Checking polynomial time complexity with types

Patrick Baillot *
Laboratoire d'Informatique de Paris-Nord (UMR 7030 CNRS)
Institut Galilée, Université Paris XIII
99 av. Jean-Baptiste Clément
93430 Villetaneuse, France
tel: (+33)1 49 40 40 67
fax: (+33)1 48 26 07 12
pb@lipn.univ-paris13.fr

Abstract

Light Affine Logic (LAL) is a logical system due to Girard and Asperti, derived from linear logic and 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.

*This work was done while the author was at LIF, Université Aix-Marseille II, Marseille.

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 [Hof00b], [LM93] and the survey [Hof00a]).

Light linear logic ([Gir98]) 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 [Asp98].

LAL can be seen as a programming language using Curry-Howard isomorphism, and indeed Roversi developed a syntax and a type inference procedure ([Rov00]). However this approach requires the user to provide key information about the structure of the program (*boxes*) ensuring the time bound.

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 first 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. A second one is to investigate techniques to analyze the concrete expressivity of LAL (in terms of algorithms) by comparing it to lambda-calculus.

Recall that the idea of decorating intuitionistic proofs with linear logic modalities has been extensively studied by Schellinx *et al* ([DJS94, Sch94]). Here we are reconsidering this idea in the more constraint setting of LAL. A related work has been done by Coppola and Martini ([CM01]) who gave a type-inference procedure for lambda-terms in Elementary Affine Logic, a system which corresponds to Kalmar 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 instantiation of the parameters making it a valid type for the term ?

Overview: In section 2: we define the type system with sequent calculus; we introduce the syntax we will use for lambda-terms, graph terms, and describe type derivation as well as (partial) typing of graph terms: these graphs do not have any boxes but *door* nodes which handle the modalities. Finally we give a theorem characterizing among partial typings those coming from valid type derivations.

In section 3 we introduce parameterized graphs which have parameters on their number of doors. We show that the problem whether there exists an instantiation of the parameters turning the graph into a valid derivation is decidable.

Finally in section 4 we show that given a term and a parameterized type, which is a type where the number of modalities is given by parameters, one can decide whether for some instantiation of the parameters the term admits the corresponding type.

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. We do not consider second-order quantifiers. LAL types are obtained by the following grammar:

$$T := \alpha \mid T \multimap T \mid !T \mid \S T$$

Modalities ! and § are called *exponentials*. We write $!^n T$ for $!\cdots!T$ with n repetitions of !. The typing rules are given on figure 1. As we deal only with terms in normal form we will not be using the *cut* rule.

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

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

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

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

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

Figure 1: LAL sequent typing

In the paragraph promotion rule (§ prom.) we can have $n = 0$ and each $\Box_i$ is either § or !.

For instance, types for tally integers and binary lists can be given respectively by:

$$\begin{aligned} N_A^{LAL} &= !(A \multimap A) \multimap \S(A \multimap A) \\ Bin_A^{LAL} &= !(A \multimap A) \multimap !(A \multimap A) \multimap \S(A \multimap A) \end{aligned}$$

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: $T := \alpha \mid T \multimap T \mid !T$.

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

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

Tally integers can be typed in EAL by $N_A^{EAL} = !(A \multimap A) \multimap !(A \multimap A)$.

The main result of [Gir98, Asp98] implies then that:

Proposition 1 *If $\vdash_{LAL} t : A \multimap B$ then there exists a polynomial P such that:*

for any u such that $\vdash_{LAL} u : A$, then $(t)u$ 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: there exists an integer h and an exponential tower function B of height h such that $(t)u$ can be evaluated in $B(|u|)$ steps.

With polymorphic types (LAL_2), a completeness result also holds ([Rov99, Gir98]): 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 way to understand EAL and LAL types is to see them as refinements of simple types, adding some intensional information about which arguments can be duplicated and which ones are used linearly. In this respect, LAL is a finer refinement than EAL, providing more information in the type.

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 $(.)^p, (.)^n : EAL \to LAL$:

$(\alpha)^p = (\alpha)^n = \alpha$

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

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

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. We will come back on this issue in section 4.

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 first

on working out *how many* modalities are needed. Given a term, there is no obvious direct way to bound the number of modalities needed for typing it, and indeed if there was one could think of deciding typability by an enumeration brutal method. In fact, in LAL the maximal nesting level of exponentials is linked to the degree of the polynomial obtained for the bound ([Gir98]).

2.3 Typing with dags:

2.3.1 Graph terms:

Given a lambda-term t, we could try to type it by writing down its syntactic tree and labeling its edges with LAL types according to a certain set of rules. However, proceeding this way we would not be able to type all terms typed with sequent calculus.

The problem is that sequent calculus proofs actually describe terms with explicit *sharing* of subterms. That is why we need a more accurate syntax to deal with terms.

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 axioms, 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.

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).

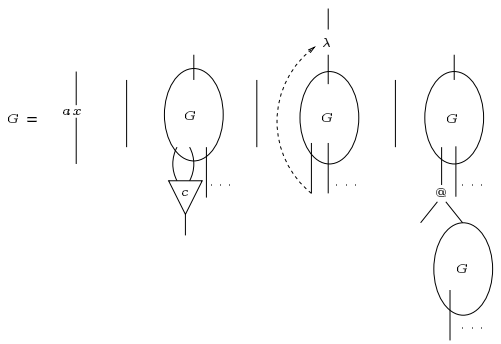

Figure 2: Graph-terms inductive definition

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 $p = e_1 \ldots e_n$ if p is obtained by concatenation of edges e_i. If N is a node of the graph term, a *path of* N is a path from the root-node to N. A *complete path*

is a path going from the root node to a variable node. A path p is *prefix* of a path q (note $p \prec q$) if q is obtained by p followed by another path r.

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

To each graph term we can associate a normal lambda-term in a natural inductive way. This mapping is not one-one though, as a lambda-term can allow various sharings of its subterms (see an example on figure 3 in appendix A with two graphs representing the same term). A simple graph term corresponding to a lambda-term is obtained by writing its syntactic tree and then choosing an explicit ordering of (binary) contractions for variables with more than one occurrence (as the graph $\mathcal{G}_1$ of our previous example)

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. This is the notion of *junction* node:

Definition 1 *The junction of contraction c (denoted $junction(c)$) is the minimal node common to all complete paths containing c and strictly superior to c.*

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, as this is the only node with more than one conclusion.

Intuitively the junction node corresponds to the minimal subterm on which the contraction can be performed. For an example see figure 3 again: for $\mathcal{G}_0$, $junction(c_0) = junction(c_1) = @_0$. 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.*

We then have the following properties:

Proposition 3 *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 $junction(c)$.*

Proposition 4 *Given two contraction nodes c_1 and c_2, if $c_1 \leq c_2$ then either $junction(c_1) \leq c_2$ or $junction(c_1) \geq junction(c_2)$.*

The proofs of these propositions can be found in appendix B.

2.3.2 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:

- !-box: if the input edge of $\mathcal{G}'$ has label (type) B, then outside the box it has label $!B$; there is at most one output with label $!A$ outside the box if A is the label of the output of $\mathcal{G}'$;

- §-box: if the input edge of $\mathcal{G}'$ has label (type) B, then outside the box it has label $\S B$; for each output of $\mathcal{G}'$ of type A, outside the box it can have type $!A$ or $\S A$.

LAL *derivation graphs* are then defined inductively following the sequent calculus rules (see figure 4 in appendix A); they are a subclass of *proof-nets* ([Gir98]): nodes λ and @ correspond respectively to $\wp$ and $\otimes$.

An important point is that in a derivation graphs two distinct boxes either are joint 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 5 *In a LAL derivation graphs we have:*

- *the following edges can have only opening doors: initial edge, right conclusion of @-node, conclusion of λ-node,*
- *the following edges can have only closing doors: conclusion of ax-node or c-node, left conclusion of @-node.*

We say that these two categories of edges are respectively in opening *and* closing mode*.*

Proof : This is easily proved by induction on the construction of derivation graphs. □

2.3.3 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 which have one premise and one conclusion and the following typing conditions:

- ! (resp. §) opening door: premise $!A$ (resp. $\S A$) and conclusion A,
- ! (resp. §) closing door: premise A and conclusion $!A$ (resp. $\S A$).

Now, *LAL typed graph terms* are graphs built from the previous λ, @, c and ax nodes and door nodes, with edges labeled by LAL formulas according to the previous conditions and such that:

- if we erase the doors and the labels we obtain a graph term,
- 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 $el(p) = -n$;
- 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 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 the following property:

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

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

Note that the formulation of the scope, junction and !-box conditions assumes boxes have already been defined. This is given by the bracketing condition.

The proof can be found in appendix B.

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 sequences of exponentials can be indexed by variables, for instance $\S^{n_1}(\S^3!^{n_2}\alpha \multimap \S^{n_2}\alpha)$. More precisely we allow variables only for certain exponentials: ! in positive position or § in negative position.

Definition 2 *Given a set of integer variables $n, m \ldots$, a* parameterized LAL type *is a P given by the grammars:*

$$P := \alpha \mid N \multimap P \mid !P \mid \S P \mid \S^n P$$

$$N := \alpha \mid P \multimap N \mid !N \mid !^n N \mid \S N$$

where n, m are any integer variables.

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

The former mapping from LAL types to simple types is naturally extended to parameterized types.

An *instance* of a parameterized T is an application $\phi : \mathcal{V}(T) \to \mathbb{N}$. By susbtituting in T each integer variable n by $\phi(n)$ we get a LAL type $\phi(T)$.

Now, a *parameterized graph term* $\mathcal{G}$ is a graph whose opening §-doors and closing !-doors can be indexed by parameters or linear combinations of parameters. As previously we consider typed parameterized graph terms with parameterized types as labels and satisfying the local typing conditions for each node. 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 the graph term $\phi(\mathcal{G})$ as expected.

Let us remark that for a parameterized typed graph the elevation function el is defined as before but its values are linear combinations over $\mathcal{V}$ with integer coefficients.

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

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

Let us say:

Definition 3 *A* constraint *on a parameterized type is a linear inequation or equation on its integer variables with integer coefficients.*

A constraints system is a boolean combination of constraints. Actually we will be dealing with conjunctions of disjunctions of constraints, and for initial constraints we are mainly interested in conjunctions.

An instance ϕ of a parameterized type T satisfies *a constraints system $\mathcal{C}$ if $\mathcal{C}$ evaluates to true when we replace each n by $\phi(n)$.*

We will express the conditions of theorem 6 by inequations.

(C1) Bracketing: Each complete path of the graph should be well bracketed. This is expressed in the following way:

$$\text{for any path } p \text{ starting from the initial edge } el(p) \geq 0.$$

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:

- 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 $q \prec p$, $el(q) \geq 0$,
- 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 conjunction of constraints $\mathcal{C}_3$.

(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 constraints system $\mathcal{C}_4$.

If we were dealing with type inference in EAL the list of constraints would now be over, as $\wedge_{i=1}^{4}\mathcal{C}_i$ would give the conditions for the graph to be an EAL derivation graph. For LAL however we still need to express the conditions on the !-boxes.

(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.

Remember that due to the initial requirement on the candidate type, there is a finite fixed number of opening !-doors and closing §-doors in the graph. We have:

Proposition 7 *Let d_0 and d_1 be two doors of the graph respectively opening and closing, such that d_0 is above d_1. Denote by p a path from d_0 to d_1, including d_1 but not d_0. The door d_1 closes the box opened by d_0 if and only if the following condition is satisfied:*

$$(\textit{for all } q \prec p, q \neq p \Rightarrow el(q) \geq 0) \textit{ and } el(p) < 0.$$

This gives a conjunction of inequations that we denote by $P(d_0, d_1)$.

Condition 1 can then be expressed in the following way: for any pair of opening !-door d_0 and closing §-door d_1 such that the first one is above the second one in the dag, $\neg P(d_0, d_1)$ is true. As $\neg P(d_0, d_1)$ is equivalent to a disjunction of inequations, the condition is thus expressed by a constraints system $\mathcal{C}_{51}$.

In order to translate condition 2 we need to be able to express the fact that an opening door is matched by a closing door belonging to a parameterized sequence of closing doors, which is done in a similar way.

Proposition 8 *Let d_0 and d_1 be respectively an opening and a parameterized closing door, such that d_0 is above d_1 in the dag. Let p denote the path from d_0 to d_1, including d_1 but not d_0.*

One of the doors of d_1 closes the box opened by d_0 if and only if condition $P(d_0, d_1)$ from proposition 7 is satisfied.

Now, condition 2 is expressed in the following way:

for any d_0, d_1, d_2 resp. opening !-door and distinct closing parameterized §-doors such that d_0 is above d_1 and d_2 we have:

$$\neg P(d_0, d_1) \vee \neg P(d_0, d_2).$$

Again, this yields a constraints system $\mathcal{C}_{52}$. 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 with type T satisfying $\mathcal{C}_0$ iff the constraints system $\mathcal{C}$ is satisfied.

Therefore an instance ϕ of T is a solution of the constraint decoration inference problem for $\mathcal{G}$, T, $\mathcal{C}_0$ iff it is a solution of $\mathcal{C}$. The problem of satisfiability for boolean combinations of linear inequations over integers is decidable (see for instance [Pap94]). Therefore we have:

Theorem 9 *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 valid is decidable.*

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

4 Type instantiation

4.1 Statement of the problem

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 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 constraints system $\mathcal{C}_0$ for T, does there exist an instance ϕ of T such that:*

- *$\phi(T)$ is a LAL type for t,*
- *ϕ satisfies $\mathcal{C}_0$?*

4.2 Type instantiation algorithm

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:

- *labeling phase*: from $\mathcal{G}$, T and $\mathcal{C}_0$ deduce all possible parameterized graphs and constraints system $(\mathcal{G}', \mathcal{C}_1)$,
- *solving phase*: apply theorem 9 to decide if one of these parameterized graphs admits a valuation 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.

4.2.1 Labeling phase:

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:

- (*labeling visit*) first we attribute (candidate) 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 labeling possible;

- (*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.

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 a variable edge, or a contraction tree whose conclusion has already been visited; in these cases, switch to upwards mode;
- upwards trip: go up until either:
  - meeting a @-node whose right conclusion has not been visited yet; in this case go down its right conclusion;
  - or meeting the root edge: the procedure is then 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. See an example on figure 5, appendix A.

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 initial edge is labeled by T.

For each λ node we cross: if the type before the node is not of the form $A = !^{n_1}\S^{n_2}\ldots!^{n_{2k-1}}\S^{n_{2k}}(B \multimap C)$ 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 $A \multimap B$ ans 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 reaching:

- either a variable node (call x the variable): then by definition of graph-terms we have crossed the λ binding this variable and given to its premise a type of the form $B \multimap C$ and to the premise of x type B;
- or a leaf of a contraction tree which is already labeled.

As described previously we then switch 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. If it is a left conclusion of @ or a conclusion of ax 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 and proceed upwards.

Lemma 10 *After the labeling visit, all edges but those in contraction trees have been typed.*

The key point is to observe that during the upwards trip all edges visited are typed. The only delicate case is when we go up from the root of a contraction tree to a right conclusion of a @. But in this case the edge has already been typed during the previous downwards trip.

See the result of this first phase on our example, appendix C figure 6.

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.

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 $s_1 B_1$ (resp. $s_2 B_2$) the type of the leaf (resp. of the root-edge), where s_i is a sequence of indexed modalities and B_i is a type which does not start with a modality. These types belong to the grammar of N's in definition 2 and so in s_1 and s_2 §'s are indexed by constants.

Definition 4 *Let A, B be parameterized types such that $[A] = [B]$; we define the constraints $\mathcal{C}(A, B)$ identifying A and B in the following way:*

- *if $A = !^n A'$ and $B = !^m B'$ (or $A = §^n A'$ and $B = §^m A'$) where A' and B' do not start with a ! (resp. a §), then: $\mathcal{C}(A, B) = (n = m) \wedge \mathcal{C}(A', B')$,*
- *if $A = !^n A'$ (resp. $A = §^n A'$) and B does not start with ! (resp. §) then: $\mathcal{C}(A, B) = (n = 0) \wedge \mathcal{C}(A', B)$,*
- *if $A = A_1 \multimap A_2$ and $B = B_1 \multimap B_2$ then: $\mathcal{C}(A, B) = \mathcal{C}(A_1, B_1) \wedge \mathcal{C}(A_2, B_2)$,*
- *if $A = B = \alpha$, then $\mathcal{C}(A, B) = true$,*
- *$\mathcal{C}(A, B) = \mathcal{C}(B, A)$.*

If types B_1 and B_2 can be identified by constraints we add $\mathcal{C}(B_1, B_2)$ to $\mathcal{C}_1$. Otherwise typing fails.

Then, if s_2 can be written as a concatenation $s'_2.s'_1$ where s'_1 can be identified to s_1: first add the corresponding constraints to $\mathcal{C}_1$; we then need to place along the branch enough doors to introduce the sequence s'_2. Observe that as contractions must be done on !-marked formulas, §-doors can not immediately precede c-nodes. Besides as there are a finite number of §'s in s'_2 there are a finite number of ways to place the §-doors in the branch; we choose one possibility. We introduce !-doors parameterized by new parameters and add to $\mathcal{C}_1$ the equations expressing the fact that the sequence of doors on this branch introduces s'_2. Finally we also add to $\mathcal{C}_1$ inequations imposing that the formula before each contraction is marked by a bang (so that contraction is valid). For an example, see appendix D.

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 $\mathcal{C}_1$.

So altogether the conditions on the contraction tree have been expressed by the system $\mathcal{C}_1$ and the parameterized types attributed to the edges (note that the parameters of these depend on the new integer variables introduced for the contraction tree).

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 9, hence we conclude:

Theorem 11 *The type instantiation problem for normal lambda-terms (problem 2) is decidable.*

References

[Asp98] Andrea Asperti. Light affine logic. In *"Proceedings of the 13th Symposium on Logic in Computer Science"*. IEEE Computer Society, 1998.

[Bai00] P. Baillot. Stratified coherent spaces: a denotational semantics for light linear logic. Tech. report 0025, University of Edinburgh, 2000. presented at Workshop on Implicit Computational Complexity (ICC) 2000.

[CM01] P. Coppola and S. Martini. Typing lambda-terms in elementary logic with linear constraints. In *Proceedings TLCA'01*, 2001.

[DJ99] V. Danos and J.-B. Joinet. Linear logic and elementary time. First Workshop on Implicit Computational Complexity (ICC'99), 1999.

[DJS94] V. Danos, J.-B. Joinet, and H. Schellinx. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33(6), 1994.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

[Hof00a] M. Hofmann. Programming languages capturing complexity classes. *SIGACT News*, 2000. Logic Column 9.

[Hof00b] M. Hofmann. Safe recursion with higher types and BCK-algebra. *APAL*, 2000. to appear.

[LM93] D. Leivant and J.-Y. Marion. Lambda-calculus characterisations of polytime. *Fundamenta Informaticae*, 19:167–184, 1993.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Rov99] L. Roversi. A P-time completeness proof for light logics. In *Proceedings CSL*, volume 1683 of *LNCS*. Springer-Verlag, 1999.

[Rov00] L. Roversi. Light affine logic as a programming language: a first contribution. *International Journal of Foundations of Computer Science*, 11(1), 2000.

[Sch94] H. Schellinx. *The Noble Art of Linear Decorating*. ILLC Dissertation Series 1994-1, Institute for Language, Logic and Computation, University of Amsterdam, 1994.

[Ter01] K. Terui. Light Affine Lambda-calculus and polytime strong normalization. In *LICS'01*, 2001.

[Wad91] Philip Wadler. Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991.

A Figures:

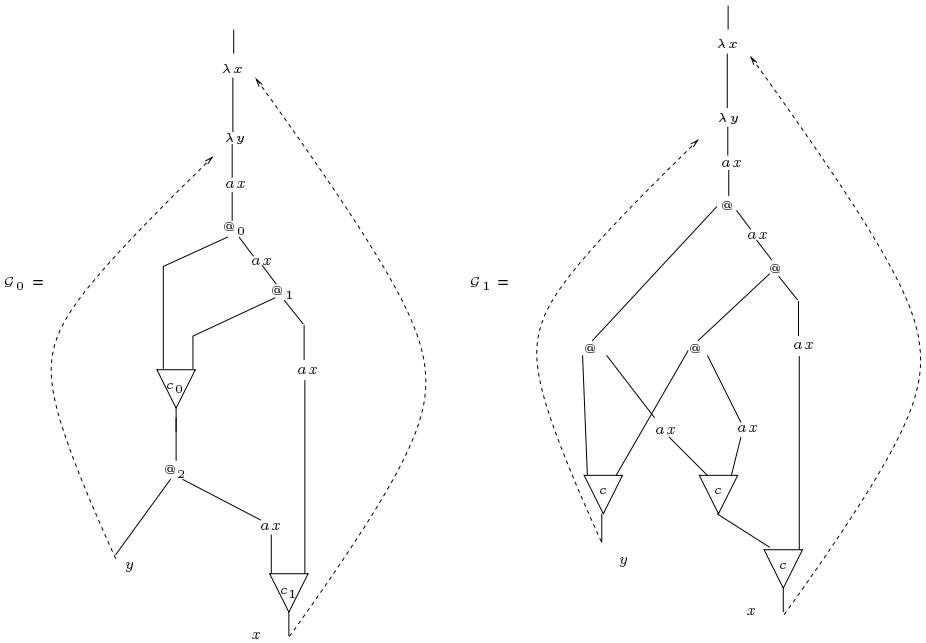

Figure 3: Two graphs corresponding to $\lambda x \lambda y (y) x ((y) x) x$

B Proofs

Proof :(proposition 3) Call x the variable node and λ_0 the lambda-node binding x. Let p be a path from the root-node to c. It can be extended to a complete path q containing x. By lemma 2, q contains λ_0. Therefore λ_0 and c are comparable. If $\lambda_0 \leq c$ we are over. Otherwise λ_0 belongs to p. As this is then true for any path from the root-node to c we conclude that $\lambda_0 \geq junction(c)$. □

Proof :(proposition 4) First, as $c_1 \leq c_2$ there is at least one path p of c_1 that contains c_2; as p also contains $junction(c_1)$ we know that c_2 and $junction(c_1)$ are comparable. If $junction(c_1) \leq c_2$ the property is valid, so let us assume $junction(c_1) \geq c_2$. Take q a path of c_2; it can be extended into a path q' of c_1. We know that q' contains $junction(c_1)$, so as $junction(c_1) \geq c_2$ we conclude that $junction(c_1)$ belongs to q. Hence $junction(c_1)$ belongs to all paths of c_2 and so $junction(c_1) \geq junction(c_2)$. □

Proof :(theorem 6)

Assume $\mathcal{G}$ satisfies bracketing, level, scope, junction and !-box conditions. First let us define the boxes: introduce one box for each opening door, with

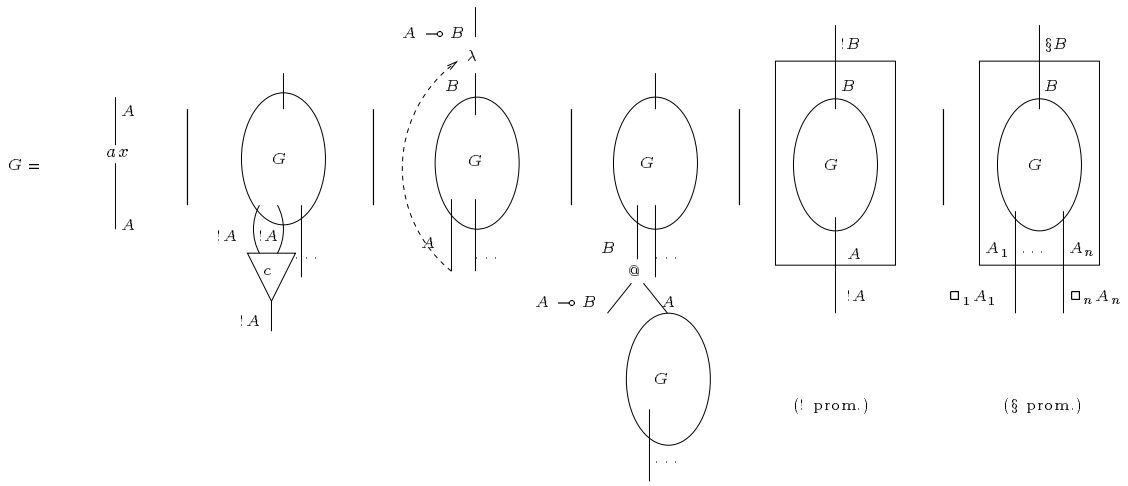

Figure 4: Derivation graphs inductive definition

outputs the closing doors matching this opening door. We claim that the junction condition implies that given a closing door, it is an output of exactly one box. Indeed, assume closing door d_3 matched with two opening doors d_1 and d_2. Call B_1 and B_2 the boxes corresponding to d_1 and d_2. Let N be a maximal node such that $d_3 \leq N \leq d_1$ and $d_3 \leq N \leq d_2$. The node N is a contraction node and it belongs to B_1 and B_2. Call N' its junction: by the junction condition, N' should belong to B_1 and B_2, so $d_3 \leq N' \leq d_1$ and $d_3 \leq N' \leq d_2$, which contradicts the maximality of N.

Now, once the boxes are defined we show that $\mathcal{G}$ is a derivation graph, proceeding by induction on its size. If $\mathcal{G}$ is a variable, then it is a derivation graph. Otherwise let us consider the first node below the root-node. By the bracketing condition we know it cannot be a closing door. If it is a λ-node we can conclude by induction hypothesis. The delicate cases are those of a @-node and of an opening door. We describe the case of an opening door, as the other one is handled in a similar way.

Call B the box corresponding to this opening door. Consider the contractions below B. We distinguish two cases:

- First case: no contraction below B has its junction inside B. Then, if below the outputs of B there are only variables nodes, then call $\mathcal{G}'$ the content of B. The typed graph $\mathcal{G}'$ also satisfies the conditions, so by induction hypothesis we know it is valid. As $\mathcal{G}$ is obtained by a box on $\mathcal{G}'$ it is also valid.

  Otherwise if B has an output with non-variable nodes below, call $\mathcal{G}_1$ the subgraph of $\mathcal{G}$ below this output. If $\mathcal{G}_1$ has bound variables, then their λs cannot be in B because of the scope variable, hence they are in $\mathcal{G}_1$. Besides, if $\mathcal{G}_1$ contains a contraction, then its junction is by hypothesis also in $\mathcal{G}_1$, so both premises of the contraction are in $\mathcal{G}_1$. It follows that $\mathcal{G}_1$ is also a graph-term. Let $\mathcal{G}_2 = \mathcal{G} \setminus \mathcal{G}_1$: both $\mathcal{G}_1$ and $\mathcal{G}_2$ satisfy the conditions, so by i.h. they are valid, hence we conclude that $\mathcal{G}$ is valid.

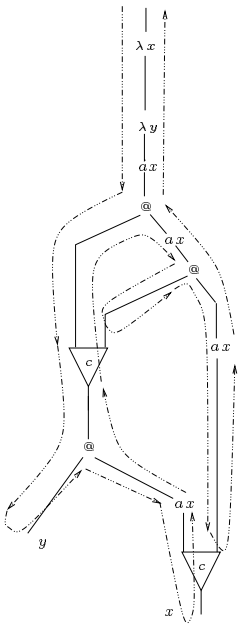

Figure 5: Example of visit

- Second case: at least one contraction below B has its junction in B. Take c_0 a contraction with this property which is minimal. Then if c_0 is a bottom-most node of the graph we can remove it and conclude by i.h. . Otherwise:

  - By proposition 3 the bound variables below c_0 have their λs above $junction(c_0)$ or below c_0. The first possibility is ruled out by the scope condition, so these variables have their λs below c_0.
  - By proposition 4 the contractions below c_0 have their junction below c_0, otherwise their junction would be inside B and hence contradict the minimality of c_0. Therefore, such a contraction has both premises below c_0.

  Let $\mathcal{G}_1$ be the subgraph of $\mathcal{G}$ below c_0. By the two previous observations we know that $\mathcal{G}_1$ is a graph-term. Let $\mathcal{G}_2 = \mathcal{G} \setminus \mathcal{G}_1$.

  The junction condition implies that c_0 does not belong to any box: indeed, if c_0 was in a box B', then $junction(c_0)$ would belong to B', and so B and B' would not be disjoint. We would have either $B \subseteq B'$ or $B' \subseteq B$, which are both impossible.

  So, as c_0 is at depth 0, the matching between closing and opening doors are the same in $\mathcal{G}_1$ and in $\mathcal{G}$. Similarly for $\mathcal{G}_2$. We deduce from that that $\mathcal{G}_1$ and $\mathcal{G}_2$ also satisfy the conditions, and so by i.h. that they are valid. This implies then that $\mathcal{G}$ is valid.

□

C Example:

We consider the graph term $\mathcal{G}_0$ of figure 2, and try to type it with parameterized type:

$$!^a\alpha \multimap !\S^b(!^c(\S^e\alpha \multimap !^f(\alpha \multimap \alpha)) \multimap \S^d\alpha)$$

See figure 6 for the labeling visit and figure 7 for contraction trees decoration.

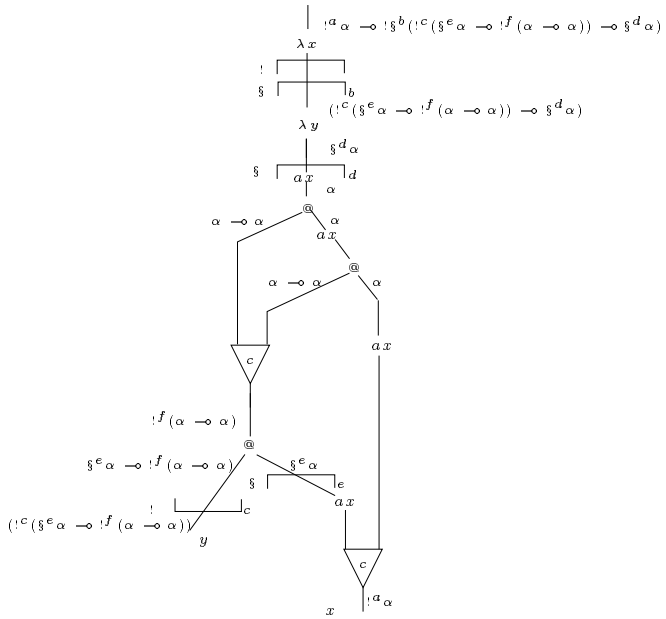

Figure 6: Labeling visit

We write down some of conditions on that example (see figures 7, 8 for the notations):

- bracketing:

  For the leftmost complete path, the condition states: $1+b+d \geq k_1+k_3+c$.

  There are 4 complete paths to consider.

- level:

  We give the example of the two paths linking λy to c_0:

  $$k_1 - d = k_2 - d.$$

- scope: for the leftmost occurrence of y we should have:

  $$\begin{cases} k_1 + k_3 + c \leq d \\ k_1 + k_3 + c = d \end{cases}$$

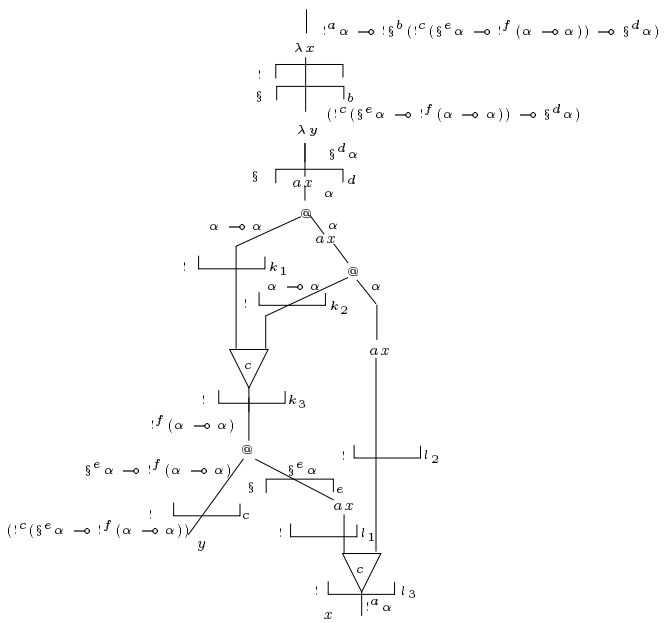

Figure 7: Labeling: contraction trees decoration

- junction: Let us express the junction condition for contraction c_1. Its junction is node $@_0$.

$$\begin{cases} -l_1 + e \leq 0 \\ -l_1 + e - k_3 - k_1 \leq 0 \\ -l_1 + e \leq 0 \\ -l_1 + e - k_3 - k_2 \leq 0 \\ -l_2 \leq 0 \end{cases}$$

- !-box conditions:

  Let us express the condition for instance for the !-door D_0 and the parameterized doors D_1 and D_2:

$$\begin{aligned} P(D_0, D_1) &= (b + d - k_1 \geq 0) \wedge (b + d - k_1 - k_3 < 0) \\ P(D_0, D_2) &= (b + d - l_2 < 0) \\ \neg P(D_0, D_1) \vee \neg P(D_0, D_2) &= (b + d - k_1 < 0) \vee (b + d - k_1 - k_3 \geq 0) \vee (b + d - l_2 \geq 0) \end{aligned}$$

D Contraction tree decoration

We give an example of application of the contraction trees decoration procedure; we apply it to one branch given the root and leaf parameterized types (figure 9).

For each choice of decoration of the branch we get the following constraints $\mathcal{C}_1$:

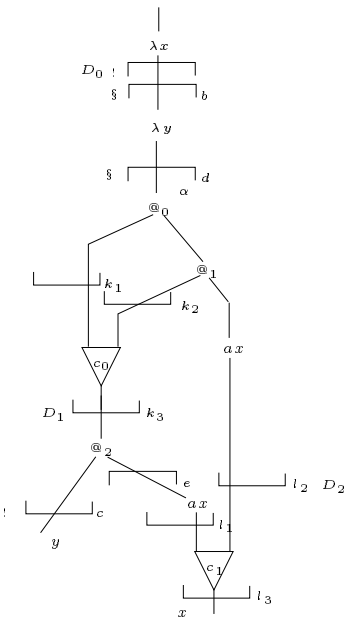

Figure 8: Notations for the conditions

- Choice 1:

$$\begin{cases} x_1 + x_2 + x_3 = c \\ x_3 \geq 1 \end{cases}$$

- Choice 2:

$$\begin{cases} x_1 + x_2 = a \\ x_3 + x_4 = c \\ x_4 \geq 1 \\ x_2 \geq 1 \end{cases}$$

- Choice 3:

$$\begin{cases} x_1 + x_2 + x_3 = a \\ x_4 = c \\ x_3 \geq 1 \end{cases}$$

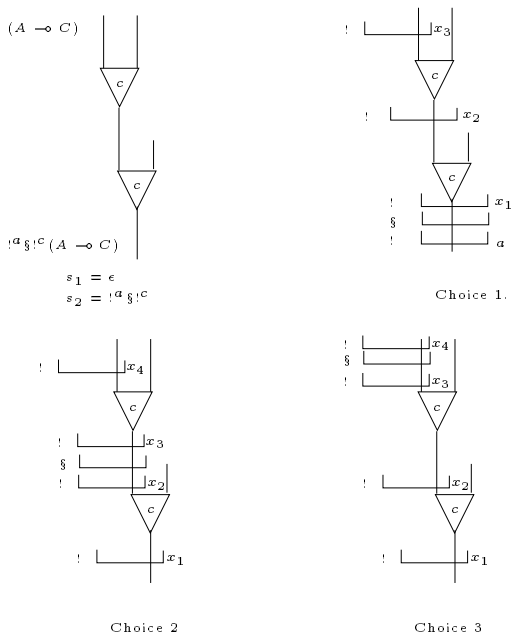

Figure 9: Example of choices of decoration