CHECKING POLYNOMIAL TIME COMPLEXITY WITH TYPES

Patrick Baillot *
Laboratoire d'Informatique de Paris-Nord (UMR 7030 CNRS)
Institut Galilée, 99 av. J.-B. Clément, 93430 Villetaneuse, France.
pb@lipn.univ-paris13.fr

Abstract Light Affine Logic (LAL) is a logical system due to Girard and Asperti offering a polynomial time cut-elimination procedure. It can be used as a type system for lambda-calculus, ensuring a well-typed program has a polynomial time bound on any input. Types use modalities meant to control duplication.

We consider parameterized types where parameters are on the number of modalities and the *type instantiation* problem: given a term and a parameterized type, does there exist a valuation of the parameters such that the term admits the corresponding type? We show that this type instantiation problem is decidable for normal terms.

1. Introduction:

Several authors have proposed these last years programming languages and calculi with intrinsic complexity property, for instance languages ensuring that all functions representable are polytime without refering to an explicit time measure (see for instance Hofmann, 2000, Leivant and Marion, 1993).

Light linear logic (Girard, 1998) is one of these systems: it is based on the proofs-as-programs approach to computation. Polynomial running time is ensured in this framework by control of duplication expressed by means of modalities in types. The system was later simplified by Asperti into Light *Affine* Logic (LAL) in Asperti, 1998.

LAL can be seen as a programming language using Curry-Howard isomorphism, and indeed Roversi developed a syntax and a type inference procedure (Roversi, 2000). However this approach requires the user to provide key information about the structure of the program (*boxes*) ensuring the time bound.

*This work was partly done while the author was at Laboratoire d'Informatique de Marseille, Université Aix-Marseille II, Marseille.

Placing boxes makes programming difficult. We believe that it is worth trying to automatize this task and so we relegate the handling of boxes to the level of the type system. A motivation for that is the perspective of designing a language where control over time complexity would be done in a way transparent to the user: he would use a regular functional language and the compiler would in case of successful typing guarantee a bound for the running time on any input.

LAL types essentially differ from simple types by modalities. Recall that the idea of decorating intuitionistic proofs with linear logic modalities has been extensively studied by Schellinx *et al.* (Danos et al., 1994; Schellinx, 1994). Here we are reconsidering this idea in the more constraining setting of LAL. A related work has been done by Coppola and Martini (Coppola and Martini, 2001) who gave a type-inference procedure for lambda-terms in Elementary Affine Logic, a system which corresponds to elementary complexity.

In this paper we carry out a first step in this direction for LAL. We consider as source language lambda-calculus and introduce an intermediary typing syntax where boxes are not precised explicitly but can be recovered. Then we show how this typing can be extended with parameters on the number of modalities (parameterized graph terms). Finally we establish the following *type instantiation theorem*: given a term in normal form and a parameterized type does there exist an instance of the parameters making it a valid type for the term?

Due to space constraints proofs are omitted in this paper. The reader can refer to Baillot, 2001 where some examples are also given.

2. Light affine logic and lambda-term typing:

2.1. Typing with sequent calculus:

We use Light Affine Logic sequent calculus as a type derivation system for lambda-calculus. Lambda-calculus terms are given by: $t ::= x \mid \lambda x\, t \mid (t)t$.

The LAL types are obtained by the following grammar (we do not consider second-order quantifiers): $T ::= \alpha | \, T \multimap T | \, !T | \, \S T$.

Modalities ! and § are called *exponentials*. We write $!^n T$ for $! \cdots ! T$ with n repetitions of !. The typing rules are given on figure 1. In the paragraph promotion rule (§ prom.) we can have $n = 0$ and each $\Box_i$ is either § or !. As we deal only with terms in normal form we will not be using the *cut* rule.

For instance, types for tally integers and binary lists are given respectively by: $N_A^{LAL} = !(A \multimap A) \multimap \S(A \multimap A)$,

$Bin_A^{LAL} = !(A \multimap A) \multimap !(A \multimap A) \multimap \S(A \multimap A)$.

These are instances depending on A, as we are in a quantifier-free setting.

A variant of this system, Elementary Affine Logic (EAL), corresponds to elementary time. EAL types are obtained by: $T ::= \alpha | \, T \multimap T | \, !T$.

As to the typing rules, EAL typing is obtained by replacing (! prom. 1), (! prom. 2) and (§ prom) by a single rule:

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : A}{x_1 : !A_1, \ldots, x_n : !A_n \vdash t : !A} \text{ (EAL prom.)}$$

$$\frac{}{x : A \vdash x : A}\ \text{(variable)} \qquad \frac{\Gamma \vdash t : A \quad \Delta, x : A \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}\ \text{(Cut)}$$

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \text{(weak.)}$$

$$\frac{x_1 : !A, x_2 : !A, \Gamma \vdash t : B}{x : !A, \Gamma \vdash t[x/x_1, x/x_2] : B}\ \text{(cont.)} \qquad \frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : B}{x_1 : \Box_1 A_1, \ldots, x_n : \Box_n A_n \vdash t : \S B}\ \text{(§ prom.)}$$

$$\frac{x : A \vdash t : B}{x : !A \vdash t : !B}\ \text{(! prom. 1)} \qquad \frac{\vdash t : B}{\vdash t : !B}\ \text{(! prom. 2)}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{(right arrow)} \qquad \frac{\Gamma, x : B \vdash t : C \quad \Delta \vdash u : A}{\Gamma, \Delta, y : A \multimap B \vdash t[(y)u/x] : C}\ \text{(left arrow)}$$

Figure 1. LAL sequent typing

Proposition 1 *If* $\vdash_{LAL} t : A \multimap B$ *then there exists a polynomial* P *such that: for any* u *such that* $\vdash_{LAL} u : A$*, the term* $(t)u$ *can be evaluated in* $P(|u|)$ *steps, where* $|u|$ *denotes the size of* u*.*

The evaluation mentioned in the result is performed by a graph-rewriting method (normalisation of proof-nets). Similarly, typing in EAL ensures an elementary recursive time bound.

With polymorphic types (LAL_2), a completeness result also holds (Roversi, 1999; Girard, 1998): for any polytime function f on binary lists there exists a term t representing f such that t is typable in LAL_2.

2.2. Translations:

One can see EAL and LAL types as refinements of simple types, adding some intensional information about which arguments can be duplicated and which ones are used linearly. To make this remark explicit, let us recall the natural forgetful functors from EAL to simple types (intuitionnistic logic IL) and from LAL to EAL: $[.]_0 : EAL \to IL$ and $[.]_1 : LAL \to EAL$. Their action on formulas is defined by:

$[\alpha]_0 = \alpha \quad [A \multimap B]_0 = [A]_0 \to [B]_0 \quad [!A]_0 = [A]_0$

$[\alpha]_1 = \alpha \quad [A \multimap B]_1 = [A]_1 \multimap [B]_1 \quad [!A]_1 = [\S A]_1 = ![A]_1$

Define $[.] = [.]_0 \circ [.]_1 : LAL \to IL$. These translations extend to sequent calculus proofs, hence to type derivations. So under each LAL/EAL type, a simple type is present... but while simple types only ensure termination, EAL/LAL types ensure termination with a time complexity bound.

It comes as no surprise that a LAL type can be weakened into an EAL type, as a polynomial bound surely yields an elementary bound, but in fact we can also give a translation $[.]_2 : EAL \to LAL$. For that we define by mutual induction two translations $(.)^p, (.)^n : EAL \to LAL$:

$(\alpha)^p = (\alpha)^n = \alpha \quad (A \multimap B)^p = (A)^n \multimap (B)^p \quad (!A)^p = \S(A)^p$

$(A \multimap B)^n = (A)^p \multimap (B)^n \quad (!A)^n = !(A)^n$

Then $[A]_2 = (A)^p$ and $[\Gamma \vdash A]_2 = (\Gamma)^n \vdash (A)^p$. So an EAL ! is transformed into a LAL ! if in negative position, and in § if in positive position. This gives a translation from EAL cut-free derivations into LAL (cut-free) derivations. Thus EAL typed normal terms are typable in LAL ... However, this translation is not compositional, and so this does not give nonsense on the complexity side.

In other words, we might in some cases be able to type a term t in LAL, and then be unable –because of its type – to apply it to any relevant argument ! For instance if $\vdash_{EAL} t : T$ with $T = N_\alpha^{EAL} \multimap N_\alpha^{EAL}$ we get in LAL a type $[T]_2 = (\S(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)) \multimap N_\alpha^{LAL}$. Thus with this typing t cannot be applied to a LAL integer ... So in general given a term t we will not be searching for *any* type, but for a type satisfying certain constraints allowing for a suitable use of t.

It should now be clear that the difficulty of typing a term in EAL/LAL lies in the problem of determining where to place modalities in the type, and *how many* modalities are needed. Given a term, there is no obvious direct way to bound the number of modalities needed for typing it.

2.3. Typing with dags:

Graph terms. Typing with sequent calculus is uneasy. However we cannot simply type lambda-terms using their syntactic tree, as explicit information about sharing of subterms as given by sequent calculus proofs is important. Therefore we will use directed acyclic graphs (*dags*) that we call *graph terms*. A graph term has one input (or root node) and a certain number of outputs (or variable nodes). On the figures, the implicit direction of edges is from top to bottom. There are four other nodes: λ-node (one premise and one conclusion), @-node (one premise and two conclusions), c-node (contraction: two premises and one conclusion), ax-node (one premise and one conclusion). The ax-node corresponds to sequent calculus axiom, and we will see its utility when we type the graphs.

A variable node can be *free* or *bound*, in which case we add a special edge (a *pointer*) from the variable node to the corresponding λ-node (there is no pointer to the λ-node if its variable is not used in the term).

Graph terms are defined inductively with these nodes according to the grammar on figure 2 (note that we do not represent root and variable nodes). Observe that in graph terms axiom nodes only appear (possibly preceded by a bunch of λ nodes) at the root or in the right conclusion of a @. The left conclusion of a @ can be followed by another @, a contraction or a variable.

A *contraction tree of the graph* is a maximal subgraph whose internal nodes are only c-nodes.

A *path* in a graph term is an oriented sequence of adjacent edges. We write $p = e_1 \dots e_n$ if p is obtained by concatenation of edges e_i. A *complete path* is a path going from the root node to a variable node. A path p is *prefix* of a path q (denoted $p \prec q$) if q is obtained by p followed by another path r (possibly empty, i.e. $p = q$).

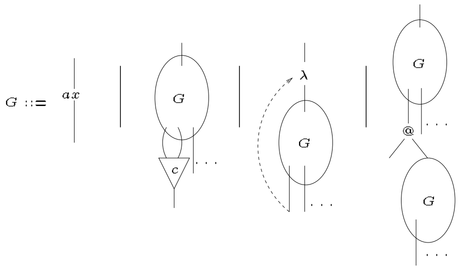

Figure 2. Graph terms inductive definition.

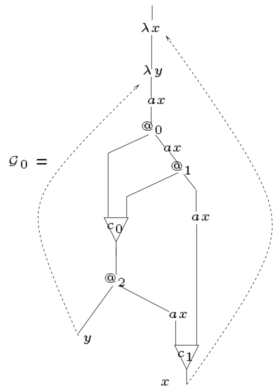

Figure 3. A graph term for $\lambda x \lambda y (y) x ((y) x) x$

Given a graph term, we denote by $\leq$ the partial order given by the dag ($N \leq N'$ if N' is above N in the figure drawing of the graph term).

To each graph term we can associate a normal lambda-term in a natural inductive way (see figure 3). This mapping is not one-one though, as a lambda-term can allow various sharings of its subterms.

Now, given a graph term $\mathcal{G}$ and a contraction c of $\mathcal{G}$, we would like to be able to determine which is the minimal sub-graph term of $\mathcal{G}$ on which this contraction could be done. We define the *junction* of contraction c (denoted $jn(c)$) as the minimal node common to all complete paths containing c and strictly superior to c (for instance on fig.3, $jn(c_0) = @_0$ and $jn(c_1) = @_1$). This definition makes sense as: all these paths have at least one node in common (the root-node), and given two nodes common to all these paths they must be comparable. Note that a junction node is necessarily a @-node. Observe that we have:

Lemma 2 *if $\mathcal{G}$ is a graph term and x is a variable node bound by a λ node λ_0, then any complete path of $\mathcal{G}$ containing x also contains λ_0.*

Proposition 3

1. *If a variable node is inferior to a contraction c and this variable is bound, then the corresponding λ node is inferior to c or superior to $jn(c)$.*
2. *Given two contraction nodes c_1 and c_2, if $c_1 \leq c_2$ then either $jn(c_1) \leq c_2$ or $jn(c_1) \geq jn(c_2)$.*

Derivation graphs. We now want to define graph terms corresponding to LAL sequent calculus type derivations. For that we need to label edges with LAL types and introduce *boxes*. A box of a graph term $\mathcal{G}$ is a subgraph $\mathcal{G}'$ which is itself a graph term. The *input* of the box is the input edge of $\mathcal{G}'$ and its *outputs* are the premises of the free variable nodes of $\mathcal{G}'$. There are two kinds of boxes, corresponding respectively to the ! and § promotion rules of the sequent calculus: !-boxes and §-boxes.

LAL *derivation graphs* are defined inductively following the sequent calculus rules (figure 4); they are a subclass of *proof-nets* (Girard, 1998): nodes λ and @ correspond respectively to $\wp$ and $\otimes$.

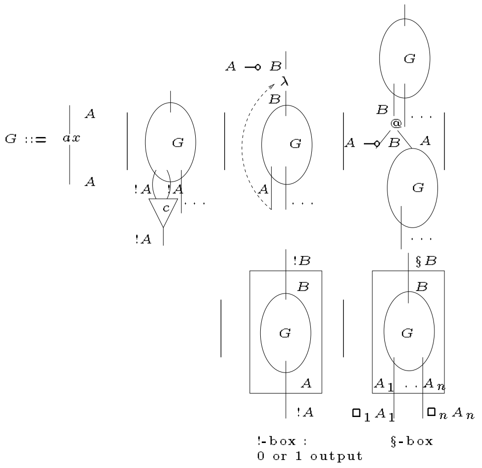

Figure 4. Derivation graphs inductive definition.

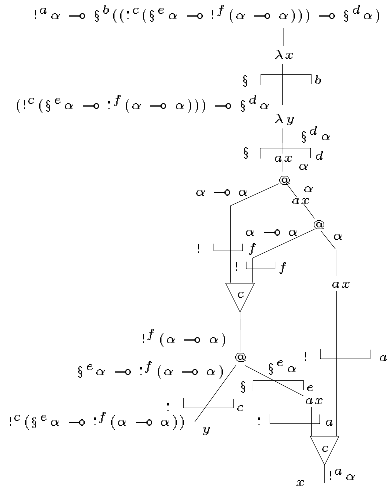

Figure 5. An example of a parameterized graph.

Note that in a derivation graph two distinct boxes either are disjoint or one is included in the other. Without loss of generality we restrict ax nodes to formulas A either atomic or of the form $B \multimap C$, as arbitrary axioms can be simulated with extra enclosing boxes. We keep this convention for the rest of the paper.

Let us call a *door* the crossing of a box by an edge; we say that the door is *opening* (resp. *closing*) if the edge enters (resp. exits) the box and we distinguish between ! doors and § doors.

Proposition 4 *In a LAL derivation graphs: root edge, right conclusion of @-node and conclusion of λ-node can have only opening doors, whereas conclusion of ax-node or c-node and left conclusion of @-node can have only closing doors. We say that these two categories of edges are respectively in* opening *and* closing *mode.*

Typed graphs. Our typing problem will be to turn a graph term into a derivation graph. For that we introduce an intermediary syntax, which forgets about the synchronization feature of boxes.

We consider doors as nodes with one premise and one conclusion and typing: a ! (resp. §) opening door has a premise $!A$ (resp. $\S A$) and a conclusion A; a ! (resp. §) closing door has a premise A and a conclusion $!A$ (resp. $\S A$).

Now, *LAL typed graph terms* are graphs built from the previous λ, @, c, ax and door nodes, with edges labeled by LAL formulas according to the typing conditions illustrated in figure 4 and such that: if we erase the doors and the labels we obtain a graph term, and only edges in opening mode (resp. closing mode) can have opening doors (resp. closing doors).

We introduce a function el from paths of a typed graph to $\mathbb{Z}$. It measures the *elevation* between the starting and the ending point of the path. Given a path p, $el(p)$ is defined in the following way: if p is an edge and n is the number of its doors, then if these are opening doors $el(p) = n$ and if they are closing doors $el(p) = -n$; otherwise if $p = e_1 \cdot e_2 \cdots e_k$, then $el(p) = \sum_{i=1}^{k} el(e_i)$.

One can obtain a LAL typed graph from a derivation graph by replacing each box by (disconnected) opening and closing doors: say a typed graph is *valid* if it can be obtained this way from a certain derivation graph.

Conversely, given a LAL typed graph we will study under which conditions we can associate opening and closing doors to define boxes so as to get a derivation graph. First let us examine the necessary conditions satisfied by valid typed graph terms.

Consider a path p in a typed graph term: we say it is *well-bracketed* if for any prefix q of p the number of closing doors in q is inferior or equal to the number of opening doors. Thus, each closing door can be matched in p with an opening door in the expected way (note that we do not require that each opening door is matched).

Now, the fact that boxes are disjoint or included one in the other ensures that valid typed graphs satisfy:

- **(C1) Bracketing condition**: any complete path p of $\mathcal{G}$ is well-bracketed.

  Second, if we consider two nodes N and N' and p, q two paths from N to N', they must cross the same boxes. Therefore we have:

- **(C2) Level condition**: given two nodes N, N' in $\mathcal{G}$ for any paths p, q from N to N' we have: $el(p) = el(q)$.

The level condition guarantees that the matching of closing doors with opening doors does not depend on the path chosen. Therefore, if a typed graph term satisfies the bracketing and level condition, then there is a unique way to associate doors in it to define boxes. Once boxes have been defined on a typed graph, we can examine further conditions:

- **(C3) Scope condition**: if x is a bound variable in $\mathcal{G}$, then the node corresponding to x and the binding λ-node belong to the same boxes.

- **(C4) Junction condition**: if a box contains a contraction node c, then it also contains the node $jn(c)$.

- **(C5) !-box conditions**: a box with opening !-door has no closing §-door and has at most one closing door.

We can check, using the inductive definition of derivations, that any valid typed graph satisfies these five conditions, but furthermore:

Theorem 5 (Synchronization) *A LAL typed graph term $\mathcal{G}$ is valid iff it satisfies conditions (C1) to (C5).*

The proof is given in Baillot, 2001.

3. Parameterized graphs

We want now to consider graph terms with a variable number of doors. This will be useful when we search for a valid type for a term. Therefore we will allow doors to be indexed by parameters, or even linear combination of parameters (on figures we write this index on the right-hand-side of the door).

We also have to change the types: intuitively, a LAL parameterized type is a type where exponentials can be indexed by variables, for instance: $\S^{n_1}(\S^3 !^{n_2}\alpha \multimap \S^{n_2}\alpha)$.

Given a set of integer variables $n, m \ldots$, *parameterized LAL types* are given by the grammar:

$$T ::= \alpha \,|\, T \multimap T \,|\, !T \,|\, \S T \,|\, \S^n T \,|\, !^n T$$

The *parameters* of a parameterized type T are the integer variables appearing in it. We denote their set by $\mathcal{V}(T)$.

The former mapping from LAL types to simple types is naturally extended to parameterized types. An *instance* of a parameterized type T is an application $\phi : \mathcal{V}(T) \to \mathbb{N}$. By substituting in T each integer variable n by $\phi(n)$ we get a LAL type $\phi(T)$.

Parameterized graph terms are defined as typed graphs except that doors are parameterized: for instance an opening ! door with parameter n has a premise $!^n A$ and a conclusion A. See figure 5 for an example.

The set of parameters of the graph $\mathcal{G}$ is denoted by $\mathcal{V}(\mathcal{G})$. From an instance $\phi : \mathcal{V}(\mathcal{G}) \to \mathbb{N}$ we can define a typed graph $\phi(\mathcal{G})$ as expected.

The elevation function el is defined for a parameterized typed graph as before but its values are linear combinations over $\mathcal{V}(\mathcal{G})$ with integer coefficients.

We consider the following problem (synchronization of parameterized graph):

Problem 1 given a parameterized graph term $\mathcal{G}$, does there exist an instance $\phi : \mathcal{V}(\mathcal{G}) \to \mathbb{N}$ such that $\phi(\mathcal{G})$ is a valid typed graph?

We define *constraints* on a parameterized graph $\mathcal{G}$ as first-order arithmetic formulas over $\mathcal{V}(\mathcal{G})$ built from: (i) linear inequations, (ii) conjunction $\wedge$, disjonction $\vee$ and universal quantification $\forall$.

An instance ϕ of a parameterized graph $\mathcal{G}$ *satisfies* a constraint $\mathcal{C}$ if $\mathcal{C}$ evaluates to true when we replace each n by $\phi(n)$. We will express the conditions of theorem 5 by constraints.

(C1) Bracketing. Each complete path of the graph is well bracketed. This is expressed by: for any path p starting from the initial edge $el(p) \geq 0$. This gives a finite conjunction of inequalities $\mathcal{C}_1$.

(C2) Level. For any pair of paths p, q with same origin and target, we have $el(p) = el(q)$. As there is a finite number of paths in the dag, this gives a finite conjunction of inequalities $\mathcal{C}_2$.

(C3) Scope. We must express the scope condition for each bound variable. Take a bound variable x and the λ node N_0 in the graph corresponding to its abstraction. Each path p from N_0 to the variable edge of x corresponds to an occurrence of the variable. For each of these we should have: (1) each closing door in the path is associated to an opening door in the path, ie the path p is well-bracketed; this is expressed by: for all $q \prec p$, $el(q) \geq 0$, (2) all boxes opened along the path are closed along the path; this is expressed by: $el(p) = 0$.

These two conditions stated for all relevant path p for all bound variables of the graph yield a constraint $\mathcal{C}_3$.

(C4) Junction. the problem of finding the junction of a given contraction in the graph is decidable, so we assume here that it has been done for all contractions of the graph. Now, given a contraction c and its junction j, we want to ensure that: each box containing c also contains j.

This is equivalent to the following condition: for any path p joining j to c, any opening door d_0 in p is associated to a closing door in p. We can express this condition by: for any suffix q of p, $el(q) \leq 0$.

Applied to each path from j to c, and so for all contractions of $\mathcal{G}$, this gives a constraint $\mathcal{C}_4$.

(C5) Bang boxes conditions. We want to express the conditions: (1) a !-box does not have any § exit door, (2) a !-box has at most one ! exit door.

We have:

Proposition 6 *Let d_0 and d_1 be two (possibly) parameterized doors of the graph respectively opening and closing, such that d_0 is above d_1. Denote by n and m their respective parameters and by p a path from d_0 to d_1, including d_0 and d_1. Given k with $1 \leq k \leq n$, one of the doors of d_1 matches the k-th door of d_0 iff the following predicate evaluates to true:*

$$P(d_0, d_1, k) = (\textit{for all } q \prec p, q \neq p \Rightarrow el(q) - k + 1 > 0) \wedge (el(p) - k + 1 \leq 0).$$

$P(d_0, d_1, k)$ is a finite conjunction of inequations.

Condition (1) can then be expressed in the following way: for any pair of parameterized opening !-door d_0 with parameter n and closing §-door d_1 such that the first one is above the second one in the dag the following should hold:

$$\forall k \leq n,\ \neg P(d_0, d_1, k).$$

Condition (1) is thus expressed by a constraint $\mathcal{C}_{51}$. Now, condition (2) is expressed in the following way:

for any d_0 parameterized opening !-door with parameter n and d_1, d_2 distinct closing !-doors such that d_0 is above d_1 and d_2 we have:

$$\forall k \leq n,\ \neg P(d_0, d_1) \vee \neg P(d_0, d_2).$$

Again, this yields a constraint $\mathcal{C}_{52}$. The !-boxes conditions are thus expressed by $\mathcal{C}_5 = \mathcal{C}_{51} \wedge \mathcal{C}_{52}$.

Solving. Let $\mathcal{C} = \wedge_{j=1}^{5} \mathcal{C}_j$. We have seen that $\mathcal{G}$ is a valid LAL derivation graph iff the constraint $\mathcal{C}$ is satisfied. Therefore an instance ϕ is a solution of the synchronization problem 1 for $\mathcal{G}$ iff it is a solution of $\mathcal{C}$. The problem of satisfiability for first-order formulas over linear inequations is decidable as these are part of Presburger arithmetic. Therefore we have:

Theorem 7 *Given a parameterized graph term $\mathcal{G}$, the problem of the existence of a valuation ϕ of $\mathcal{V}(\mathcal{G})$ such that $\phi(\mathcal{G})$ is a valid typed graph is decidable.*

Actually it is clear from the proof that we can even require the variables of $\mathcal{V}(\mathcal{G})$ to satisfy an initial constraint $\mathcal{C}_0$.

4. Type instantiation

Ideally, we would like starting from a lambda-term to decide whether it is typable in LAL. At the present we do not know whether this problem is decidable. It is the case if we consider a lambda-term in normal form. In general however we are not merely interested in obtaining a type for a normal lambda-term but we would like it to satisfy certain constraints. The most obvious requirement is that we want to be able to specify a data-type for arguments and result (recall the discussion in section 2.2). For instance:

does t admit a LAL type of the form $!^k N_A^{LAL} \multimap \S^l N_B^{LAL}$? where A, B are parameterized LAL types.

The compromise we adopt here is to ask the user to provide a parameterized type T, and to search whether there exists an instance of T which types t. In fact the user could also provide some conditions together with the parameterized type. For instance in the previous example he could require that $l \geq k$ holds.

Given a simple type T, the least constraining (and hence least informative) parameterized type over T is defined in the following way: decorate each positive (resp. negative) subformula of T with $\S^n$ (resp. $!^n$) where n is a fresh parameter. Observe that for checking typability in LAL for normal terms in a narrow-minded way such parameterized types suffice. Now we can state the problem we are considering:

Problem 2 (Type instantiation) *Given a (closed normal) lambda-term t, a LAL parameterized type T and a constraint $\mathcal{C}_0$ for T, does there exist an instance ϕ of T such that: $\phi(T)$ is a LAL type for t and ϕ satisfies $\mathcal{C}_0$?*

We now give the main lines of our decision procedure for problem 2. We start from a graph term $\mathcal{G}$ instead of a lambda-term, but since there is a finite number of graph terms corresponding to a lambda-term, this is not a problem for decidability. The dag specifies the information on sharing of subterms and order in which contractions are performed. We can also assume that $[T]$ is a valid simple type for T.

The algorithm will proceed in two phases: (i) *labeling phase*: from $\mathcal{G}$, T and $\mathcal{C}_0$ deduce all possible parameterized graphs with associated constraint $(\mathcal{G}', \mathcal{C}_1)$;

(ii) *solving phase*: apply theorem 7 to decide if one of these parameterized graphs admits an instance which makes it a valid typed graph.

Let us stress that the procedure we give has no pretention to efficiency, but aims at establishing decidability of problem 2 in a simple way.

Assume given a closed graph term $\mathcal{G}$ and a parameterized type T. We will give the corresponding parameterized graphs. We proceed in two steps: (a)(*labeling visit*) first we attribute (parameterized) types and doors to all edges of the graph but those belonging to contraction trees; this is done through one visit of the graph and there is only one possible labeling; (b)(*contraction trees decoration*) then we place (closing) doors with parameters in the contraction trees and add the corresponding types; there will be a finite number of ways to do this, and this step will determine the number of variables of our constraints system. At the end we obtain several possible parameterized graph terms with constraints.

(a) Labeling visit: For this first task, we perform a depth-first leftmost visit of the dag; more precisely we alternate downwards and upwards trips according to the following strategy:

- downwards trip: start from the root-edge and go down, choosing the left conclusion whenever you meet a @-node, until reaching an edge already labeled, then switch to upwards mode;
- upwards trip: go up until either meeting a @-node whose right conclusion has not been visited yet, in which case go down its right conclusion, or meeting the root edge, in which case the procedure is over. When meeting the root of a contraction tree go up its leaf edge who has been visited last (at least one leaf has already been visited);

It is easy to check that by this strategy we do visit the whole graph (but the edges in the contraction trees) and that we go through each edge two times, first down and then up.

Now we must say how we do the labeling during this visit. We denote by s an arbitrary sequence of indexed modalities.

Downwards trip. The root edge is labeled by T. For each λ node we cross: if the type before the node is not of the form $A = !^{n_1}\S^{n_2} \dots !^{n_{2k-1}}\S^{n_{2k}}(B \multimap C)$ for some k (some n_i's can be constants, in particular 0), then the procedure fails. Otherwise for each $1 \leq i \leq 2k$ such that $n_i \neq 0$ put an opening door (either ! or §) with parameter n_i. Then: the type before the doors is A; the type below the doors is $B \multimap C$; the conclusion of λ is typed by C and the variable edge bound by the λ, if there is one, is typed by B.

We necessarily meet an axiom node (by construction of the graphs). Denote by sC the current formula, where C is either atomic or of the form $A \multimap B$ and s is a sequence of modalities. Put the opening doors corresponding to s and type the premise of the axiom by C.

We then proceed with the trip but do not label anymore until switching to upwards mode.

Upwards trip.

- If we arrive to the root of a contraction tree with type sA where $A = \alpha$ or $B \multimap C$ we go up the leaf who has been visited last (it is a left conclusion of @ or a conclusion of ax) and we type it with A.
- If we arrive to a @ node from the left conclusion: if the type is not of the form $s(A \multimap B)$, the procedure fails; if it is, put the closing doors corresponding to s, type the premise of @ by B and the right conclusion by A; then go down the right conclusion of @.
- If we arrive to a @ node from the right conclusion: then this edge has already been typed; we continue upwards.
- If we arrive to an axiom node with type sC where C is either atomic or of the form $A \multimap B$, put the closing doors corresponding to s below the axiom, type their premise with C and proceed upwards.

Lemma 8 *After the labeling visit, all edges but those in contraction trees have been typed.*

(b) Contraction trees decoration: At this point, for each contraction tree we have a type for its root-edge and a type for each of its leaves, but not for the intermediary edges. We give a non-deterministic method for finding all possible contraction tree decorations.

Note that the leaves of the trees are conclusion of ax nodes or left conclusion of @ nodes and their types are of the form α or $B \multimap C$.

Say $\mathcal{C}_1$ is initially an empty constraints system. We will gradually need to add inequations to $\mathcal{C}_1$.

We will first deal with each branch of the tree separately. Consider one of these branches and denote by sB_1 (resp. B_2) the type of the leaf (resp. of the root-edge), where s is a sequence of indexed modalities and B_i is a type which does not start with a modality.

If $[B_1] = [B_2]$, identification of B_1 and B_2 can be expressed by a constraint which we add to $\mathcal{C}_1$, otherwise typing fails.

We then need to place along the branch enough doors to introduce the sequence s. As in s the number of alternances between ! and § is finite there is a finite number of ways to place parameterized closing doors along the branch to match this sequence. We choose one possibility, with a new parameter variable for each door, and add to $\mathcal{C}_1$ the equations expressing the fact that the sequence of doors on this branch introduces s. Finally we also add to $\mathcal{C}_1$ inequations imposing that the formula before each contraction starts with a ! (so that contraction is valid).

We proceed similarly for each branch of the contraction tree. Then for each contraction node, the types of both premises should be identified; again this is expressed by constraints which we include in $\mathcal{C}_1$.

Applying this method to each contraction tree and calling $\mathcal{C}_1$ again the resulting constraints system, we have completed the labeling phase. So we end up with a finite number of possible parameterized graph terms and associated constraints. The initial graph $\mathcal{G}$ is typable with the parameterized type T iff one of these systems admits a valuation making it valid. This can be decided according to theorem 7, hence we conclude:

Theorem 9 *The type instantiation problem for normal lambda-terms (problem 2) is decidable.*

Acknowledgments

The author wishes to thank Roberto Amadio and Laurent Régnier for suggestions and encouragements, as well as Kazushige Terui for useful discussions. The results in this paper were first presented at the 'Linear' TMR network meeting held in Bertinoro in april 2001 and we wish to thank the organizers.

References

Asperti, A. (1998). Light affine logic. In *Proceedings LICS'98*. IEEE Computer Society Press.

Baillot, P. (2000). Stratified coherent spaces: a denotational semantics for light linear logic. LFCS Tech. report 0025, Univ. of Edinburgh. presented at ICC'00.

Baillot, P. (2001). Checking polynomial time complexity with types (extended version). Tech. report 2001-09, Laboratoire d'Informatique de Paris-Nord.

Bellantoni, S., Niggl, K.-H., and Schwichtenberg, H. (2000). Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3).

Coppola, P. and Martini, S. (2001). Typing lambda-terms in elementary logic with linear constraints. In *Proceedings TLCA'01*, volume 2044 of *LNCS*. Springer-Verlag.

Danos, V. and Joinet, J.-B. (1999). Linear logic and elementary time. First Workshop on Implicit Computational Complexity (ICC'99).

Danos, V., Joinet, J.-B., and Schellinx, H. (1994). On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33(6).

Girard, J.-Y. (1998). Light linear logic. *Information and Computation*, 143:175–204.

Hofmann, M. (2000). Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3).

Leivant, D. and Marion, J.-Y. (1993). Lambda-calculus characterisations of polytime. *Fundamenta Informaticae*, 19:167–184.

Roversi, L. (1999). A P-time completeness proof for light logics. In *Proceedings CSL'99*, volume 1683 of *LNCS*. Springer-Verlag.

Roversi, L. (2000). Light affine logic as a programming language: a first contribution. *International Journal of Foundations of Computer Science*, 11(1).

Schellinx, H. (1994). *The Noble Art of Linear Decorating.* ILLC Dissertation Series 1994-1, Institute for Language, Logic and Computation, University of Amsterdam.

Terui, K. (2001). Light Affine Lambda-calculus and polytime strong normalization. In *Proceedings LICS'01*. IEEE Computer Society Press.

Wadler, P. (1991). Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut.