correspondence between the rules of the Datalog program program($\mathscr{G}$) and the rules of the CFLG $\mathscr{G}$, the set F can also be taken to be a shared forest representation of the set of all derivation trees of $\mathscr{G}$ for the input λ-term N. Thus, given an almost CFLG $\mathscr{G}$, the problem of computing the shared forest of all derivation trees of $\mathscr{G}$ for an input λ-term N is in functional LOGCFL.

4.1.3 Transduction with almost linear synchronous CFLGs

Suppose we are given a synchronous CFLG consisting of a pair of almost linear CFLGs. Given an input λ-term M generated by one of the component CFLGs (call it the "source-side" grammar), the set of all derivation trees of M can be efficiently computed in the form of a shared forest, as we have seen above. In order to find a "target-side" λ-term N that the synchronous grammar pairs with M, we can take one of the derivation trees T, construct a λ-term P that the "target-side" CFLG associates with T, and then compute the β-normal form $N = |P|_\beta$ of P. It is of course impossible to explicitly enumerate all such N, because there may be infinitely many derivation trees of M; nor is there any simple "packed" representation of all such N (because the set of all such N is in general as complex as the language of an arbitrary almost linear CFLG). Let us therefore consider the computational complexity of finding *one* λ-term N that the synchronous grammar pairs with M.

As in [26], a deterministic log-space-bounded Turing machine with a LOGCFL oracle can compute a single derivation tree T of M (whenever there is one). It is easy to see that given a derivation tree T, the λ-term P that the target-side grammar associates with T can be computed in logarithmic space. Although the size of $|P|_\beta$ is in general exponential in the size of P and so it is not feasible to compute $|P|_\beta$ explicitly, the pair $(\text{database}(P), \text{tuple}(P))$ can be computed in logarithmic space as in the proof of Lemma 3.68. Since P is almost linear, by Lemma 3.54, $|P|_\beta$ is the only λ-term in η-long β-normal form in $\Lambda(\text{database}(P), \langle\gamma\rangle(\text{tuple}(P)))$, where γ is the type that the target-side grammar assigns to P. So $(\text{database}(P), \text{tuple}(P))$ serves as a kind of compact representation of $|P|_\beta$. (In fact, when $|P|_\beta$ is a tree,

$(\text{database}(P), \text{tuple}(P))$ is a representation of a term graph that unfolds to (the hypergraph representation of) $|P|_\beta$.) All in all, given a fixed almost linear synchronous CFLG, the problem of finding one target-side λ-term N corresponding to an input source-side λ-term M is in functional LOGCFL, if we allow as output a compact representation of N in the form of a pair of a database and a tuple of constants.

Note that in the special case where P is linear and $|P|_\beta$ is an encoding of a string or a tree, $(\text{database}(P), \text{tuple}(P))$ is nothing but an explicit hypergraph representation of the latter. Thus, with respect to a fixed synchronous grammar consisting of a linear string grammar (e.g., a CFG or MCFG) and an almost linear Montague semantics, the problem of explicitly computing one surface realization of an input logical form is in functional LOGCFL.

4.2 Regular sets as input

4.2.1 Parsing as intersection for linear CFLGs

In ordinary parsing/recognition of string languages, it is sometimes useful to allow as input a regular set of strings (usually represented as a finite automaton), rather than a single string. The resulting generalization of the problem is a key element of the view of “parsing as intersection”, where the “shared parse forest” that is the output of parsing is given in the form of a grammar generating the intersection of the language of the original grammar and the input regular set. Various dynamic parsing techniques can then be regarded as variants of Bar-Hillel et al.’s [5] original proof of the closure of the context-free languages under intersection with regular sets [52].

Many well-known grammar formalisms, including context-free grammars, tree-adjoining grammars [37], (parallel) multiple context-free grammars [66], and IO macro grammars [24], have the property that given a regular set R, any grammar G can be “specialized” into a grammar G' generating the intersection of the language of G and R, in such a way that G is the image of G' under a simple “projection” that maps nonterminals of G' to nonterminals of G. Kanazawa [40] has shown that the same property holds of de Groote’s [17] abstract categorial grammars. *Linear* context-free λ-term grammars are nothing but abstract categorial grammars whose abstract vocabulary is second-order. Via encoding in linear CFLGs, Kanazawa’s [40] result provides a uniform proof of closure under intersection with regular sets for *linear* formalisms such as context-free grammars, (multi-component) tree-adjoining grammars, and multiple context-free grammars.

Theorem 3.40 shows how the recognition problem for linear CFLGs in a generalized form, where an input is a set of λ-terms represented by a pair of a database and a type, reduces to Datalog query evaluation. It is easy to see that any regular set of strings or trees can be represented in this way. In the string case, a nondeterministic finite automaton with an initial state q_I and just one final state q_F translates into the pair $(D, q_F \to q_I)$, where D is the database consisting of all facts of the form $c(q, r)$ such that the automaton has a transition from state q to state r labeled by c. In the tree case, a nondeterministic bottom-up finite automaton with a unique final state q_F translates into the pair (D, q_F), where D is the database consisting of all facts $f(q, q_n, \dots, q_1)$ such that the automaton has a transition rule $f(q_1(x_1), \dots, q_n(x_n)) \to q(f(x_1, \dots, x_n))$. More generally, any set of λ-terms that can be expressed as the set $\Lambda(D, \alpha)$ with a database D and a type α can be used as an input to recognition with a linear CFLG.

With Lemma 3.31, the problem of parsing in this generalized setting reduces to the problem of computing (a representation of) the set of all derivation trees from a Datalog program $\mathbf{P}$ and a database D. The connection to parsing as intersection is that the specialized grammar generating the intersection language corresponds to the propositional Horn clause program consisting of the database D and an appropriate subset of the set $\bigcup_{\pi \in \mathbf{P}} \text{ground}(\pi, U_D)$ of ground instances of rules in $\mathbf{P}$.

4.2.2 Almost linear CFLGs and deterministic databases

As we noted, Theorem 3.40 does not hold of almost linear CFLGs, because there is no analogue of part (ii) of Lemma 3.34 for non-erasing almost non-duplicating β-reduction: if D is a database over $\mathcal{D}_{\Sigma,U}$ and $\alpha \in \mathscr{T}(A)$, the set $\Lambda(D, \alpha)$ is not always closed under the converse of non-erasing almost non-duplicating β-reduction. One sufficient condition for this closure property to hold is given by the following definition:

- D is said to be *deterministic* if for all types $\gamma_1, \dots, \gamma_m$ and all atomic types p, q,
$$\Lambda(D, \gamma_1 \to \dots \to \gamma_m \to p) \cap \Lambda(D, \gamma_1 \to \dots \to \gamma_m \to q) \neq \varnothing$$
implies $p = q$.

It is not difficult to show that determinism is a decidable property of databases, but I leave a detailed analysis of this notion for another occasion.[39]

Lemma 4.1. *Let $\Sigma = (A, C, \tau)$ be a higher-order signature, $M, M' \in \Lambda(\Sigma)$ be typable λ-terms, and D be a deterministic database over $\mathcal{D}_{\Sigma,U}$. If $M' \twoheadrightarrow_\beta M$ by non-erasing almost non-duplicating β-reduction, then $M \in \Lambda(D, \alpha)$ implies $M' \in \Lambda(D, \alpha)$.*

[39] In particular, I have been unable to settle the question whether database(N°) is deterministic whenever N° is almost linear.

Proof. Let $M = (\mathcal{T}, f, b), M' = (\mathcal{T}', f', b')$ be typable closed λ-terms, and let t' be a principal type decoration of M'. Assume $M \in \Lambda(D, \alpha)$ and $M' \overset{w}{\rightarrow}_\beta M$ by a non-erasing almost non-duplicating one-step β-reduction. Let $\{v_1, \dots, v_k\} = \{\, v \mid b'(w00v) = w0 \,\}$. By assumption, $k \geq 1$. Since the case $k = 1$ is taken care of by Lemma 3.34, part (ii), assume $k \geq 2$ and $t'(w1) = p$ for some atomic type p. Since $M \in \Lambda(D, \alpha)$, there is a type decoration $\hat{t}$ for

$$z_1 : \delta_1, \dots, z_m : \delta_m \Rightarrow \widehat{M}[z_1, \dots, z_m] : \alpha,$$

where $\overrightarrow{\mathrm{Con}}(M) = (c_1, \dots, c_m)$ and for $i = 1, \dots, m$, we have $c_i(\overline{\delta_i}) \in D$ and $\langle \tau(c_i) \rangle = \langle \delta_i \rangle$.

To show that $M' \in \Lambda(D, \alpha)$, it suffices to prove that

$$\hat{t}(wv_1) = \hat{t}(wv_i) \quad \text{for all } i \in \{1, \dots, n\}. \tag{61}$$

For, if (61) holds, then it is easy to see that there are a subset $\{i_1, \dots, i_{m'}\}$ of $\{1, \dots, m\}$ and a function $g\colon \{1, \dots, m\} \to \{i_1, \dots, i_{m'}\}$ satisfying the following conditions:

$$\begin{gathered} c_i = c_{g(i)} \quad \text{for all } i \in \{1, \dots, m\}, \\ \widehat{M'}[z_{i_1}, \dots, z_{i_{m'}}] \overset{w}{\rightarrow}_\beta \widehat{M}[z_{g(1)}, \dots, z_{g(m)}], \\ \vdash z_{i_1} : \delta_{i_1}, \dots, z_{i_{m'}} : \delta_{i_{m'}} \Rightarrow \widehat{M}[z_{g(1)}, \dots, z_{g(m)}] : \alpha, \\ \vdash z_{i_1} : \delta_{i_1}, \dots, z_{i_{m'}} : \delta_{i_{m'}} \Rightarrow \widehat{M'}[z_{i_1}, \dots, z_{i_{m'}}] : \alpha, \\ \widehat{M'}[c_{i_1}, \dots, c_{i_{m'}}] = M'. \end{gathered}$$

The reasoning here is similar to that in the proof of Lemma 3.52.

We prove (61). Let ℓ' be a writing of M'. There exists a writing ℓ of M that agrees with ℓ' on $\{\, u \in \mathcal{T}'^{(1)} \mid u < w \,\}$ such that $\mathrm{sub}_{M',\ell'}(w1) = \mathrm{sub}_{M,\ell}(wv_i)$ for $i = 1, \dots, k$. Let $N = \mathrm{sub}_{M',\ell'}(w1)$ and let n be the number of occurrences of constants in N. Clearly, we have a function $h\colon \{1, \dots, k\} \times \{1, \dots, n\} \to \{1, \dots, m\}$ such that

$$\mathrm{sub}_{\widehat{M}[z_1,\dots,z_m],\ell}(wv_i) = \widehat{N}[z_{h(i,1)}, \dots, z_{h(i,n)}].$$

Let $\mathrm{FV}(N) = \{y_1, \dots, y_r\}$. Then ℓ and $\hat{t}$ determine types $\gamma_1, \dots, \gamma_r$ such that

$$\vdash y_1 : \gamma_1, \dots, y_r : \gamma_r, z_{h(i,1)} : \delta_{h(i,1)}, \dots, z_{h(i,n)} : \delta_{h(i,n)} \Rightarrow \widehat{N}[z_{h(i,1)}, \dots, z_{h(i,n)}] : \hat{t}(wv_i).$$

Similarly, ℓ' and t' determine types $\gamma'_1, \dots, \gamma'_r$ such that

$$\vdash_\Sigma y_1 : \gamma'_1, \dots, y_r : \gamma'_r \Rightarrow N : p.$$

There are two cases to consider.

Case 1. $|N|_\beta = y_j\vec{Q}$ and $\gamma'_j = \vec{\beta'} \to p$. Then we must have $\gamma_j = \vec{\beta} \to \hat{t}(wv_i)$ for all $i \in \{1,\dots,k\}$. Hence $\hat{t}(wv_1) = \hat{t}(wv_i)$ for all $i \in \{1,\dots,k\}$.

Case 2. $|N|_\beta = c_j\vec{Q}$ and $\tau(c_j) = \vec{\beta'} \to p$. Then for all $i \in \{1,\dots,k\}$, $|\widehat{N}[z_{h(i,1)},\dots,z_{h(i,n)}]|_\beta = z_{h(i,1)}\vec{P_i}$ and $\delta_{h(i,1)} = \vec{\beta_i} \to \hat{t}(wv_i)$ for some $\vec{P_i}$ and $\vec{\beta_i}$ such that $\vec{\beta'}$ and $\vec{\beta_i}$ are sequences of types of the same length. Since $c_{h(i,1)} = c_j$, it must be that $\langle\tau(c_j)\rangle = \langle\delta_{h(i,1)}\rangle$, which implies that $\hat{t}(wv_i) = q_i$ for some atomic q_i. Then we have

$$\vdash z_{h(i,1)}\!:\!\delta_{h(i,1)},\dots,z_{h(i,n)}\!:\!\delta_{h(i,n)} \Rightarrow \lambda y_1\dots y_r.\widehat{N}[z_{h(i,1)},\dots,z_{h(i,n)}]\!:\!\gamma_1\to\dots\to\gamma_r\to q_i,$$

which implies

$$\lambda y_1\dots y_r.N \in \Lambda(D, \gamma_1 \to \dots \to \gamma_r \to q_i).$$

Since D is deterministic, it follows that $q_1 = q_i$. □

Lemma 4.2. *Let $\Sigma = (A, C, \tau)$ be a higher-order signature, U be a set of database constants, D be a deterministic database over $\mathcal{D}_{\Sigma,U}$, and $\alpha \in \mathscr{T}(A)$. For every almost linear closed λ-term $M \in \Lambda(\Sigma)$, $M \in \Lambda(D,\alpha)$ if and only if $|M|_\beta \in \Lambda(D,\alpha)$.*

Proof. The "only if" direction is by Lemma 3.34, part (i). Since the β-reduction $M \twoheadrightarrow_\beta |M|_\beta$ must be non-erasing and almost non-duplicating (Lemma 3.49), the "if" direction follows from Lemma 4.1. □

Theorem 4.3. *Let $\mathscr{G} = (\mathscr{N}, \Sigma, f, \mathscr{P}, S)$ be an almost linear CFLG and $B \in \mathscr{N}$. Let U be some set of constants, D be a deterministic database over $\mathcal{D}_{\Sigma,U}$, and $\vec{s}$ be a sequence of constants from U such that $|\vec{s}| = |f(S)|$. The following are equivalent:*

(i) $L(\mathscr{G}) \cap \Lambda(D, \langle f(S)\rangle(\vec{s})) \neq \varnothing$.

(ii) $\text{program}(\mathscr{G}) \cup D \vdash S(\vec{s})$.

Proof. The implication from (ii) to (i) is by Lemma 3.36.

(i) $\Rightarrow$ (ii). Assume (i). Then there is an almost linear λ-term $P \in \Lambda(\Sigma)$ such that $\vdash_{\mathscr{G}} S(P)$ and $|P|_\beta \in \Lambda(D, \langle f(S)\rangle(\vec{s}))$. Since P is almost linear, Lemma 4.2 implies $P \in \Lambda(D, \langle f(S)\rangle(\vec{s}))$. Then (ii) follows by Lemma 3.35. □

It is easy to see that if D is a database representing a finite automaton $\mathscr{A}$ (on strings), then D is deterministic if and only if $\mathscr{A}$ is. If D is a database representing a bottom-up tree automaton $\mathscr{A}$, then, again, D is deterministic if and only if $\mathscr{A}$ is. So Theorem 4.3 applies when a regular set is given as input in the form of a deterministic finite (string or tree) automaton.[40] The string case of this result is not useful, however, because every almost linear CFLG generating λ-term encodings of strings is equivalent to some linear CFLG.[41]

With respect to tree languages, almost linear CFLGs are more powerful than linear CFLGs, and can encode grammars that allow copying of subtrees, like IO context-free tree grammars. For these grammars, Theorem 4.3 implies that parsing as intersection where input is given in the form of a deterministic bottom-up tree automaton reduces to Datalog query evaluation.

4.2.3 An application to string grammars with copying

This last point can be exploited to show that there is a way of representing recognition/parsing (of ordinary single-string input) with respect to some *string* grammars with copying operations, such as IO macro grammars and parallel multiple context-free grammars, in terms of Datalog query evaluation, even though Theorem 3.65 is powerless for that purpose. Such a string grammar can always be turned into a corresponding tree grammar that generates a tree language whose yield image is the language of the string grammar. Since tree copying can be represented by almost linear λ-terms, these tree grammars can be encoded in almost linear CFLGs. Moreover, we can associate with every string w a regular set of trees that yield w so that the language of the tree grammar has a non-empty intersection with that set of trees if and only if w is in the language of the original string grammar.

For example, consider the following parallel multiple context-free grammar [66]:[42]

$$\begin{aligned}
&S(x_1 x_2) \mathrel{:-} A(x_1, x_2).\\
&A(\mathtt{1}, \mathtt{0}).\\
&A(x_1 x_2 \mathtt{1}, x_2 \mathtt{0}) \mathrel{:-} A(x_1, x_2).
\end{aligned}$$

[40] When the automaton has more than one final state, non-empty intersection is equivalent to a disjunction of queries of the form "$?\!- S(q_I, q)$" (in the string case) or "$?\!- S(q)$" (in the tree case), one for each final state q. To reduce this to a single query, one can add the rules of the form "$S' \mathrel{:-} S(q_I, q)$" or "$S' \mathrel{:-} S(q)$" for all final states q, and use the query "$?\!- S'$".

[41] This can be seen as follows. Suppose that $P \in \Lambda(\Sigma)$ is an almost linear closed λ-term such that $|P|_\beta = /c_1 \dots c_n/ = \lambda z.c_1(\dots(c_n z)\dots)$. Then by Lemma 3.49, $P \twoheadrightarrow_\beta |P|_\beta$ by non-erasing, almost non-duplicating β-reduction. But Lemma 3.60 implies that $\overrightarrow{\mathrm{Con}}(P)$ is some permutation $(c_{j_1}, \dots, c_{j_n})$ of $(c_1, \dots, c_n)$, and $\widehat{P}[z_{j_1}, \dots, z_{j_n}] \twoheadrightarrow_\beta \lambda z.z_1(\dots(z_n z)\dots)$ by non-erasing, almost non-duplicating β-reduction. However, it is easy to see that the set of non-affine pure λ-terms is closed under non-erasing almost non-duplicating β-reduction. Since $\lambda z.z_1(\dots(z_n z)\dots)$ is linear, it follows that $\widehat{P}[z_{j_1}, \dots, z_{j_n}]$, and hence P, must be linear.

[42] The notation here follows that of *elementary formal systems* [72, 4, 28], which are logic programs on strings.

This grammar generates the language $\{ w_n \mid n \geq 1 \}$, where $w_n = \mathtt{1010}^2 \dots \mathtt{10}^n$. The third rule involves copying of the variable x_2. The translation of this grammar into a CFLG looks as follows:

$$\begin{array}{l}
S(\lambda z.X(\lambda x_1 x_2.x_1(x_2 z))) \mathrel{:-} A(X). \\
A(\lambda w.w(\lambda z.\mathtt{1}z)(\lambda z.\mathtt{0}z)). \\
A(\lambda w.X(\lambda x_1 x_2.w(\lambda z.x_1(x_2(\mathtt{1}z)))(\lambda z.x_2(\mathtt{0}z)))) \mathrel{:-} A(X).
\end{array}$$

Here, $f(S) = o \to o$, and $f(A) = ((o \to o) \to (o \to o) \to o) \to o$. This grammar is not almost linear, since the bound variable x_2 in the λ-term on the left-hand side of the third rule must have a non-atomic type in the principal typing of the λ-term.

Here is a grammar that generates a set of trees whose yield image is the language of the above PMCFG:

$$\begin{array}{l}
S(c(x_1, x_2)) \mathrel{:-} A(x_1, x_2). \\
A(\mathtt{1}, \mathtt{0}). \\
A(c(x_1, c(x_2, \mathtt{1})), c(x_2, \mathtt{0})) \mathrel{:-} A(x_1, x_2).
\end{array}$$

Here, c is a symbol of rank 2, and $\mathtt{1}$ and $\mathtt{0}$ are symbols of rank 0. A grammar like this, where a nonterminal denotes a relation on trees and a rule may duplicate trees, may be called a *parallel multiple regular tree grammar*, in analogy with a *multiple regular tree grammar* [60, 23]. For example, the tree

$$c(c(\mathtt{1}, c(\mathtt{0}, \mathtt{1})), c(\mathtt{0}, \mathtt{0}))$$

is generated by the above tree grammar with the following derivation:

$$\begin{array}{c}
S(c(c(\mathtt{1}, c(\mathtt{0}, \mathtt{1})), c(\mathtt{0}, \mathtt{0}))) \\
| \\
A(c(\mathtt{1}, c(\mathtt{0}, \mathtt{1})), c(\mathtt{0}, \mathtt{0})) \\
| \\
A(\mathtt{1}, \mathtt{0})
\end{array}$$

The yield of this tree is $\mathtt{10100} = w_2$.

It is straightforward to encode the above tree grammar into an almost linear CFLG:

$$\begin{array}{l}
S(X(\lambda x_1 x_2.c x_1 x_2)) \mathrel{:-} A(X). \\
A(\lambda w.w\mathtt{10}). \\
A(\lambda w.X(\lambda x_1 x_2.w(c x_1(c x_2 \mathtt{1}))(c x_2 \mathtt{0}))) \mathrel{:-} A(X).
\end{array}$$

Here, $f(S) = o$ and $f(A) = (o \to o \to o) \to o$. This CFLG $\mathscr{G}$ translates into the following Datalog program $\mathbf{P}_{\mathscr{G}}$:

$$S(i_1) :\!-\ c(i_2, i_4, i_3), A(i_1, i_2, i_4, i_3).$$
$$A(i_1, i_1, i_3, i_2) :\!-\ \mathtt{1}(i_2), \mathtt{0}(i_3).$$
$$A(i_1, i_2, i_8, i_3) :\!-\ c(i_3, i_5, i_4), c(i_5, i_7, i_6), \mathtt{1}(i_7), c(i_8, i_9, i_6), \mathtt{0}(i_9), A(i_1, i_2, i_6, i_4).$$

The above Datalog program $\mathbf{P}_{\mathscr{G}}$ can be used to parse input strings with respect to the original PMCFG. For example, if the input string is 10100, we first form a deterministic bottom-up tree automaton $\mathscr{A}$ that recognizes the set of trees over the ranked alphabet $\{\mathtt{1}^{(0)}, \mathtt{0}^{(0)}, c^{(2)}\}$ whose yield is 10100. The states of this automaton are of the form q_w, where w is one of the non-empty substrings of this string:

$$\mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{010}, \mathtt{100}, \mathtt{101}, \mathtt{0100}, \mathtt{1010}, \mathtt{10100}$$

For each of these strings w and nonempty strings u, v such that $w = uv$, the automaton $\mathscr{A}$ has the rule

$$c(q_u(x_1), q_v(x_2)) \to q_w(c(x_1, x_2)).$$

which gives rise to the extensional fact

$$c(q_w, q_v, q_u).$$

Moreover, for each symbol a occurring in w, the automaton has the rule

$$a \to q_a(a)$$

which translates into the extensional fact

$$a(q_a).$$

The database obtained this way is deterministic. In the present case, we get the database D consisting of the following facts (we write w instead of q_w):[43]

$$\mathtt{0}(\mathtt{0}). \quad \mathtt{1}(\mathtt{1}).$$
$$c(\mathtt{00}, \mathtt{0}, \mathtt{0}). \quad c(\mathtt{01}, \mathtt{0}, \mathtt{1}). \quad c(\mathtt{10}, \mathtt{1}, \mathtt{0}).$$

[43]If the PMCFG rules contain occurrences of the empty string ϵ, then the corresponding PMRTG will have a special rank 0 symbol corresponding to ϵ, and one needs to take *all* substrings of the input string, not just non-empty ones, in the construction of the automaton $\mathscr{A}$. The automaton will then represent the *syntactic monoid* of the singleton set consisting of the input string.

$$c(\mathtt{010},\mathtt{0},\mathtt{10}).\quad c(\mathtt{010},\mathtt{01},\mathtt{0}).$$
$$c(\mathtt{100},\mathtt{1},\mathtt{00}).\quad c(\mathtt{100},\mathtt{10},\mathtt{0}).$$
$$c(\mathtt{101},\mathtt{1},\mathtt{01}).\quad c(\mathtt{101},\mathtt{10},\mathtt{1}).$$
$$c(\mathtt{0100},\mathtt{0},\mathtt{100}).\quad c(\mathtt{0100},\mathtt{01},\mathtt{00}).\quad c(\mathtt{0100},\mathtt{010},\mathtt{0}).$$
$$c(\mathtt{1010},\mathtt{1},\mathtt{010}).\quad c(\mathtt{1010},\mathtt{10},\mathtt{10}).\quad c(\mathtt{1010},\mathtt{101},\mathtt{0}).$$
$$c(\mathtt{10100},\mathtt{1},\mathtt{0100}).\quad c(\mathtt{10100},\mathtt{10},\mathtt{100}).\quad c(\mathtt{10100},\mathtt{101},\mathtt{00}).\quad c(\mathtt{10100},\mathtt{1010},\mathtt{0}).$$

By Theorem 4.3,

$$\mathbf{P}_{\mathscr{G}} \cup D \vdash S(\mathtt{10100}) \tag{62}$$

if and only if $\mathscr{G}$ generates (the λ-term representation of) a tree whose yield is `10100`. This is so if and only if the original PMCFG generates this string. Since the rules of the PMCFG are in one-one correspondence with the rules of $\mathscr{G}$, parsing the string with this PMCFG reduces to the problem of computing all derivation trees for (62), in the form of a shared forest.

This reduction generally applies to the yield images of the tree languages that can be generated by almost linear CFLGs. It is shown in unpublished work [44] that the class of tree languages generated by almost linear CFLGs coincides with the class of output languages of *tree-valued attribute grammars* or *attributed tree transducers* (see [11]). As a consequence, the class of yield images of these tree languages is simply the class of output languages of *string-valued attribute grammars*, studied by Engelfriet [22].

Clearly, the deterministic bottom-up tree automaton $\mathscr{A}$ (and the corresponding database) associated with the input string can be constructed in logarithmic space. Note that all trees accepted by $\mathscr{A}$ have the same number of constants, namely $2n-1$ for input string of length n.[44] This implies that recognition and parsing with these grammars are in (functional) LOGCFL, matching the result of Engelfriet [22].[45]

4.2.4 An application to generation from underspecified semantics

Koller et al. [49] have proposed to use a regular tree grammar as an underspecified representation of various readings of sentences with multiple scope-taking operators. However, when the operators include variable-binders, a tree is not ideally suited to represent the scope relation because one needs to associate a variable name to each

[44]This number assumes that $\mathscr{A}$ does not have a special symbol representing the empty string.

[45]Note that parsing as intersection with these grammars, where the input is a regular set of strings, can also be represented as Datalog query evaluation. The deterministic bottom-up tree automaton that determines the database and query can be obtained from the syntactic monoid of the input regular set.

$\mathsf{S}(\lambda z.Y(\lambda y_1y_2.y_1(y_2z)),\ X)$:– $\mathsf{NP\_VP}(Y,X)$.
$\mathsf{NP\_VP}(\lambda w.Y(\lambda y_1y_2.w(\lambda z.y_1z)(\lambda z.\mathsf{didn't}(y_2z))),\ \neg X)$:– $\mathsf{NP\_VP}(Y,X)$.
$\mathsf{NP\_VP}(\lambda w.w(\lambda z.Y_1z)(\lambda z.Y_2z),\ X_1(\lambda x.X_2x))$:– $\mathsf{NP}(Y_1,X_1), \mathsf{VP}(Y_2,X_2)$.
$\mathsf{NP\_VP}(\lambda w.Y_1(\lambda y_1y_2.w(\lambda z.y_1z)(\lambda z.y_2(Y_2z))),\ X_2(\lambda x.X_1x))$:– $\mathsf{NP\_V}(Y_1,X_1), \mathsf{NP}(Y_2,X_2)$.
$\mathsf{NP\_V}(\lambda w.Y(\lambda y_1y_2.w(\lambda z.y_1z)(\lambda z.\mathsf{didn't}(y_2z))),\ \lambda x.\neg(Xx))$:– $\mathsf{NP\_V}(Y,X)$.
$\mathsf{NP\_V}(\lambda w.w(\lambda z.Y_1z)(\lambda z.Y_2z),\ \lambda y.X_1(\lambda x.X_2yx))$:– $\mathsf{NP}(Y_1,X_1), \mathsf{V}(Y_2,X_2)$.
$\mathsf{VP}(\lambda z.\mathsf{didn't}(Yz),\ \lambda x.\neg(Xx))$:– $\mathsf{VP}(Y,X)$.
$\mathsf{VP}(\lambda z.Y_1(Y_2z),\ \lambda x.X_2(\lambda y.X_1yx))$:– $\mathsf{V}(Y_1,X_1), \mathsf{NP}(Y_2,X_2)$.
$\mathsf{NP}(\lambda z.Y_1(Y_2z),\ \lambda v.X_1(\lambda x.X_2x)(\lambda x.vx))$:– $\mathsf{Det}(Y_1,X_1), \mathsf{N}(Y_2,X_2)$.
$\mathsf{Det}(/\mathsf{a}/,\ \lambda uv.\exists(\lambda x.\wedge(ux)(vx)))$.
$\mathsf{Det}(/\mathsf{every}/,\ \lambda uv.\forall(\lambda x.\rightarrow(ux)(vx)))$.
$\mathsf{Det}(/\mathsf{no}/,\ \lambda uv.\forall(\lambda x.\rightarrow(ux)(\neg(vx))))$.
$\mathsf{Det}(/\mathsf{not\ every}/,\ \lambda uv.\neg(\forall(\lambda x.\rightarrow(ux)(vx))))$.
$\mathsf{N}(/\mathsf{book}/,\ \lambda x.\mathbf{book}\,x)$.
$\mathsf{N}(/\mathsf{student}/,\ \lambda x.\mathbf{student}\,x)$.
$\mathsf{V}(/\mathsf{read}/,\ \lambda yx.\mathbf{read}\,y\,x)$.

Figure 13: A synchronous CFLG.

occurrence of a binder to represent the binding relation. These variable names must be chosen in such a way as to avoid clashes of variables, and some mechanism is needed to identify α-equivalent representations (i.e., representations that differ only in renaming of bound variables).

A compact representation of a set of λ-terms, rather than trees, will improve upon Koller et al.'s [49] approach. We can use a deterministic database D over a database schema $\mathcal{D}_{\Sigma,U}$ associated with a higher-order signature Σ as a representation of a set of λ-terms over Σ. If the syntax-semantics is given as a "synchronous" CFLG whose semantics side is an almost linear CFLG $\mathscr{G}$, then Theorem 4.3 tells us that D can serve as an "underspecified" input to surface realization.

For example, the synchronous CFLG in Figure 13 generates $\mathsf{every\ student\ didn't\ read\ a\ book}$ with six possible readings:

$$\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\neg(\exists(\lambda y.\wedge(\mathbf{book}\,y)(\mathbf{read}\,y\,x)))))$$
$$\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\exists(\lambda y.\wedge(\mathbf{book}\,y)(\neg(\mathbf{read}\,y\,x)))))$$
$$\neg(\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\exists(\lambda y.\wedge(\mathbf{book}\,y)(\mathbf{read}\,y\,x)))))$$
$$\neg(\exists(\lambda y.\wedge(\mathbf{book}\,y)(\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\mathbf{read}\,y\,x)))))$$
$$\exists(\lambda y.\wedge(\mathbf{book}\,y)(\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\neg(\mathbf{read}\,y\,x)))))$$

$$\exists(\lambda y.\wedge(\mathbf{book}\,y)(\neg(\forall(\lambda x.\rightarrow(\mathbf{student}\,x)(\mathbf{read}\,y\,x)))))$$

The set of these λ-terms can be represented by the following database:

$$\begin{gathered}
\mathbf{student}(\mathbf{s},x). \quad \mathbf{book}(\mathbf{b},y). \quad \mathbf{read}(\mathbf{r},x,y). \\
\neg(\neg,\mathbf{r}). \quad \neg(\exists\neg,\exists). \quad \neg(\forall\neg,\forall). \quad \neg(\forall\exists\neg,\forall\exists). \\
\wedge(\wedge,\mathbf{r},\mathbf{b}). \quad \wedge(\wedge\neg,\neg,\mathbf{b}). \quad \wedge(\wedge\forall,\forall,\mathbf{b}). \quad \wedge(\wedge\forall\neg,\forall\neg,\mathbf{b}). \\
\exists(\exists,\wedge,y). \quad \exists(\exists\neg,\wedge\neg,y). \quad \exists(\forall\exists,\wedge\forall,y). \quad \exists(\forall\exists\neg,\wedge\exists\neg,y). \\
\rightarrow(\rightarrow,\mathbf{r},\mathbf{s}). \quad \rightarrow(\rightarrow\neg,\neg,\mathbf{s}). \quad \rightarrow(\rightarrow\exists,\exists,\mathbf{s}). \quad \rightarrow(\rightarrow\exists\neg,\exists\neg,\mathbf{s}). \\
\forall(\forall,\rightarrow,x). \quad \forall(\forall\neg,\rightarrow\neg,x). \quad \forall(\forall\exists,\rightarrow\exists,x). \quad \forall(\forall\exists\neg,\rightarrow\exists\neg,x).
\end{gathered}$$

In this database (call it D), we use mnemonic names like $\forall\exists\neg$, instead of integers, as database constants. For instance, λ-terms in $\Lambda(D,\forall\exists)$ contain $\forall$ and $\exists$, but not $\neg$. A database like this can be thought of as a hypergraph that can be obtained from the disjoint union of the hypergraphs corresponding to the above six almost linear λ-terms by identifying certain nodes and hyperedges. It is easy to check that this database is deterministic; it can then be used together with the Datalog program associated with the semantic side of the synchronous grammar in Figure 13 to obtain a shared parse forest of all derivation trees of sentences that have at least one reading in common with the sentence every student didn't read a book—namely, no student read a book, not every student read a book, and the same sentence itself. This procedure is more efficient than the brute-force method, where each reading of the sentence is input to a surface realization routine in turn.[46]

4.3 Magic sets and Earley-style algorithms

The *magic-sets* rewriting of a Datalog program allows bottom-up evaluation to avoid deriving useless facts by mimicking top-down evaluation of the original program. The result of the *generalized supplementary magic-sets* rewriting of Beeri and Ramakrishnan [8] applied to the Datalog program representing a CFG essentially coincides with the *deduction system* [69] or *uninstantiated parsing system* [70] for Earley parsing [20]. By applying the same rewriting method to Datalog programs representing almost linear CFLGs, we can obtain efficient parsing and generation algorithms for various grammar formalisms with context-free derivations.

[46]There is the question of how a deterministic database representing the range of possible readings of a sentence can be found, if one exists. In the case at hand, there is a way of constructing the desired database from the shared parse forest of the sentence by duplicating certain nodes (namely, the NP nodes and the Det nodes). However, it is easy to see that no such deterministic database may exist in general. It is an open question when and how a desired database can be constructed efficiently.

We illustrate this approach with the program in (6), repeated below, following the presentation of Ullman [77, 78]. We assume the query to take the form "?$-$ $S(0, x)$.", so that the input database can be processed incrementally.

$$\begin{array}{l} S(i_1, i_3) \mathrel{:-} A(i_1, i_3, i_2, i_2). \\ A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}(i_1, i_2), \mathtt{b}(i_3, i_4), \mathtt{c}(i_5, i_6), \mathtt{d}(i_7, i_8), A(i_2, i_7, i_3, i_6). \\ A(i_1, i_2, i_1, i_2). \end{array} \tag{6}$$

The program is first made *safe* by eliminating the rule with empty right-hand side:

$$\begin{array}{l} S(i_1, i_3) \mathrel{:-} A(i_1, i_3, i_2, i_2). \\ A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}(i_1, i_2), \mathtt{b}(i_3, i_4), \mathtt{c}(i_5, i_6), \mathtt{d}(i_7, i_8), A(i_2, i_7, i_3, i_6). \\ A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}(i_1, i_2), \mathtt{b}(i_2, i_4), \mathtt{c}(i_5, i_6), \mathtt{d}(i_6, i_8). \end{array}$$

The *subgoal rectification* removes duplicate arguments from subgoals, creating new predicates as needed:

$$\begin{array}{l} S(i_1, i_3) \mathrel{:-} B(i_1, i_3, i_2). \\ A(i_1, i_8, i_4, i_5) \mathrel{:-}, \mathtt{a}(i_1, i_2), \mathtt{b}(i_3, i_4), \mathtt{c}(i_5, i_6), \mathtt{d}(i_7, i_8), A(i_2, i_7, i_3, i_6). \\ A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}(i_1, i_2), \mathtt{b}(i_2, i_4), \mathtt{c}(i_5, i_6), \mathtt{d}(i_6, i_8). \\ B(i_1, i_8, i_4) \mathrel{:-}, \mathtt{a}(i_1, i_2), \mathtt{b}(i_3, i_4), \mathtt{c}(i_4, i_6), \mathtt{d}(i_7, i_8), A(i_2, i_7, i_3, i_6). \\ B(i_1, i_8, i_4) \mathrel{:-} \mathtt{a}(i_1, i_2), \mathtt{b}(i_2, i_4), \mathtt{c}(i_4, i_6), \mathtt{d}(i_6, i_8). \end{array}$$

We then attach to predicates *adornments* indicating the free/bound status of arguments in top-down evaluation, reordering subgoals so that as many arguments as possible are marked as bound:

$$\begin{array}{l} S^{bf}(i_1, i_3) \mathrel{:-} B^{bff}(i_1, i_3, i_2). \\ B^{bff}(i_1, i_8, i_4) \mathrel{:-} \mathtt{a}^{bf}(i_1, i_2), A^{bfff}(i_2, i_7, i_3, i_6), \mathtt{b}^{bf}(i_3, i_4), \mathtt{c}^{bb}(i_4, i_6), \\ \qquad\qquad\qquad \mathtt{d}^{bf}(i_7, i_8). \\ B^{bff}(i_1, i_8, i_4) \mathrel{:-} \mathtt{a}^{bf}(i_1, i_2), \mathtt{b}^{bf}(i_2, i_4), \mathtt{c}^{bf}(i_4, i_6), \mathtt{d}^{bf}(i_6, i_8). \\ A^{bfff}(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}^{bf}(i_1, i_2), A^{bfff}(i_2, i_7, i_3, i_6), \mathtt{b}^{bf}(i_3, i_4), \mathtt{c}^{bb}(i_5, i_6), \\ \qquad\qquad\qquad\qquad \mathtt{d}^{bf}(i_7, i_8). \\ A^{bfff}(i_1, i_8, i_4, i_5) \mathrel{:-} \mathtt{a}^{bf}(i_1, i_2), \mathtt{b}^{bf}(i_2, i_4), \mathtt{c}^{ff}(i_5, i_6), \mathtt{d}^{bf}(i_6, i_8). \end{array}$$

The generalized supplementary magic-sets rewriting finally gives the following rule set:

$$\begin{array}{l} r_1\colon m\_B(i_1) \mathrel{:-} m\_S(i_1). \\ r_2\colon S(i_1, i_3) \mathrel{:-} m\_B(i_1), B(i_1, i_3, i_2). \\ r_3\colon sup_{2.1}(i_1, i_2) \mathrel{:-} m\_B(i_1), \mathtt{a}(i_1, i_2). \end{array}$$

$$
\begin{aligned}
r_4\colon\ & \mathit{sup}_{2.2}(i_1, i_7, i_3, i_6) \mathrel{:-} \mathit{sup}_{2.1}(i_1, i_2), A(i_2, i_7, i_3, i_6).\\
r_5\colon\ & \mathit{sup}_{2.3}(i_1, i_7, i_6, i_4) \mathrel{:-} \mathit{sup}_{2.2}(i_1, i_7, i_3, i_6), \mathtt{b}(i_3, i_4).\\
r_6\colon\ & \mathit{sup}_{2.4}(i_1, i_7, i_4) \mathrel{:-} \mathit{sup}_{2.3}(i_1, i_7, i_6, i_4), \mathtt{c}(i_4, i_6).\\
r_7\colon\ & B(i_1, i_8, i_4) \mathrel{:-} \mathit{sup}_{2.4}(i_1, i_7, i_4), \mathtt{d}(i_7, i_8).\\
r_8\colon\ & \mathit{sup}_{3.1}(i_1, i_2) \mathrel{:-} m\_B(i_1), \mathtt{a}(i_1, i_2).\\
r_9\colon\ & \mathit{sup}_{3.2}(i_1, i_4) \mathrel{:-} \mathit{sup}_{3.1}(i_1, i_2), \mathtt{b}(i_2, i_4).\\
r_{10}\colon\ & \mathit{sup}_{3.3}(i_1, i_4, i_6) \mathrel{:-} \mathit{sup}_{3.2}(i_1, i_4), \mathtt{c}(i_4, i_6).\\
r_{11}\colon\ & B(i_1, i_8, i_4) \mathrel{:-} \mathit{sup}_{3.3}(i_1, i_4, i_6), \mathtt{d}(i_6, i_8).\\
r_{12}\colon\ & m\_A(i_2) \mathrel{:-} \mathit{sup}_{2.1}(i_1, i_2).\\
r_{13}\colon\ & m\_A(i_2) \mathrel{:-} \mathit{sup}_{4.1}(i_1, i_2).\\
r_{14}\colon\ & \mathit{sup}_{4.1}(i_1, i_2) \mathrel{:-} m\_A(i_1), \mathtt{a}(i_1, i_2).\\
r_{15}\colon\ & \mathit{sup}_{4.2}(i_1, i_7, i_3, i_6) \mathrel{:-} \mathit{sup}_{4.1}(i_1, i_2), A(i_2, i_7, i_3, i_6).\\
r_{16}\colon\ & \mathit{sup}_{4.3}(i_1, i_7, i_6, i_4) \mathrel{:-} \mathit{sup}_{4.2}(i_1, i_7, i_3, i_6), \mathtt{b}(i_3, i_4).\\
r_{17}\colon\ & \mathit{sup}_{4.4}(i_1, i_7, i_4, i_5) \mathrel{:-} \mathit{sup}_{4.3}(i_1, i_7, i_6, i_4), \mathtt{c}(i_5, i_6).\\
r_{18}\colon\ & A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathit{sup}_{4.4}(i_1, i_7, i_4, i_5), \mathtt{d}(i_7, i_8).\\
r_{19}\colon\ & \mathit{sup}_{5.1}(i_1, i_2) \mathrel{:-} m\_A(i_1), \mathtt{a}(i_1, i_2).\\
r_{20}\colon\ & \mathit{sup}_{5.2}(i_1, i_4) \mathrel{:-} \mathit{sup}_{5.1}(i_1, i_2), \mathtt{b}(i_2, i_4).\\
r_{21}\colon\ & \mathit{sup}_{5.3}(i_1, i_4, i_5, i_6) \mathrel{:-} \mathit{sup}_{5.2}(i_1, i_4), \mathtt{c}(i_5, i_6).\\
r_{22}\colon\ & A(i_1, i_8, i_4, i_5) \mathrel{:-} \mathit{sup}_{5.3}(i_1, i_4, i_5, i_6), \mathtt{d}(i_6, i_8).
\end{aligned}
$$

The following is a version of the seminaive bottom-up evaluation algorithm expressed in the form of chart parsing:

1. (init) Initialize the chart to the empty set, the agenda to the singleton $\{m\_S(0)\}$, and n to 0.

2. Repeat the following steps:

   (a) Repeat the following steps until the agenda is exhausted:

      i. Remove a fact from the agenda, called the *trigger*.
      ii. Add the trigger to the chart.
      iii. Generate all facts that are immediate consequences of the trigger together with all facts in the chart, and add to the agenda those generated facts that are neither already in the chart nor in the agenda.

(b) (SCAN) Remove the next fact from the input database and add it to the agenda, incrementing n. If there is no more fact in the input database, go to step 3.

3. If $S(0, n)$ is in the chart, accept; otherwise reject.

The following is the trace of the algorithm on input string `aabbccdd`; the derived facts are recorded in the order they enter the agenda:

1.	$m\_S(0)$	INIT	14.	$\mathtt{c}(4,5)$	SCAN
2.	$m\_B(0)$	$r_1, 1$	15.	$sup_{5.3}(1,3,4,5)$	$r_{21}, 12, 14$
3.	$\mathtt{a}(0,1)$	SCAN	16.	$\mathtt{c}(5,6)$	SCAN
4.	$sup_{2.1}(0,1)$	$r_3, 2, 3$	17.	$sup_{5.3}(1,3,5,6)$	$r_{21}, 12, 16$
5.	$sup_{3.1}(0,1)$	$r_8, 2, 3$	18.	$\mathtt{d}(6,7)$	SCAN
6.	$m\_A(1)$	$r_{12}, 4$	19.	$A(1,7,3,5)$	$r_{22}, 17, 18$
7.	$\mathtt{a}(1,2)$	SCAN	20.	$sup_{2.2}(0,7,3,5)$	$r_4, 4, 19$
8.	$sup_{4.1}(1,2)$	$r_{14}, 6, 7$	21.	$sup_{2.3}(0,7,5,4)$	$r_5, 20, 13$
9.	$sup_{5.1}(1,2)$	$r_{19}, 6, 7$	22.	$sup_{2.4}(0,7,4)$	$r_6, 21, 14$
10.	$m\_A(2)$	$r_{13}, 8$	23.	$\mathtt{d}(7,8)$	SCAN
11.	$\mathtt{b}(2,3)$	SCAN	24.	$B(0,8,4)$	$r_7, 22, 23$
12.	$sup_{5.2}(1,3)$	$r_{20}, 9, 11$	25.	$S(0,8)$	$r_2, 2, 24$
13.	$\mathtt{b}(3,4)$	SCAN			

Note that unlike previous Earley-style parsing algorithms for TAGs, the present algorithm is an instantiation of a general schema that applies to parsing with more powerful grammar formalisms as well as to generation with Montague semantics.[47]

5 Conclusion

This paper has shown that recognition and parsing for a wide range of grammars with "context-free" derivations, as well as surface realization (tactical generation) for those grammars coupled with a certain restricted kind of Montague semantics, all reduce to Datalog query evaluation and hence allow highly efficient algorithms. The method of reduction is uniform for both recognition/parsing and surface realization, and the complexity upper bound that has been established, namely, LOGCFL, is

[47]The above Earley-style recognition algorithm for tree-adjoining languages does not have the *correct prefix property*, a desirable feature for Earley-style algorithms for string grammars. See [43] for how to supplement magic-sets rewriting with another simple rewriting to achieve the correct prefix property.

tight. By regarding the problem of surface realization as the problem of recognition/parsing of languages of λ-terms, this paper has demonstrated that it is possible to study surface realization abstractly in the style of formal language theory, just like parsing. I hope that the methods employed here help pave the way for eliminating much of the ad hoc methodology that is so common in computational linguistics.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

[2] Takahito Aoto. Uniqueness of normal proofs in implicational intuitionistic logic. *Journal of Logic, Language and Information*, 8:217–242, 1999.

[3] Takahito Aoto and Hiroakira Ono. Uniqueness of normal proofs in $\{\to, \wedge\}$-fragment of NJ. Research Report IS-RR-94-0024F, School of Information Science, Japan Advanced Institute of Science and Technology, 1994.

[4] Setsuo Arikawa, Takeshi Shinohara, and Akihiro Yamamoto. Learning elementary formal systems. *Theoretical Computer Science*, 95(1):97–113, 1992.

[5] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14(2):143–172, 1961.

[6] Hendrik Pieter Barendregt. *The Lambda Calculus.* North-Holland, Amsterdam, 1984. Revised Edition.

[7] Jon Barwise and Robin Cooper. Generalized quantifiers and natural language. *Linguistics and Philosophy*, 4:159–219, 1981.

[8] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–299, 1991.

[9] N.D. Belnap. The two-property. *Relevance Logic Newsletter*, 1:173–180, 1976.

[10] Inge Bethke, Jan Willem Klop, and Roel de Vrijer. Descendants and origins in term rewriting. *Information and Computation*, 159:59–124, 2000.

[11] Roderick Bloem and Joost Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *Journal of Computer and System Sciences*, 61:1–50, 2000.

[12] Samuel R. Buss. The undecidability of k-provability. *Annals of Pure and Applied Logic*, 53(1):75–102, 1991.

[13] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.

[14] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[15] Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. Minimal recursion semantics: An introduction. *Research on Language and Computation*, 3:281–332, 2005.

[16] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374–425, 2001.

[17] Philippe de Groote. Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155, 2001.

[18] Philippe de Groote. Tree-adjoining grammars as abstract categorial grammars. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*, pages 145–150. Universitá di Venezia, 2002.

[19] Philippe de Groote and Sylvain Pogodalla. On the expressive power of abstract categorial grammars: Representing context-free formalisms. *Journal of Logic, Language and Information*, 13:421–438, 2004.

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

[21] J. Engelfriet and E. M. Schmidt. IO and OI, part I. *The Journal of Computer and System Sciences*, 15:328–353, 1977.

[22] Joost Engelfriet. The complexity of languages generated by attribute grammars. *SIAM Journal on Computing*, 15:70–86, 1986.

[23] Joost Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Langauges, Volume 3: Beyond Words*, pages 125–213. Springer, Berlin, 1997.

[24] Michael J. Fischer. *Grammars with Macro-Like Productions.* PhD thesis, Harvard University, 1968.

[25] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, Mass., 1985.

[26] Georg Gottlob, Nicola Lenoe, and Francesco Scarcello. Computing LOGCFL certificates. *Theoretical Computer Science*, 270:761–777, 2002.

[27] Sheila A. Greibach. The hardest context-free language. *SIAM Journal on Computing*, 2:304–310, 1973.

[28] Annius Groenink. *Surface without Structures.* PhD thesis, Utrech University, 1997.

[29] Leon Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[30] Gerd G. Hillebrand, Paris C. Kanellakis, and Harry G. Mairson. Database query languages embedded in the typed lambda calculus. *Information and Computation*, 127:117–144, 1996.

[31] J. Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, Cambridge, 1997.

[32] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, Cambridge, 2008.

[33] Sachio Hirokawa. Balanced formulas, BCK-minimal formulas and their proofs. In Anil Nerode and Mikhail Taitslin, editors, *Logical Foundations of Computer Science — Tver '92*, pages 198–208, Berlin, 1992. Springer.

[34] Gérard Huet. *Résolution d'équation dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, 1976.

[35] David S. Johnson. A catalog of complexity classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 67–161. Elsevier, Amsterdam, 1990.

[36] Neil D. Jones and William T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3:105–117, 1977.

[37] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegoz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69–123. Springer, Berlin, 1997.

[38] Yuichi Kaji, Ryuichi Nakanishi, Hiroyuki Seki, and Tadao Kasami. The universal recognition problems for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems*, E 75–D(1):78–88, 1992.

[39] Yuichi Kaji, Ryuichi Nakanishi, Hiroyuki Seki, and Tadao Kasami. The universal recognition problems for parallel multiple context-free grammars and their subclasses. *IEICE Transactions on Informaiton and Systems*, E 75–D(4):499–508, 1992.

[40] Makoto Kanazawa. Abstract families of abstract categorial languages. *Electronic Notes in Theoretical Computer Science*, 165:65–80, 2006. Proceedings of the 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006), Logic, Language, Information and Computation 2006.

[41] Makoto Kanazawa. Computing interpolants in implicational logics. *Annals of Pure and Applied Logic*, 142(1–3):125–201, 2006.

[42] Makoto Kanazawa. Parsing and generation as Datalog queries. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*, Prague, Czech Republic, 2007.

[43] Makoto Kanazawa. A prefix-correct Earley recognizer for multiple context-free grammars. In *Proceedings of the Ninth International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+ 9)*, pages 49–56, 2008.

[44] Makoto Kanazawa. A lambda calculus characterization of MSO definable tree transductions (abstract). *Bulletin of Symbolic Logic*, 15(2):250–251, 2009.

[45] Makoto Kanazawa. Second-order abstract categorial grammars as hyperedge replacement grammars. *Journal of Logic, Language and Information*, 19(2):37–161, 2010.

[46] Makoto Kanazawa. Almost affine lambda terms. In Andrzej Indrzejczak, Janusz Kaczmarek, and Michaał Zawidzki, editors, *Trends in Logic XIII*, pages 131–148, Łódź, 2014. Łódź University Press.

[47] Makoto Kanazawa and Ryo Yoshinaka. Lexicalization of second-order ACGs. NII Technical Report NII-2005-012E, National Institute of Informatics, Tokyo, 2005.

[48] Paris C. Kanellakis. Logic programming and parallel complexity. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 547–585. Morgan Kaufmann, Los Altos, CA, 1988.

[49] Alexander Koller, Michaela Regneri, and Stafan Thater. Regular tree grammars as a formalism for scope underspecification. In *46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Proccedings of the Conference*, pages 218–226, Columbus, Ohio, 2008.

[50] Alexander Koller and Kristina Striegnitz. Generation as dependency parsing. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 17–24, Philadelphia, 2002.

[51] Eric Kow. *Surface Realization: Ambiguity and Determinism.* PhD thesis, l'université Henri Poincaré, 2007.

[52] Bernard Lang. Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494, 1994.

[53] J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Springer, Berlin, 2003.

[54] Ralph Loader. Notes on simply typed lambda calculus. Technical Report ECS-LFCS-98-381, Laboratory for Foundations of Computer Science, School of Informatics, The University of Edinburgh, Edinburgh, 1998.

[55] Markus Lohrey. On the parallel complexity of tree automata. In A. Middeldorp, editor, *RTA 2001*, volume 2051 of *Lecture Notes in Computer Science*, pages 201–215, Berlin, 2001. Springer.

[56] Grigori Mints. *A Short Introduction to Intuitionistic Logic.* Kluwer Academic/Plenum Publishers, New York, 2000.

[57] Richard Montague. The proper treatment of quantification in ordinary English. In J. Hintikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics.* Reidel, Dordrecht, 1973.

[58] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[59] Detlef Plump. Term graph rewriting. In Hartmut Ehrig, G. Engels, H.-J. Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 2, pages 3–61. World Scientific, Singapore, 1999.

[60] Jean-Calude Raoult. Rational tree relations. *Bulletin of the Belgian Mathematical Society*, 4:149–176, 1997.

[61] Walter L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[62] Sylvain Salvati. *Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites.* PhD thesis, l'Institut National Polytechnique de Lorraine, 2005.

[63] Sylvain Salvati. Encoding second order string ACG with deterministic tree walking transducers. In Shuly Wintner, editor, *Proceedings of FG 2006: The 11th conference on Formal Grammar*, FG Online Proceedings, pages 143–156, Stanford, CA, 2007. CSLI Publications.

[64] Sylvain Salvati. Recognizability in the simply typed lambda-calculus. In Hiroakira

Ono, Makoto Kanazawa, and Ruy de Queiroz, editors, *WoLLIC 2009*, volume 5514 of *Lecture Notes in Artificial Intelligence*, pages 48–60, Berlin, 2009. Springer.

[65] Sylvain Salvati. On the membership problem for non-linear abstract categorial grammars. *Journal of Logic, Language and Information*, 19:163–183, 2010.

[66] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

[67] Ehud Y. Shapiro. Alternation and the computational complexity of logic programs. *Journal of Logic Programming*, 1:19–33, 1984.

[68] Stuart Shieber. The problem of logical-form equivalence. *Computational Linguistics*, 19:179–190, 1993.

[69] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementations of deductive parsing. *Journal of Logic Programming*, 24:3–36, 1995.

[70] Klaas Sikkel. *Parsing Schemata.* Springer, Berlin, 1997.

[71] Klaas Sikkel. Parsing schemeta and correctness of parsing algorithms. *Theoretical Computer Science*, 199(1–2):87–103, 1998.

[72] Raymond M. Smullyan. *Theory of Formal Systems.* Princeton University Press, Princeton, N.J., 1961.

[73] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* Elsevier, Amsterdam, 2006.

[74] Rick Statman. On the complexity of alpha conversion. *Journal of Symbolic Logic*, 72(4):1197–2003, 2007.

[75] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, Cambridge, second edition edition, 2000.

[76] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville, MD, 1988.

[77] Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, 1989.

[78] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The New Technologies. Computer Science Press, Rockville, MD, 1989.

[79] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[80] David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* PhD thesis, University of Pennsylvania, 1988.

Received 23 December 2015