Algorithmic Specifications in Linear Logic with Subexponentials

Vivek Nigam Dale Miller

INRIA Saclay - Île-de France and LIX/École Polytechnique
Route de Saclay, 91128 PALAISEAU Cedex FRANCE
nigam at lix.polytechnique.fr, dale.miller at inria.fr

Abstract

The linear logic exponentials !, ? are not canonical: one can add to linear logic other such operators, say $!^l$, $?^l$, which may or may not allow contraction and weakening, and where l is from some pre-ordered set of labels. We shall call these additional operators *subexponentials* and use them to assign *locations* to multisets of formulas within a linear logic programming setting. Treating locations as subexponentials greatly increases the algorithmic expressiveness of logic. To illustrate this new expressiveness, we show that focused proof search can be precisely linked to a simple algorithmic specification language that contains while-loops, conditionals, and insertion into and deletion from multisets. We also give some general conditions for when a focused proof step can be executed in constant time. In addition, we propose a new logical connective that allows for the creation of new subexponentials, thereby further augmenting the algorithmic expressiveness of logic.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Computational Logic

General Terms Algorithms, Design, Theory

Keywords proof search, linear logic, subexponentials

1. Introduction

Computation in the *proof-search* paradigm (a.k.a. logic programming) can be characterized as the process of searching for a cut-free sequent proof. The expressiveness of logic programming can be judged, in part, by examining the kind of changes that can take place within sequents during the search for a proof. Let Ξ be a cut-free proof of $\Gamma \vdash \Delta$ and let $\Gamma' \vdash \Delta'$ be a sequent occurring in Ξ. The dynamics of proof search in this setting can be partially judged by examining the possible differences that can occur between Γ and Γ' and between Δ and Δ'.

When proof search is conducted within intuitionistic logic, Γ is usually treated as a set of formulas and Δ as a single formula. If we restrict further to Horn clauses, we find that $\Gamma = \Gamma'$ and that Δ and Δ' are atomic formulas. Thus, the only real dynamics during proof search with Horn clauses is that atomic goals change as we move upward through a proof. As a result, all data structures and their various relationships must be encoded as terms within atomic formulas: that is, all the dynamics of computation is buried within *non-logical contexts* (within the scope of predicates). If one uses hereditary Harrop formulas instead of Horn clauses, slightly richer changes are possible: in particular, $\Gamma \subseteq \Gamma'$. When proof search is conducted within linear logic, both Γ and Δ can be treated as multisets and the logic program is free to specify arbitrary, computable relationships between Γ and Γ' and between Δ and Δ'. In linear logic, some data structures and their relationships can be encoded directly in the *logical contexts* of proofs.

Many data structures can be encoded, of course, naturally as sets or multisets of atomic formulas: for example, a graph given by a set of nodes N and an adjacency relation A can be encoded as the multiset of atomic formulas

$$\{node\ x \mid x \in N\} \cup \{adj\ x\ y \mid \langle x, y\rangle \in A\},$$

where $node$ and adj are predicates. A major obstacle to describing algorithms using linear logic programs is that logic does not provide enough test on contexts. While it is possible in linear logic to detect that the global multiset context is empty, it is not possible to perform this test on less than the entire context. Given the multiset encoding of graphs above, linear logic provides a simple mechanism to detect that both the set of nodes and the adjacency information are empty but the logic does not provide a means to check emptiness of just N or just A.

The exponentials of linear logic are not canonical: it is possible to fill the gap between the "linear" modality ("use exactly once") and the ? modality ("use arbitrarily") with a pre-ordered set of exponential-like operators. These intermediate modalities may or may not permit weakening and contraction. We use the term *subexponential* for such modalities

PPDP' 09 September 7 – 9, 2009, Coimbra, Portugal

since the equivalence $?(A \oplus B) \equiv (?A \parr ?B)$—which relates the exponential ?, the additive $\oplus$, and the multiplicative $\parr$—fails when ? does not admit contraction and weakening.

Subexponentials can be used to "locate" data and the promotion rule can be used to test selected locations for emptiness. These subexponentials provide linear logic specifications with enough checks on data to allow for a range of algorithms to be emulated *exactly* via (focused) proof search. We shall illustrate this claim by specifying a simple programming language, called BAG, containing loop instructions, conditionals, and operations that insert into and delete from a multiset, which is powerful enough to specify algorithms such as *Dijkstra's algorithm* for finding the shortest distances in a positively weighted graph. We show that for any BAG program there is a one-to-one correspondence between the set of its (partial) computations and the set of (open) focused proofs of its logic interpretation.

Since there is an exact correspondence between synthetic connectives in the logic and steps in the algorithmic language, we can vary the operational semantics of the algorithmic language by varying certain focusing-related features of the logic. In particular, by either inserting or removing *delay operators* into a logic specification, we can *package* more or fewer *operations* inside a synthetic connective. For example, reading two items from a multiset can be described as two algorithmic steps or as one algorithmic step.

In order to turn a logic specification into an algorithm, one must usually adopt an *interpreter* for the logic and then understand the algorithmic nature of that interpreter. Top-down, depth-first interpreters are traditionally used to describe the algorithmic content of, for example, Horn clauses as Prolog programs. Other algorithmic rendering of logic clauses use bottom-up interpreters [8]. In this paper, we shall not use any explicit interpreter for this role: instead, recent advances in proof theory will be used to organize proof search in algorithmically explicit but still fully declarative ways. In particular, we describe proof systems for which possibly large sets of connectives can be treated as a single, monolithic, synthetic connective and where, in many situations, the inference rules related to such synthetic connectives can be applied in constant time. As a result, the particular nature of whichever interpreter one eventually uses for finding proofs can be largely eliminated when attempting to understand the algorithmic content of a logic specification.

After motivating our particular interest in specifying algorithms using proof search in linear logic in the next section, we introduce subexponentials in Section 3 and their (focused) proof system in Section 4 and then, in Section 5, we illustrate its algorithmic content with a simple example. In Section 6, we introduce some further machinery, such as *definitions* and new connectives that are able to create new subexponentials, and show how to specify arithmetic operations and how to use subexponentials to locate data. In Section 7 we present the BAG algorithmic specification language, whose operational semantics is captured precisely by the focused proof system. Sections 8 and 9 present some example algorithm specifications along with some comments about proof search complexity. Finally, in Section 10 we describe some related work and in Section 11 we finish with some concluding remarks.

2. Why use logic to specify algorithms?

Why should one care about describing algorithms with logic specifications that claim to be equivalent to them in some sense?

A natural proof theory motivation There is a long tradition of using logic and proof theory as a framework for both functional programming (via proof-normalization) and logic programming (via proof search). While proof theory provides a remarkably robust and deep analysis of abstraction, substitution, and duality, there are several computational phenomena that it alone does not provide information. For examples, proof theory does not offer canonical treatments of first-order quantification, the structure of worlds in modal logics, focusing polarity of atomic formulas, and the exponentials. Such non-canonical aspects of proof theory can often be exploited by the computer scientist. For example, first-order quantification is richly applied in a wide range of applications with greatly varying domains of quantification. Changes in the assignment of focusing bias for atomic formulas in proof search allow one to mix forward-chaining and backward-chaining style proof search to suit applications [12]. This paper provides a partial answer to the question: What computer science relevance can subexponentials serve in the logical specification of computation?

A high-level and declarative specification of algorithms Using logic formulas as specifications has a number of appealing features. First, the operational semantics of such logic specifications can be given within a well understand and rich setting, such as linear logic: in such a setting, operational meaning is clear and precise. In particular, cut-free, focused proof search is used to describe the operational semantics of logic specifications. Second, if algorithm specifications are logic specifications, one would expect that the rich mechanisms available for manipulating and reasoning with logical formulas can then be immediately seen as being manipulations and reasoning on algorithm specifications. Third, logic provides a high-level specification that is often suitable for the specification of algorithms. For example, non-deterministic, algorithmic specifications such as *pick any element from a set* are easily modeled with a precisely matching non-deterministic construction in logic. Of course, if one is specifying a deterministic algorithm, then that algorithm would need to replace the set by, say, a list and select the first element of the list—all these steps can be mirrored in logic specifications as well.

3. Linear Logic and Subexponentials

While we assume that the reader is familiar with the basics of linear logic, we recall here a bit of terminology concerning its syntax and proof theory. *Literals* are either atomic formulas or their negations. The connectives $\otimes$ and $\parr$ and their units 1 and $\bot$ are *multiplicative*; the connectives $\oplus$ and $\&$ and their units 0 and $\top$ are *additive* connectives; $\forall$ and $\exists$ are (first-order) quantifiers; and ! and ? are the exponentials. We shall assume that all formulas are in *negation normal form*, meaning that all negations have atomic scope.

The exponentials in linear logic are not canonical: if one considers, say, a pair of blue exponentials, $?^b$ and $!^b$, and a pair of red exponentials, $?^r$ and $!^r$, then $?^r A$ and $?^b A$ (and $!^r A$ and $!^b A$) are not provably equivalent. Danos *et al.* proposed [5] a linear logic system with non-canonical bangs and question marks: we introduce the term *subexponentials* to denote these. A *subexponential signature* is a tuple $\langle I, \preceq, \mathcal{W}, \mathcal{C}\rangle$ where I is a set of indexes (naming the subexponentials), $\preceq$ is a pre-order on I, and $\mathcal{W}$ and $\mathcal{C}$ are both subsets of I. The sets $\mathcal{W}$ and $\mathcal{C}$ contain those indexes for which the corresponding subexponential allows weakening and contraction, respectively. Both these sets are closed under the order relation: in particular, if $l \preceq k$ and one of these sets contains l, it also contains k.

Given a subexponential signature $\Sigma = \langle I, \preceq, \mathcal{W}, \mathcal{C}\rangle$, the logic SELL_Σ is linear logic with the rules for the exponentials replaced by the rules for the subexponentials. In particular, the dereliction, contraction, and weakening rules are given as follows: here, $y \in \mathcal{C}$ and $z \in \mathcal{W}$.

$$\frac{\vdash C, \Delta}{\vdash ?^x C, \Delta}\ D \qquad \frac{\vdash ?^y C, ?^y C, \Delta}{\vdash ?^y C, \Delta}\ C \qquad \frac{\vdash \Delta}{\vdash ?^z C, \Delta}\ W$$

The promotion rule, a particular focus of this paper, is given by the following inference rule:

$$\frac{\vdash ?^{x_1} C_1, \ldots, ?^{x_n} C_n, C}{\vdash ?^{x_1} C_1, \ldots, ?^{x_n} C_n, !^a C}\ !^a$$

where $a \preceq x_i$ for all $i = 1, \ldots, n$. Notice that if $x \not\preceq y$ then the promotion rule can be applied to the sequent

$$\vdash ?^x C_1, \ldots, ?^x C_n, ?^y D_1, \ldots, ?^y D_m, !^y C$$

only if $n = 0$: the premise of that promotion rule would then be $\vdash ?^y D_1, \ldots, ?^y D_m, C$. The promotion rule can then be seen as a kind of "guard" that allows a certain proof-search reduction only when the collection $\{C_1, \ldots, C_n\}$ "located" at x is empty. Danos *et al.* also showed that *SELL* admits cut-elimination and that the initial rule can be restricted to atomic formulas [5].

In the scope of the current paper, we shall make two additional assumptions about the collection of subexponentials that we consider. First, we shall always assume that $\mathcal{C} = \mathcal{W}$: that is, a subexponential either admits weakening and contraction (these will be the *unbounded* subexponentials) or admits neither weakening nor contraction (these will be the *bounded* subexponentials). Notice that the unbounded subexponentials are the only ones that should be called *exponentials* since for them it is possible to prove the equivalence between $?(A \oplus B)$ and $?A \parr ?B$. Subexponentials that admit either weakening or contraction, but not both, do not play a role in this paper. Second, we shall assume that the pre-ordered set $\langle I, \preceq\rangle$ has a maximal element, written ∞, which is unbounded.

SELL is not a new logic: it is simply linear logic in which the exponentials are allowed to have weaker behavior (that is, allow fewer proofs). Other ways to exploit the non-canonical nature of linear logic's exponential are explored in, for example, light and elementary linear logics [10].

4. Focused proofs with Subexponentials

In [1], Andreoli presented a *focused proof system* for linear logic: similar focused proof systems have subsequently been introduced for classical and intuitionistic logic (see, for example, [12]). Such proof systems provide a normal form for cut-free proofs where introduction rules are organized in such a way that the "micro-rules" of sequent calculus that introduce individual logical connectives are combined into "macro-rules" that can be seen as introduction rules for synthetic connectives. The back-chaining proof search step of logic programming can be seen as interpreting logic programming clauses as such synthetic connectives.

Before we introduce the focused system for *SELL*, we classify as *positive* the formulas whose main connective is either $\otimes, \oplus, \exists$, the subexponential bang, the unit 1 and positive literals. All other formulas are classified as *negative*. Figure 1 contains the focused proof system *SELLF* that is a rather straightforward generalization of Andreoli's original system. There are two kinds of arrows in this proof system. Sequents with the $\Downarrow$ belong to the *positive* phase and introduce the logical connective of the "focused" formula (the one to the right of the arrow): building proofs of such sequents maybe require non-invertible proof steps to be taken. Sequents with the $\Uparrow$ belong to the *negative* phase and decompose the formulas on their right in such a way that only invertible inference rules are applied. The rules $[R\Downarrow]$, $[D_l]$, and $[!^l]$ provide the only inference rules that mix these two kinds of sequents: as such, these rules mark the boundary between the macro rules, that is, between negative and positive phases. Synthetic connectives can be seen as being introduced by such macro rules.

Similarly as in the usual presentation of linear logic, there is a pair of contexts to the left of $\Uparrow$ and $\Downarrow$ of sequents, written here as $\mathcal{K} : \Gamma$. The second context, Γ, collects the formulas whose main connective is not a question-mark, behaving as the bounded context in linear logic. But differently from linear logic, where the first context is a multiset of formulas whose main connective is a question-mark, we generalize $\mathcal{K}$

to be an *indexed context*, which is a mapping from each index in the set I (for some given and fixed subexponential signature) to a finite multiset of formulas, in order to accommodate for more than one subexponential in *SELLF*. In Andreoli's focused system for linear logic, the index set contains just ∞ and $\mathcal{K}[\infty]$ contains the set of unbounded formulas. Given a subexponentials signature, $\Sigma = \langle I, \preceq, \mathcal{W}, \mathcal{C}\rangle$, we specify the following operations over these contexts:

- $\mathcal{K} \leq_i [l] = \begin{cases} \mathcal{K}[l] & \text{if } i \preceq l \\ \emptyset & \text{if } i \not\preceq l \end{cases}$ where $i \in I$ is a subexponential index.
- $\mathcal{K}[\mathcal{S}] = \bigcup\{\mathcal{K}[i] \mid i \in \mathcal{S}\}$
  where $\mathcal{S} \subseteq I$ is a subset of subexponential indexes.
- $(\mathcal{K} +_l A)[i] = \begin{cases} \mathcal{K}[i] \cup \{A\} & \text{if } i = l \\ \mathcal{K}[i] & \text{otherwise} \end{cases}$
  where A is a formula.
- Let $\mathcal{S} \subseteq I$ be a set of subexponential indexes, and let $\star \in \{=, \subset, \subseteq\}$ be a binary connective. Then $(\mathcal{K}_1 \star \mathcal{K}_2) \mid_{\mathcal{S}}$ is true if and only if $\forall i \in \mathcal{S}.(\mathcal{K}_1[i] \star \mathcal{K}_2[i])$.
- $(\mathcal{K}_1 \otimes \mathcal{K}_2)[i] = \begin{cases} \mathcal{K}_1[i] \cup \mathcal{K}_2[i] & \text{if } i \notin \mathcal{C} \\ \mathcal{K}_1[i] & \text{if } i \in \mathcal{C} \cap \mathcal{W} \end{cases}$

The following soundness and completeness of *SELLF* can be proved, for example, using techniques from [1, 19].

PROPOSITION 1. *Let $\Sigma = \langle I, \preceq, \mathcal{W}, \mathcal{C}\rangle$ be a subexponential signature, such that $\mathcal{W} = \mathcal{C}$. Then SELLF$_\Sigma$ is sound and complete with respect to SELL$_\Sigma$.*

5. Example: a minimal element of a multiset

Before we make some simple extensions to *SELLF*, we illustrate with a small example the increase of expressivity obtained by using subexponentials. Consider the nonempty multiset of natural numbers $\{m_1, \ldots, m_n\}$. Let $\langle\{\infty, l, k\}, \{k \preceq \infty, l \preceq \infty\}, \{\infty\}, \{\infty\}\rangle$ be a subexponential signature where l and k are not $\preceq$-comparable. Also, assume that all atoms are assigned with negative polarity and let $\mathcal{K}$ be the indexed context where $\mathcal{K}[\infty]$ is the set

$$\{\ \exists x \exists y [l(x)^\perp \otimes l(y)^\perp \otimes (x \leq y) \otimes ?^l l(x)],\\ \exists x [l(x)^\perp \otimes !^k\, min(x)]\ \},$$

$\mathcal{K}[k] = \emptyset$, and $\mathcal{K}[l] = \{l(m_1), \ldots, l(m_n)\}$. This context contains exactly two positive formulas and, hence, there are only two formulas on which to focus. We now derive in detail these two synthetic connectives.

Focusing on the first formula requires building the following derivation bottom-up:

$$\dfrac{\dfrac{\vdash \mathcal{K} : \cdot \Downarrow l(m_i)^\perp \otimes l(m_j)^\perp \otimes (m_i \leq m_j) \otimes ?^l l(m_i)}{\vdash \mathcal{K} : \cdot \Downarrow \exists x \exists y [l(x)^\perp \otimes l(y)^\perp \otimes (x \leq y) \otimes ?^l l(x)]}\ [2 \times \exists]}{\vdash \mathcal{K} : \cdot \Uparrow}\ [D_\infty]$$

Continuing this phase of the proof requires finding four indexed contexts such that $\mathcal{K}_1 \otimes \mathcal{K}_2 \otimes \mathcal{K}_3 \otimes \mathcal{K}_4[l] = \mathcal{K}[l]$ for all l and such that the following four sequents $\vdash \mathcal{K}_1 : \cdot \Downarrow l(m_i)^\perp$, $\vdash \mathcal{K}_2 : \cdot \Downarrow l(m_j)^\perp$, $\vdash \mathcal{K}_3 : \cdot \Downarrow m_i \leq m_j$, and $\vdash \mathcal{K}_4 : \cdot \Downarrow ?^l l(m_i)$ are provable. The first two sequents are provable if and only if $\mathcal{K}_1[l] = \{l(m_i)\}$ and $\mathcal{K}_2[l] = \{l(m_j)\}$[1]. The third sequent is provable if m_i is less than or equal to m_j and $\mathcal{K}_3[l] = \{\}$. This means that $\mathcal{K}_4$ is the same as $\mathcal{K}$ except that $\mathcal{K}_4[l]$ is the multiset $\mathcal{K}[l]$ less the two distinct elements $l(m_i)$ and $l(m_j)$ (hence, $n > 1$). The remainder of this proof phase is necessarily of the form:

$$\dfrac{\dfrac{\vdash \mathcal{K}_4 +_l l(m_i) : \cdot \Uparrow \cdot}{\vdash \mathcal{K}_4 : \cdot \Uparrow ?^l l(m_i)}\ [?^l]}{\vdash \mathcal{K}_4 : \cdot \Downarrow ?^l l(m_i)}\ [R\Downarrow]$$

In other words, the synthetic connective arising from focusing on the first formula in the logic specification provides a proof of the sequent $\vdash \mathcal{K} : \cdot \Uparrow \cdot$ from the premise $\vdash \mathcal{K}' : \cdot \Uparrow \cdot$ exactly when $\mathcal{K}[l]$ contains at least 2 elements and $\mathcal{K}'$ is the same as $\mathcal{K}$ except that $\mathcal{K}'[l]$ results from $\mathcal{K}[l]$ by deleting one atom holding an integer greater than or equal to another integer in that multiset.

If we focus on the second formula, the resulting "macro" rule is built from the following "micro" rules.

$$\dfrac{\dfrac{\dfrac{\vdash \mathcal{K}_1 : \cdot \Downarrow l(m)^\perp}{}\ [I] \quad \dfrac{\dfrac{\vdash \mathcal{K}_2 : min(m) \Uparrow}{\vdash \mathcal{K}_2 : \cdot \Uparrow min(m)}\ [R\Uparrow]}{\vdash \mathcal{K}_2 : \cdot \Downarrow !^k\, min(m)}\ [!^k]}{\vdash \mathcal{K} : \cdot \Downarrow l(m)^\perp \otimes !^k\, min(m)}\ [\otimes]}{\vdash \mathcal{K} : \cdot \Downarrow \exists x [l(x)^\perp \otimes !^k\, min(x)]}\ [\exists]}{\vdash \mathcal{K} : \cdot \Uparrow \cdot}\ [D_\infty]$$

Here, $\mathcal{K}_1 \otimes \mathcal{K}_2 = \mathcal{K}$ and $\mathcal{K}_1[l] = \{l(m)\}$. Also, $\mathcal{K}_2[l]$ is empty, a fact guaranteed by the promotion rule and the fact that l and k are not $\preceq$-comparable. Thus, the corresponding synthetic connective provides a proof of the sequent $\vdash \mathcal{K} : \cdot \Uparrow$ from the premise $\vdash \mathcal{K}' : min(m) \Uparrow$ only when $\mathcal{K}[l]$ contains exactly one element (m) and $\mathcal{K}'$ is the result of setting the multiset $\mathcal{K}'[l]$ to the empty multiset.

The logic specification above clearly computes the minimal member of a multiset in a structured fashion: if the number of elements in the multiset (in location l) is one, then the minimum is found; and if the number of elements is more than one, then one element is discarded that does not affect the minimum. These two steps are described by focusing on different clauses. Notice that a proof using these clauses does not involve any backtracking from the point of synthetic connectives, while internal to the synthetic connective one might envision possible backtracking search (for example, to find m_i and m_j such that $m_i \leq m_j$).

[1] Remember that atoms, such as $l(m)$, are assigned with negative polarity and hence, $l(m)^\perp$ is assigned with positive polarity. Moreover, only the initial rule can introduce a focused literal with positive polarity.

$$\frac{}{\vdash \mathcal{K} : \Gamma \Uparrow L, \top}\ [\top] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A \quad \vdash \mathcal{K} : \Gamma \Uparrow L, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \,\&\, B}\ [\&] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, \bot}\ [\bot] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \parr B}\ [\parr]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A\{c/x\}}{\vdash \mathcal{K} : \Gamma \Uparrow L, \forall x\, A}\ [\forall] \qquad \frac{\vdash \mathcal{K} +_l A : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, ?^l A}\ [?^l]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow A_i}{\vdash \mathcal{K} : \Gamma \Downarrow A_1 \oplus A_2}\ [\oplus_i] \qquad \frac{\vdash \mathcal{K}_1 : \Gamma \Downarrow A \quad \vdash \mathcal{K}_2 : \Delta \Downarrow B}{\vdash \mathcal{K}_1 \otimes \mathcal{K}_2 : \Gamma, \Delta \Downarrow A \otimes B}\ [\otimes],\ \text{provided } (\mathcal{K}_1 = \mathcal{K}_2)\ |_{\mathcal{C} \cap \mathcal{W}}$$

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow 1}\ [1],\ \text{provided } \mathcal{K}[I \setminus \mathcal{W}] = \emptyset \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow A\{t/x\}}{\vdash \mathcal{K} : \Gamma \Downarrow \exists x\, A}\ [\exists]$$

$$\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l\, A}\ [!^l],\ \text{provided } \mathcal{K}[\{x \mid l \not\preceq x \wedge x \notin \mathcal{W}\}] = \emptyset$$

$$\frac{}{\vdash \mathcal{K} : \Gamma \Downarrow A_p}\ [I],\ \text{provided } A_p^{\perp} \in (\Gamma \cup \mathcal{K}[I]) \text{ and } (\Gamma \cup \mathcal{K}[I \setminus \mathcal{W}]) \subseteq \{A_p^{\perp}\}$$

$$\frac{\vdash \mathcal{K} +_l P : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l],\ \text{provided } l \in \mathcal{C} \cap \mathcal{W} \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l],\ \text{provided } l \notin \mathcal{C} \cap \mathcal{W}$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} : \Gamma, P \Uparrow \cdot}\ [D_1] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow N}{\vdash \mathcal{K} : \Gamma \Downarrow N}\ [R\Downarrow] \qquad \frac{\vdash \mathcal{K} : \Gamma, S \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, S}\ [R\Uparrow]$$

Figure 1. The focused linear logic system *SELLF*. Here, A_p is a positive literal; S is a positive formula or a literal; P is a not a negative polarity literal; and N is a negative formula.

6. Deploying *SELLF*

In order to better illustrate some algorithm specifications in *SELLF*, we introduce the following machinery. First, we add the ability to *define* atomic formulas and introduce rules for unfolding such definitions during proof search. Such definitions then allow us to specify *arithmetic operations*. We then explain how we can represent *data structures* in *SELLF*, using subexponentials to *locate* data structures. Finally, we propose a new connective to linear logic that can be used to create new locations during proof search.

6.1 Adding Definitions

A *definition* is a finite set of *clauses* which are written as $\forall \bar{x}[p\,\bar{x} \triangleq B]$: here p is a predicate and every free variable of B (the *body* of the clause) is contained in the list $\bar{x}$. The symbol $\triangleq$ is not a logical connective but is used to indicate a definitional clause. We consider that every defined predicate occurs at the head of exactly one clause. The following two "unfolding" rules are added to *SELLF*.

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow B\theta}{\vdash \mathcal{K} : \Gamma \Downarrow p\,\bar{t}}\ [def\Downarrow] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, B\theta}{\vdash \mathcal{K} : \Gamma \Uparrow L, p\,\bar{t}}\ [def\Uparrow]$$

The proviso for both of these rules is: $\forall \bar{x}[p\,\bar{x} \triangleq B]$ is a definition clause and θ is the substitution that maps the variables $\bar{x}$ to the terms $\bar{t}$, respectively. Thus, in either phase of focusing, if a defined atom is encountered, it is simply replaced by its definition and the proof search phase does not change. The proof theory of inference rules such as these is well studied (see, for example, [2, 16]).

6.2 Including arithmetic

Several of the examples and algorithms we consider in this paper will need integers and some basic arithmetic operators on them. These all can be accommodated easily within *SELLF* in a purely "positive" setting. In particular, the arithmetic comparisons for integers, $\leq, <, =, \neq, >, \geq$, are available as binary predicates within *SELLF* by using definitions. For example, the definition for $\leq$ is

$$x \leq y \ \triangleq\ [x = z] \oplus \\ [\exists x' y' (x = s\,x') \otimes (y = s\,y') \otimes x' \leq y'].$$

Here, zero is denoted by the constant $z$and successor by the constructor s. The other arithmetic comparisons are specified in a similar way.

If $\blacktriangledown$ denotes one of these relations, then the formula $m \blacktriangledown n$ is positive and provable instances of it are composed of exactly one positive phase and without the consumption of any formulas from the context. More formally, if $\mathcal{K}$ is an indexed context then $\vdash \mathcal{K} : \Gamma \Downarrow m \blacktriangledown n$ is provable if and only if m and n are integers that stand in the relation intended by $\blacktriangledown$ and $\Gamma \cup \mathcal{K}[I \setminus \mathcal{W}]$ is empty. We write $\widetilde{\blacktriangledown}$ to be the comparison that is the complement to the one denoted by $\blacktriangledown$: *e.g.*, $s \,\widetilde{\leq}\, t$ is $s > t$.

We assume that basic integer addition and multiplication are also available as purely positive synthetic connectives. In particular, expressions such as $x \leq y + w$ are replaced by $\exists u.plus\ y\ w\ u \otimes x \leq u$, where $plus\ y\ w\ u$ denotes the relation between y and w and their sum u and is specified by

the following definition:

$$plus\ y\ w\ u \ \stackrel{\Delta}{=}\ [y = z \otimes w = u] \oplus [\exists y' u' (y = s\, y') \otimes (u = s\, u') \otimes plus\ y'\ w\ u'].$$

6.3 Representing Data Structures

As we described in Section 1, most of the dynamics of logic programming within classical and intuitionistic logic occurs within atomic formulas: thus, data structures are usually encoded as term structures so that they can appear within the scope of predicate constants. For example, a set of pairs $\{\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\}$ can be encoded as the term $((x_1 :: y_1 ::\text{nil}) :: \cdots :: (x_n :: y_n ::\text{nil}) ::\text{nil})$, where :: and nil are the non-empty and empty set constructors. In *SELLF*, it is possible to encode many data structures using multisets of formulas instead of terms. For example, the same set of pairs can be represented as

$$?^l\text{rel}(x_1, y_1), \ldots, ?^l\text{rel}(x_n, y_n)$$

in which the *subexponential* l provides a "location" for this data structure. Furthermore, the collection of formulas above encodes a *set* if $l \in \mathcal{C} \cap \mathcal{W}$ or a *multiset* if $l \notin (\mathcal{C} \cup \mathcal{W})$.

In the rest of this paper, we constrain indexed contexts as follows: for any subexponential $l \in I$, the multiset $\mathcal{K}[l]$ contains only atomic formulas and these are built with a predicate whose name is the same as l. Linking the predicate name of atomic formulas to their locations in this way is a convenience for the examples we shall consider. We also assume that all atoms used to encode data, *i.e.*, atoms in $\mathcal{K}[l]$ will be assigned negative polarity.

6.4 Complements of locations

Since we will soon turn our attention to algorithm specifications in *SELLF*, we shall make two further restrictions in how we deploy *SELLF*.

First, we shall assume that all locations, l, except the special unbounded maximal location ∞, are bounded (that is $\mathcal{W} = \mathcal{C} = \{\infty\}$) and $l \preceq \infty$. Thus, no two locations will be considered sublocations.

Second, as the example above illustrated, testing that a given location l is empty required the promotion rule with a location k such that $k \not\preceq l$. To ensure that we have the ability to perform all such tests, we shall define the *complement* to the subexponential signature $\langle\{\infty\} \cup I, \preceq, \{\infty\}, \{\infty\}\rangle$ to be the signature $\langle\{\infty\} \cup I \cup \hat{I}, \hat{\preceq}, \{\infty\}, \{\infty\}\rangle$ where $\hat{I}$ is a copy of I containing elements of the form $\hat{l}$ whenever $l \in I$. The order relation $\preceq$ is extended with all pairs $\hat{l}\ \hat{\preceq}\ k$ such that l and k are distinct members of I. Thus, in the complemented signature, $\hat{l}$ can be seen as a sublocation of all locations in I different from l. The promotion rule with the subexponential $!^{\hat{l}}$ succeeds only if the indexed context is empty at location l: all other locations need not be considered. Data will not be "store" in complemented locations: that is, $\mathcal{K}[\hat{l}]$ will always be empty.

$$\frac{\Sigma \cup \Sigma_l \vdash \mathcal{K}, C}{\Sigma \vdash \mathcal{K}, \Cap\Sigma_l.C} \text{ if } \Sigma \cup \Sigma_l \text{ is a subexponential signature.}$$

$$\frac{\Sigma \vdash \mathcal{K}, C[s_1/l_1, \ldots, s_n/l_n]}{\Sigma \vdash \mathcal{K}, \Cup\Sigma_l.C} \text{ if } \Sigma_l[s_1/l_1, \ldots, s_n/l_n] \subseteq \Sigma$$

Figure 2. The introduction rules for $\Cap$ and $\Cup$. Here, the subexponential signatures $\Sigma = \langle I, \preceq, \mathcal{W}, \mathcal{C}\rangle$ and $\Sigma_l = \langle I_l, \preceq_l, \mathcal{W}_l, \mathcal{C}_l\rangle$ are such that $I_l = \{l_1, \ldots, l_n\}$ is a set of new indexes, the relation $\preceq_l \subseteq \bar{I} \times \bar{I}$ is a pre-order, and the sets $\mathcal{W}_l$ and $\mathcal{C}_l$ are subsets of $\bar{I} = I_l \cup I$.

$$\frac{\mathcal{L} \cup \{loc\} \vdash \mathcal{K}, C}{\mathcal{L} \vdash \mathcal{K}, \Cap_l loc.C}\ [\Cap_l] \text{ provided } loc \text{ is a new location}$$

$$\frac{\mathcal{L} \vdash \mathcal{K}, C[s/loc, \widehat{s}/\widehat{loc}]}{\mathcal{L} \vdash \mathcal{K}, \Cup_l loc.C}\ [\Cup_l] \text{ provided } s \in \mathcal{L}$$

Figure 3. The introduction rules for $\Cap_l$ and $\Cup_l$. Here $\mathcal{L}$ is a set of locations.

6.5 Creation of new locations

Up to now, all locations are fixed throughout a proof. One can imagine simple extensions to linear logic that allow the creation of new locations within proofs. We show here two possible extensions. The first extension is the logic *SELL*$^{\Cap}$ which allows more arbitrary changes of the subexponential signature. It extends *SELL* with two new connectives, $\Cap$ and $\Cup$: the proof rules for these connectives are given in Figure 2. We write the union of two signatures to be their point-wise union and the inclusion of two signatures to be their point-wise inclusion. These connectives act as binding that can introduce new locations ($\Cap$) and be instantiated by old locations ($\Cup$). It is a simple matter to see that cut-elimination holds for *SELL*$^{\Cap}$ and that focusing proof systems are complete when we assign $\Cap$ a negative polarity and $\Cup$ a positive polarity.

THEOREM 2. *The cut rule is admissible in* *SELL*$^{\Cap}$.

The second extension is an specialization of *SELL*$^{\Cap}$, called *SELL*$^{\Cap_l}$, that instead of considering general subexponential signatures, assumes only a set $\mathcal{L}$, containing all the bounded locations available to store data, and the existence of their complement locations, as discussed Subsection 6.4. It contains the connectives $\Cap_l$ and $\Cup_l$, whose introduction rules are given in Figure 3. The introduction rule for $\Cap_l$ creates a new location and its complement location, and the introduction rule for $\Cup_l$ instantiates all occurrences in C of loc by s and of $\widehat{loc}$ by $\widehat{s}$.

7. Specifying Algorithms

There is a high-degree of "algorithmic context" in the description of synthetic connectives within *SELL*, especially once we made a few restrictions to that logic. In order to make the scope of such algorithmic specifications more evident, we present a small specification language that can be used to describe some *single-threaded* algorithms: while *multi-threaded* algorithmic specifications are possible in linear logic (see, for example, [17]), we focus here on more traditional and determinant algorithmic specifications.

The following grammar introduces a high-level syntax for a small specification language we call BAG. We shall take as given a subexponential signature Σ (restricted as described in Section 6.4). The set of constants $\mathbb{C}$ is also fixed and contains the natural numbers plus other tokens that we may need, such as $blue$, red, *etc*. We allow for two kinds of variables: members of $var \in \mathcal{V}$ denote variables over the first-order domain $\mathbb{C}$, while members of $K \in \mathcal{K}$ denote variables over programs (continuations). To facilitate the construction of specifications in BAG, we introduce a new kind of variable $L \in \mathcal{L}$ for locations and introduce a set of constants $name \in \mathcal{N}$ for module names. The other syntactic classes can be defined as follow.

$$
\begin{array}{lll}
t & ::= c \in \mathbb{C} \mid var & tup ::= \langle t_1, \ldots, t_n \rangle \ (n \geq 0) \\
pat & ::= tup \mid \lambda var.pat & \\
cond_a & ::= t_1 \blacktriangledown t_2 & cond_l ::= \textbf{is\_empty}\ loc_b \\
cond & ::= cond_a \mid cond_l & \\
prog & ::= \textbf{load}\ tup\ loc\ prog \mid \textbf{unload}_i\ loc\ pat\ bprog & \\
 & \quad \mid \textbf{while}\ cond_a\ (\lambda K.prog)\ prog & \\
 & \quad \mid \textbf{loop}_i\ loc_b\ kprog\ prog \mid \textbf{new}\ loc\ \lambda L.prog & \\
 & \quad \mid \textbf{if}\ cond\ prog \mid prog \,[\!]\, prog \mid K \mid \textbf{end} & \\
bprog & ::= prog \mid \lambda var.bprog & \\
kprog & ::= \lambda K.prog \mid \lambda var.kprog & \\
lprog & ::= \lambda K.prog \mid \lambda L.lprog \mid \lambda var.lprog & \\
mod & ::= name \times lprog. &
\end{array}
$$

Conditions (tests) are of two kind: $cond_a$ are arithmetic tests (see Section 6.2) and $cond_l$ are test that determine if a given location is empty. The syntactic variable loc_b ranges over all bounded locations (here, all locations other than ∞). In the $\textbf{unload}_i$ (respectively, $\textbf{loop}_i$) instruction, we will also insist that pat and $bprog$ (respectively, $kprog$) both have exactly i variables bindings. Moreover, when a module is used in a program, execution proceeds by computing the program resulting from performing the necessary λ-conversions. Since BAGis single-threaded, modules contain one and only one abstracted continuation variable.

The BAG language has the following eight kinds of construction. (1) ($\textbf{load}\ tup\ loc\ prog$) inserts the tuple tup in the location loc and then continues with $prog$. (2) ($\textbf{unload}_i\ loc\ pat\ bprog$) picks an element, $\langle t_1, \ldots, t_n \rangle$, from the location loc such that it matches with the term $pat\, t^1 \cdots t^i$ for some $t^j \in \{t_1, \ldots, t_n\}$ and then executes the program $(bprog\ t^1 \cdots t^i)$. (3) ($\textbf{if}\ cond\ prog$) executes $prog$ if the condition $cond$ holds. (4) ($\textbf{loop}_i\ loc_b\ kprog\ bprog$) repeatedly executes an **unload** of location loc_b with the general pattern $\lambda x_1 \ldots \lambda x_i.\langle x_1, \ldots, x_i \rangle$, using the continuation $kprog$ if the unload is successful and $bprog$ if the unload is not successful. Intuitively, this loop is used to process all members of a location. (5) ($\textbf{while}\ cond_a\ kprog\ prog$) repeatedly applies $kprog$ until the condition is not true; then $prog$ is executed. (6) ($\textbf{new}\ loc\ lprog$) creates a new location loc and then executes the program $(lprog\ loc)$. (7) $(prog_1 \,[\!]\, prog_2)$ is an *alternative* instruction, where the computation proceeds to either $prog_1$ or $prog_2$. Lastly, (8) **end** ends the computation thread. Notice that this language is similar to Dijkstra's Guarded Command Language (GCL) [6]: in particular, the $[\!]$ instruction is similar to GCL's *if* constructs, and the **while** and **loop** instructions are similar to GCL's *loop* constructs.

Our wish here is not to describe a new specification language but to highlight the algorithmic aspects already present within focused proof search in *SELL*. To this end, we show how the intended operational semantics of the BAG language can be specified by mapping it directly into *SELL* formulas. In particular, the non-determinism that exists in an algorithmic description using BAG matches exactly the non-determinism in *SELLF*'s at the level of synthetic connectives. The exact algorithmic nature of computing a synthetic connective *internally* will be considered in Section 9: there we show that it is often possible to implement the inference rules for synthetic connective in constant time.

Being able to specify when a synthetic connective ends is critical to our claims that focused proof search and algorithms in BAG are closely related. The two *delay operators* $\delta^-(\cdot)$ and $\delta^+(\cdot)$ can be used to replace a formula with a provably equivalent formula of a given polarity. In particular, $\delta^-(C)$ is negative no matter what polarity C is: it can be defined as $C \parr \bot$. Similarly, $\delta^+(C)$ is positive no matter what polarity C is: it can be defined as $C \otimes 1$.

The definition $\mathcal{D}$ in Figure 4 specifies a "proof theoretic" semantics of the BAG language. (For readability, we have suppressed writing the outermost universal quantifiers on these clauses.) The alternation of polarities, the use of the subexponential $!^{\hat{l}}$, and the placement of delays in this definition are particularly important to notice. For example, the meaning of the **load** command is given using a negative formulas as its body: this command proceeds without needing any coordination with anything in the context, as illustrates the following derivation:

$$
\dfrac{\dfrac{\dfrac{\vdash \mathcal{K} +_l l(\bar{t}) : \cdot \Uparrow \delta^+(prog)}{\vdash \mathcal{K} : \cdot \Uparrow ?^l l(\bar{t}), \delta^+(prog)}\,[?^l]}{\vdash \mathcal{K} : \cdot \Uparrow ?^l l(\bar{t}) \parr \delta^+(prog)}\,[\parr]}{\vdash \mathcal{K} : \cdot \Uparrow \textbf{load}\ \langle \vec{t} \rangle\ prog}\,[def\ \Uparrow]
$$

Because of the positive delay $\delta^+(\cdot)$, it must be the case that the negative phase ends by performing $[R\Uparrow]$. Thus, this

$$
\begin{array}{lcl}
\textbf{load}\ \langle \bar{t} \rangle\ l\ prog & \triangleq & ?^{l} l(\bar{t}) \parr \delta^{+}(prog) \\
\textbf{unload}_i\ l\ pat\ bprog & \triangleq & l(pat\ v_1 \cdots v_i)^{\perp} \otimes [\delta^{-}(bprog\ v_1 \cdots v_i)] \\
\textbf{while}\ (t_1 \blacktriangledown t_2)\ kprog\ prog & \triangleq & [(t_1 \blacktriangledown t_2) \otimes \delta^{-}(kprog\ (\textbf{while}\ (t_1 \blacktriangledown t_2)\ kprog\ prog))] \oplus [(t_1 \widetilde{\blacktriangledown} t_2) \otimes \delta^{-}(prog)] \\
\textbf{loop}_i\ l\ kprog\ prog & \triangleq & [l(v_1, \ldots, v_i)^{\perp} \otimes \delta^{-}((kprog\ v_1 \cdots v_i)\ (\textbf{loop}_i\ l\ kprog\ prog))] \oplus\ !^{\hat{l}}(prog) \\
prog_1 \,[]\, prog_2 & \triangleq & prog_1 \oplus prog_2 \\
\textbf{if}\ (\textbf{is\_empty}\ l)\ prog & \triangleq & !^{\hat{l}}(prog) \\
\textbf{if}\ (t_1 \blacktriangledown t_2)\ prog & \triangleq & t_1 \blacktriangledown t_2 \otimes \delta^{-}(prog) \\
\textbf{new}\ loc\ lprog & \triangleq & \Bbbn_l\ lprog \\
\textbf{end} & \triangleq & \perp
\end{array}
$$

Figure 4. The definition clauses for specifying the execution of BAG programs.

specification for **load** corresponds to the intended operation of loading *exactly one* tuple in a location.

All other instructions (except for **end** and **new**) are defined by positive formulas. In these cases, choices must be made and backtracking might be necessary inside a positive phase. For example, if one is focused on a **while** instruction then that focus continues on a formula of the form

$$\Downarrow [(t_1 \blacktriangledown t_2) \otimes \delta^{-}(C)] \oplus [(t_1 \widetilde{\blacktriangledown} t_2) \otimes \delta^{-}(D)]$$

At the "micro-rule level," proof search must pick between the two branches of the $\oplus$ and then determine which branch succeeds: at this level, some search may be required to compute the proper macro-step, but in the end, proof search will continue with either $\Uparrow C$ or with $\Uparrow D$ (the occurrences of $\delta^{-}(\cdot)$ forces the flip of $\Downarrow$ to $\Uparrow$): here, the choice is completely determined by the guards and this is reflected also with the "macro-level" inference rules.

Notice that there are no delays written into the definition of the [] operator since we wish that the choice provided by that operator is merged with choices in the instructions it accumulates. For example, the instructions

$$(\textbf{if}\ (x \leq y)\ prog_1) \,[]\, (\textbf{if}\ (\textbf{is\_empty}\ l)\ prog_2)$$

are equated, via the definition mechanism, to the formula

$$((x \leq y) \otimes \delta^{-}(prog_1)) \oplus\ !^{\hat{l}} prog_2.$$

This synthetic connective combines internally the test $x \leq y$ with the emptiness check of location l. As described in Section 4, the rule for $!^{\hat{l}}$ terminates the $\Downarrow$ focus.

The correspondence between focused inference rules and algorithmic steps is precise: in particular, all partial proofs involving synthetic connectives match exactly the algorithmic steps that are possible. Thus, algorithmic steps that lead to failures are matched exactly with partial proofs that cannot be extended to complete proofs. As the behavior of an algorithm corresponds to the set of all its possible computation runs, this implies that the focused derivations obtained from Figure 4 capture exactly the behavior of BAG programs.

We now illustrate how the *full adequacy* result can be used to control the size of synthetic connectives: as a result, it is possible to capture different intended semantics for BAG and to change the behavior of its programs. For example, the operational semantics for alternation [] can be changed from the *guarded* choice given before to the purely *local* choice, where a choice is made before considering any aspect of different alternatives. Such a local choice can be specified simply by using delays:

$$prog_1 \,[]\, prog_2 \quad \triangleq \quad \delta^{-}(prog_1) \oplus \delta^{-}(prog_2).$$

Because of the extra negative delay operators, the positive phase must stop before applying the first instruction of the selected program. In this case, while the number of successful computation runs of a program does not change, the number of computation runs that fail might increase.

On the other hand, increasing the size of synthetic connectives, by removing delay operators, increases the amount of computation *packaged* in a synthetic connective. Consider, for example, a new definition for the **unload** instruction that does not contain a negative delay operator. In this case, one captures the intended semantics where all consecutive **unload** commands are performed in a single step. Since the non-determinism involved in picking the right tuples to unload is contained in the execution of a single transition step, the number of computation runs that succeed does not change, but the number of computation runs that fail might decrease.

Notice that since the **unload** and **load** operations are defined using dual connectives ($\otimes$ and $\parr$, respectively), they cannot be part of the same synthetic connective. Such a restriction on a synthetic connective (and on the associated algorithmic step) is sensible since the order in which one performs these operations can lead to different results.

8. Examples

The module $extractMin$, that extracts the minimum element from a multiset, is depicted in Figure 5. (For readability, the λ-abstractions associated with **unload** and **new** statements

are elided, and we denote programs of the form $A\ (B\ C)$ as $(A;\ B\ C)$.) This module takes three locations, l_i, l_o, and min, and a continuation program $prog$. The module moves the minimum element of the multiset, located in l_i, to the location min, and moves its remaining elements to the location l_o.

$extractMin = \lambda l_i \lambda l_o \lambda min \lambda prog.$
unload$_2$ l_i $\langle n, v\rangle$
  load $\langle n, v\rangle$ min
  loop$_2$ l_i $\lambda n_1 \lambda v_1 \lambda lcont$
    unload$_2$ min $\langle n_m, v_m\rangle$
      if $(v_m \leq v_1)$
        load $\langle n_m, v_m\rangle$ min (**load** $\langle n_1, v_1\rangle$ l_o $lcont$)
      [] **if** $(v_m > v_1)$
          load $\langle n_1, v_1\rangle$ min (**load** $\langle n_m, v_m\rangle$ l_o $lcont$)
  $prog$

Figure 5. Extracting the minimum element

The BAG program $\mathcal{P}^{\mathcal{G}}_{bp}$ in Figure 6 checks if a graph, $\mathcal{G}$, is bipartite. It takes as input three locations, for which all, except ver, are empty. Initially, all nodes are $gray$ and later their color can change to $blue$ or red. We use the location ver to store the nodes that are gray and the location col to store the nodes' color information. First, we create two auxiliary locations pr and $edges$. The first loop performs the initialization of the nodes' colors. Then, the second loop starts to traverse a new component of the graph, by picking any node from ver, assigning it the color $blue$, and inserting it in the auxiliary location pr. The inner loop, that traverses through a component of the graph, starts by picking any node, s, in pr. It then, invokes the module $getEdges$ that loads the edges connected to s in the location $edges$. This module can be seen as a series of alternatives of **if** instructions, that checks the input node and loads accordingly the edges in a specified location. The third loop traverses through these edges. There are two alternatives, either s is $blue$ or it is red. If it is blue, it checks if all adjacent nodes, adj, are assigned the correct color (red), or assigns it the correct color and insert it in the location pr, or alternatively if adj is $blue$ then the answer no is loaded in location ans and program finishes by proceeding to $prog$. A similar procedure is performed when s is red. If all nodes in ver are consumed then the graph is bipartite and the answer yes is loaded in the location ans.

The second example is the Dijkstra's algorithm that finds the shortest distance in a positively weighted graph, $\mathcal{G}$, which is specified by the program, $\mathcal{P}^{\mathcal{G}}_{dj}$, depicted in Figure 7. It contains two modules, the main module initializes the location ver by assigning the distance to all nodes to infinity, except the source node, src, whose distance is zero, and then calls the second module $dijkstra$. This module starts with two alternatives: if ver is empty, then the program ends with the shortest distances located in $dist$; or, it invokes the $extractMin$ module, described before, to extract from ver the node, n_m, that has the minimum distance, which will be located in the auxiliary location min. The remaining nodes are transferred to the auxiliary location ver'. Then it adds n_m together with its distance in the location $dist$. Next, it invokes the module $getEdges$ which loads in the auxiliary location $edges$ all nodes adjacent to n_m with the associated cost of the edge. The program proceeds by looping among these edges and updating the distances of all nodes adjacent to n_m, in ver', accordingly. Finally, the $dijkstra$ module is called again but this giving as input the auxiliary location ver', as the remaining nodes are now located there.

9. Complexity Analysis

The strong adequacy obtained for the encoding of BAG does not provide, alone, the means to analyze the complexity of algorithms, but only ensures that any logic interpreter that searches for focused proofs by decomposing synthetic connectives will construct objects that correspond to computation runs of BAG programs. We must also enter into implementation details of the interpreter. We now briefly propose an implementation that can, in many situations, compute in *constant time* if a synthetic connective can be used to help prove a given sequent. In particular, it is easy to show that it takes constant time to build a focusing phase with the body of the **load**, **while**, and **if** clauses, since arithmetic operations and comparisons are assumed to be evaluated in constant time. Checking that the body of an alternative can be decomposed requires a search over all alternatives, which is bound by the size of the program, again a constant. The more interesting case involves determining if the body of an **unload** clause can be used since this clause involves pattern matching. In order to do pattern matching in constant time, we shall restrict tuples to be at most to arity 2. In that case, we represent the contents of such binary locations by using three linked hash-tables: one for when the pattern matching is on the first element; another hash-table when the pattern matching is on the second element; and finally the third hash-table is used when the pattern matching is on both elements. Hence, pattern matching is reduced to simple hash-table look-ups. Notice that one could do, in a similar fashion, constant time pattern matching even if tuples had arity greater than two: however that would come with a high cost in space.

Many algorithms, such as those described in Section 8, do not need to backtrack since all of their computation runs yield the same output. In the case of Dijkstra's algorithm, all of its computation runs end and has the same final output: namely, the multiset containing the shortest distances. For these algorithms, we can use an interpreter that picks among several possible synthetic connectives and does not backtrack. Since decomposing a synthetic connective can take constant time, we can infer the complexity of an algorithm by counting the number of decide rules (the num-

```
bipartite = λcolλverλansλprog.          //col - location with the colors of the nodes;
                                        //ver – location with the graph's unvisited vertices;
                                        //ans – output location with the answer yes or no.
new pr; new edges                       //create auxiliary locations.
loop1 ver λnλlcont                      //set node colors to gray.
  load ⟨n⟩ ver; load ⟨n, gray⟩ col lcont
loop1 ver λnλlcont1                     //pick a vertex, n, from a new component of the graph.
  unload0 col ⟨n, gray⟩                 //n must be gray.
    load ⟨n, blue⟩ col; load ⟨n⟩ pr     //set n's color as blue, and store it in pr.
    loop1 pr λsλlcont2                  //unload a vertex, s, that is in the same component.
      getEdges s edges                  //loads the edges connected to s in the location edges.
      loop2 edges λsλadjλlcont3         //loop over the neighbors of s.
        unload0 col ⟨s, blue⟩; load ⟨s, blue⟩ col   //if the color of s is blue.
          unload0 col ⟨adj, red⟩        //and if the neighbor of s is red
            load ⟨adj, red⟩ col lcont3  //proceed.
          [] unload0 col ⟨adj, blue⟩    //if the neighbor of s is blue.
            load ⟨no⟩ ans prog          //graph not bipartite.
          [] unload0 col ⟨adj, gray⟩    //if the neighbor of s is gray,
              unload0 ver ⟨adj⟩         //then it has not been yet visited, hence
                load ⟨adj, red⟩ col (load ⟨adj⟩ pr lcont3)   //assign it with the color red.
        [] unload0 col ⟨s, red⟩         //similar to the first alternative.
      lcont2
    lcont1
load ⟨yes⟩ ans prog                     //all nodes visited, hence the graph is bipartite.
```

Figure 6. Bipartite graph checking $\mathcal{P}^{\mathcal{G}}_{bp}$

ber of synthetic connectives) in a derivation that witnesses a complete computation run of an algorithm. For example, any derivation obtained from ($\mathcal{P}^{\mathcal{G}}_{bp}$ *col ver ans* **end**) where *ver* contains the nodes of the graph and all other locations are empty, contains $\mathcal{O}(|N| + |E|)$ decide rules, where $|N|$ and $|E|$ are the number of nodes and edges in a graph. Nodes are used at most three times and edge are used at most four times. Hence, the complexity of $\mathcal{P}^{\mathcal{G}}_{bp}$ is $\mathcal{O}(|N|+|E|)$. Notice that as we are using simple data structures to store the nodes and edges of a graph, the Dijkstra's algorithm presented in Figure 7 has linear complexity on the size of the graph. One could, however, implement more complicated data structures in BAG, such as Fibonacci heaps, and obtain better complexity results for this algorithm.

10. Related Work

Various proposals for describing algorithms via rewriting multisets have been developed in the past. Probably one the earliest such proposals is the Gamma programming language [3] although the even older specification language of Petri nets is also closely related to multiset rewriting. The Linda coordination model [9] also makes use of primitive operations similar to those used in the manipulation of multisets. The close relationship between multiset-based computation and linear logic has been known and exploited for many years within early linear logic programming languages such as LinLog [1], Lolli/Forum [17], MSR [4], and Lollimon [13].

It is often difficult to directly relate the *search* for proofs (say, in a logic programming setting) to performing computations in a step-by-step, algorithmic sense. Probably the largest single problem in making this connection is the need to do backtracking during the search for proofs. Such backtracking might be acceptable if it can contained within "internal" and invisible processing steps, but it is unacceptable if such backtracking is done between "visible" steps, such inputting and outputting. In this paper, we tried to group possible backtracking points that are to be internal into single, macro-level inference steps: other non-deterministic choices are then left to the algorithm developer to organize appropriately.

Another approach to the treatment of backtracking is more global. Proof search can be organized around forward chaining. If one saturates a set of forward chaining rules with all possible consequences of a set of formulas, then failure to prove some atomic goal with respect to that saturation does not lead to backtracking. If some forward chaining is used but saturation is not done, then the failure to prove an atomic formula might be due to its not being provable or to not having accumulated this particular consequence yet: in the later case, one would need to backtrack and attempt to add more consequences. Saturation has been used in both

```
dijkstra = λver λdist λprog.
new ver'; new min; new edges                    //create auxiliary locations
if (is_empty ver) prog                          //finish if there are no more nodes to traverse
[] extractMin ver ver' min                      //otherwise, call the extractMin module.
unload_2 min ⟨n_m, c_m⟩; load ⟨n_m, c_m⟩ dist    //unload the minimum node, n_m.
  getEdges n_m edges                            //get the edges connected to n_m.
  loop_2 edges λadj λd λlcont                   //update the distances of n_m's neighbors, adj.
    unload_1 dist ⟨adj, c⟩                      //either, the shortest distance to adj is already computed.
      load ⟨adj, c⟩ dist lcont                  //proceed.
    [] unload_1 ver' ⟨adj, c⟩                   //otherwise, check if there is a shorter path to adj.
      if (c ≤ d + c_m) (load ⟨adj, c⟩ ver' lcont)
      [] if (c > d + c_m) (load ⟨adj, d + c_m⟩ ver' lcont)
  dijkstra ver' dist prog                       //call the dijkstra module.

main = λnodes λdist λsrc λprog.                 //nodes – location with the graph's nodes;
                                                //dist – location with the shortest distances.
                                                //src – name of the source node.
new ver                                         //create auxiliary location
loop_1 nodes λn λlcont                          //set the distance of all nodes to ∞, except the source node.
  if (n ≠ src) (load ⟨n, ∞⟩ ver lcont)
  []
  if (n = src) (load ⟨s, 0⟩ ver lcont)
dijkstra ver dist prog                          //call the dijkstra module.
```

Figure 7. Dijkstra's algorithm $\mathcal{P}^{\mathcal{G}}_{dj}$.

the Gamma and the Lollimon setting as means for dispelling backtracking. We have not pursued this approach here since we know of no proof theoretic treatment of saturation.

McAllester & Ganzinger [15, 8, 7] developed a style of algorithm specification, called "logical algorithms," that was inspired by bottom-up, logic programming specifications. In order to account for more algorithms, they moved beyond logic in order to incorporate the deletion of atomic formulas and the assignment of priorities to inference rule application. Their framework was able to specify algorithms that efficiently solved problems from domains such as graph theory (*e.g.*, bipartite checking and the shortest distance problem), efficient data structures (*e.g.*, the Union/Find algorithm), and polymorphic type inference [14]. Simmons & Pfenning [20] revisited this style of logic specification and used linear logic inspired proof search to provide a sound foundation for the deletion of atomic facts.

Both the approaches by McAllester & Ganzinger and Simmons & Pfenning use a bottom-up, generative interpreter that relies on saturation to control the scope of backtracking. By a careful and, at times, complex analysis of that particular interpreter, it is possible to guarantee efficient implementations for the specified logic programs.

There are two essential differences between our work and that on "logical algorithms." First, we have remained within logic and proof theory. We have asked for algorithms to be accounted for using logic in both a sound and complete fashion. In fact, we have asked for more: we have insisted that the focused proofs that are built within that logic are in one-to-one correspondence with the steps of a simple algorithmic specification language. Second, we have not introduced the notion of an interpreter that directs search: in the "logical algorithm" papers, an algorithm's description is split between the logic specification *and* the interpreter. In our setting, the steps taken by an algorithm are matched precisely to focused inference rules. The only explicit role of an interpreter in our work here is in the determination as to whether or not inference rules can be efficiently implemented in, say, constant time.

11. Conclusions

In this paper, we show that a wide range of algorithms can be specified in the linear logic system with subexponentials called *SELL*. In order to better illustrate the algorithmic power of *SELL*, we propose some very simple extensions, such as, *definitions* and new connectives that allow creating new locations. Then, we describe how to use subexponentials to locate data, and propose a programming language, called BAG, containing loops, conditionals, and operations that insert and delete elements from subexponentials. Finally, we give a proof theoretic semantics for BAG in such a way that there is a one-to-one correspondence between the set of (partial) computation runs of an intended semantics and the set of (open) focused derivations. We also discuss

that, by using different focusing annotations to change the size of synthetic connectives, we can capture different intended operational semantics. At the end, we illustrate the power of *SELL* by encoding some complicated algorithms, such as Dijkstra's shortest path algorithm and an algorithm for checking if a graph is bipartite.

Clearly, one can use subexponentials to capture more computational behaviors. We have not yet used locations that contain sublocations. One could imagine a more complicated use of the subexponential pre-order where some locations are inside other locations. In this case, the test for emptiness of a super-location would succeed only if all of its sublocations are also empty.

We have restricted our attention to single-threaded algorithms. One might consider specifying more general, concurrent algorithms by extending *SELLF* to allow *multifocusing* [19]. The multifocus inference rule allows a synthetic connective to be build from focusing on more than one formula. A transition step attached to such an inference rule could model, say, the independent application of algorithmic steps in separate threads.

Throughout this paper, we assumed a global polarity assignment where all atoms are assigned negative polarity. Although different polarity assignments do not affect provability, they can affect considerably the shape of the focused proofs obtained. In [12], Liang & Miller show that if more flexible polarity assignments are used, one can mix forward and backward-chaining behaviors. This observation was used in [11, 18], to specify the computational behaviors of constraint systems and of tabled deduction. One could investigate what different types of algorithm specifications can be captured by using different polarity assignments in *SELLF*.

Acknowledgments This work has been supported in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[2] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, eds., *Intern. Conference on Logic for Programming and Automated Reasoning (LPAR)*, *LNCS* 4790, pp. 92–106, 2007.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model: ten years after. In *Coordination programming: mechanisms, models and semantics*, pp. 3–41. World Scientific Publishing, IC Press, 1996.

[4] Iliano Cervesato. Typed MSR: Syntax and examples. In *MMMACNS: Intern. Workshop on Methods, Models and Architectures for Network Security*, *LNCS* 2052, pages 159–177. Springer, 2001.

[5] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In G. Gottlob, A. Leitsch, and D. Mundici, eds., *Kurt Gödel Colloquium*, *LNCS* 713, pp. 159–171. Springer, 1993.

[6] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] H. Ganzinger and D. McAllester. Logical algorithms. In *Proc. ICLP 2002*, *LNCS* 2401, pp. 209–223. Springer, 2002.

[8] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In R. Goré, A. Leitsch, and T. Nipkow, eds., *Automated Reasoning, First Intern. Joint Conference (IJCAR)*, *LNCS* 2083, pp. 514–528. Springer, 2001.

[9] David Gelenter. Generative communication in Linda. *ACM Trans. on Prog. Lang. and Systems*, 7(1):80–112, 1986.

[10] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143, 1998.

[11] Radha Jagadeesan, Gopalan Nadathur, and Vijay Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In *FSTTCS*, *LNCS* 3821, pp. 517–528, Hyderabad, 2005. Springer.

[12] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, eds., *CSL 2007 LNCS* 4646, pp. 451–465. Springer, 2007.

[13] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In P. Barahona and A. Felty, eds., *Intern. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pp. 35–46, 2005.

[14] D. McAllester. A logical algorithm for ML type inference. In R. Nieuwenhuis, ed., *Rewriting Techniques and Applications, 14th Intern. Conference, RTA-03*, *LNCS* 2706, pp. 436–451, Valencia, Spain, 2003. Springer.

[15] David A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

[16] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[17] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[18] Dale Miller and Vivek Nigam. Incorporating tables into proofs. In J. Duparc and T. A. Henzinger, eds., *Computer Science Logic*, *LNCS* 4646, pp. 466–480. Springer, 2007.

[19] Dale Miller and Alexis Saurin. From proofs to focused proofs: a modular proof of focalization in linear logic. In J. Duparc and T. A. Henzinger, eds., *Computer Science Logic*, *LNCS* 4646, pp. 405–419. Springer, 2007.

[20] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, eds., *ICALP: Intern. Colloquium Automata, Languages and Programming, Reykjavik, Iceland*, *LNCS* 5126, pp. 336–347. Springer, July 2008.