Quantitative Inhabitation for Different Lambda Calculi in a Unifying Framework

VICTOR ARRIAL, Université Paris Cité, CNRS, IRIF, France
GIULIO GUERRIERI, Aix Marseille Univ, CNRS, LIS, Marseille, France; and Edinburgh Research Centre, Central Software Institute, Huawei, UK
DELIA KESNER, Université Paris Cité, CNRS, IRIF; Institut Universitaire de France, France

We solve the inhabitation problem for a language called $\lambda!$, a subsuming paradigm (inspired by call-by-push-value) being able to encode, among others, call-by-name and call-by-value strategies of functional programming. The type specification uses a non-idempotent intersection type system, which is able to capture quantitative properties about the dynamics of programs. As an application, we show how our general methodology can be used to derive inhabitation algorithms for different lambda-calculi that are encodable into $\lambda!$.

51

CCS Concepts: • **Mathematics of computing → Lambda calculus**; • **Theory of computation → Lambda calculus**.

Additional Key Words and Phrases: inhabitation, call-by-push-value, quantitative types, lambda-calculus

ACM Reference Format:
Victor Arrial, Giulio Guerrieri, and Delia Kesner. 2023. Quantitative Inhabitation for Different Lambda Calculi in a Unifying Framework. *Proc. ACM Program. Lang.* 7, POPL, Article 51 (January 2023), 31 pages. https://doi.org/10.1145/3571244

1 INTRODUCTION

Inhabitation. *Type systems* are formalisms assigning a *type* to the constructs of a programming language, usually represented by a *term calculus*. Types enforce some particular specification (*e.g.* termination, memory safety, deadlock freeness, etc), so that they guarantee the construction of well-behaved terms: "well-typed programs cannot go wrong" [Milner 1978]. A judgment in a given type system $\mathcal{X}$ is written $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$, where t is a term, σ is the type assigned to t, and Γ is an *environment* assigning types to the (free) variables of t. There are at least three problems naturally arising for a given type system: (1) *type checking*: given an environment Γ, a term t and a type σ, decide whether $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$; (2) *typability*: given a term t, decide whether there exists an environment Γ and a type σ such that $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$; (3) *inhabitation*: given an environment Γ, and a type σ, decide whether there is a term such that $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$. Inhabitation corresponds to decide the existence of a program (term t) that satisfies the given specification (type σ) under some assumptions (environment Γ). The inhabitation problem is naturally related to proof-search, where the types are seen as propositions in some underlying logic. Decidability of the inhabitation problem can be also seen as a particular tool for type-based *program synthesis* [Bessai et al. 2018; Manna and Waldinger 1980], whose task is to construct—from scratch—a program that satisfies some high-level formal specification.

Authors' addresses: Victor Arrial, Université Paris Cité, CNRS, IRIF, France, arrial@irif.fr; Giulio Guerrieri, Aix Marseille Univ, CNRS, LIS, Marseille, France; and Edinburgh Research Centre, Central Software Institute, Huawei, UK, giulio.guerrieri@lis-lab.fr; Delia Kesner, Université Paris Cité, CNRS, IRIF; and Institut Universitaire de France, France, kesner@irif.fr.

2475-1421/2023/1-ART51
https://doi.org/10.1145/3571244

Quantitative Type Systems. Intersection type systems [Coppo and Dezani-Ciancaglini 1978, 1980] were introduced for the λ-calculus to increase the typability power of simple types by introducing a new *intersection* type constructor $\wedge$, which is, in principle, associative, commutative and *idempotent* (*i.e.* $\sigma \wedge \sigma = \sigma$). Intersection types allow terms to have different types simultaneously, *e.g.* a term t has type $\sigma \wedge \tau$ whenever t has both the type σ and the type τ. In these (idempotent) systems typability and inhabitation are both undecidable [Urzyczyn 1999]. However, intersection types constitute a powerful tool to reason about *qualitative* semantic properties of programs, for example, there are intersection type systems characterizing different notions of normalization [Coppo et al. 1981; Pottinger 1980], in the sense that a term t is typable in a given system if and only if t is normalizing for that particular notion. By removing idempotency [de Carvalho 2007; Gardner 1994], a term of type $\sigma \wedge \sigma \wedge \tau$ can be seen as a resource that, during execution, can be used once as a data of type τ and twice as a data of type σ. The resulting *non-idempotent* type systems for the λ-calculus do not only provide qualitative characterization of operational properties, but also *quantitative* ones, in the sense that a term t is still typable if and only if is normalizing, and in addition, any type derivation of t gives a *bound* to the execution time for t (the number of steps to reach a normal form) [Accattoli et al. 2020; de Carvalho 2018]. In such a setting, typability is still undecidable, nevertheless inhabitation becomes *decidable* [Bucciarelli et al. 2014, 2018]. In particular, an algorithm solving the inhabitation problem for a quantitative type system can be seen as a decidable tool for type-based *quantitative program synthesis*, which aims to construct—from scratch—a program that satisfies some quantitative specification.

Call-by-Push-Value. P.B. Levy [1999] introduced Call-by-Push-Value (CBPV) as a *subsuming* paradigm, so that different evaluation strategies of the λ-calculus can be captured in a uniform framework by the simple use of two primitives: *thunk* (to pause a computation) and *force* (to resume a computation). This mechanism is powerful enough to encode, in particular, Call-by-Name (CBN) and Call-by-Value (CBV), the two most well-known evaluation mechanisms in functional programming. The original CBPV has been introduced in a *simply typed* framework, but the underlying (untyped) syntax and operational semantics—the ones we are interested in here—already provide a powerful *untyped* subsuming mechanism. Despite that, CBN and CBV have always been studied notably by developing different techniques for one and the other. Some rare exceptions are [Bucciarelli et al. 2020; Faggian and Guerrieri 2021; Kesner and Viso 2021], where some particular property for CBN/CBV (*e.g.* quantitative typing in the first one, factorization in the second case, tight typing in the third one) is derived from the corresponding property for a language that is a restriction of CBPV, via a suitable CBN/CBV encoding. Such a language can be the *bang calculus* [Ehrhard and Guerrieri 2016; Guerrieri and Manzonetto 2018; Guerrieri and Olimpieri 2021] (in turn inspired by Ehrhard [2016], combining ideas from Levy's CBPV [1999] and Girard's linear logic [1987]), or its variant the *λ!-calculus* [Bucciarelli et al. 2020; Kesner and Viso 2021] where reduction rules act at a *distance*.

What this paper is about. We address the challenging problem of inhabitation for $\lambda!$ in the framework of *quantitative* type systems. This constitutes a first proposal in the literature addressing inhabitation for such class of languages. Our language is given by the $\lambda!$-calculus, and its associated quantitative (non-idempotent) type system is $\mathcal{U}$ [Bucciarelli et al. 2020]. We do not simply give an algorithm searching for a term that can be typed with a given environment Γ and type σ, but we solve a more ambitious goal: our algorithm generates (a finite representation of) *all* and *only* such typable terms. Indeed, our algorithm is parametrized by a (tree) grammar, so that a *finite* set of answers can be generated by this grammar, from which the whole *infinite* set of solutions can be finally recognized. Our inhabitation algorithm is shown to be terminating, sound and complete: every solution to the inhabitation problem is found, and no solution is forgotten.

As a general application, we also address inhabitation for other models of computation—such as CBN and CBV—that are encodable in $\lambda!$. This is done by essentially using two crucial tools: (1) an embedding that encodes each model of computation into $\lambda!$; (2) a grammar generating the embedding of the *finite* set of answers for each model of computation. By instantiating the general inhabitation algorithm for the $\lambda!$-calculus with the image of each embedding, we solve the inhabitation problem for each corresponding model of computation. Thus, our general methodology derives an inhabitation algorithm for each alternative model of computation subsumed by our unifying framework $\lambda!$, thus providing a strong and powerful tool for (quantitative) inhabitation.

Concretely, as special cases, we recover the well-known algorithm to solve the inhabitation problem for CBN [Bucciarelli et al. 2014, 2018], and we obtain a *new* algorithm (the first one in the literature) to solve the problem for CBV. The main contributions of the paper are:

(1) We give an inhabitation algorithm for the $\lambda!$-calculus equipped with type system $\mathcal{U}$. This is *novel* and far from trivial because of the built-in constructors of $\lambda!$, which are essential to encode evaluation strategies with different calling methods, and more sophisticated than those of the λ-calculus.
(2) Our approach is a *several-for-one deal*! We solve the type inhabitation problem for $\lambda!$ once and for all: from that, we derive the inhabitation algorithms for other models of computations, including CBN and CBV, *for free*. While the inhabitation problem was already solved for CBN [Bucciarelli et al. 2014], the solution for CBV is a particular novel contribution.
(3) Our algorithm gives not only one but *all* the solutions for a given typing. This is made by a fine analysis of the completeness property: a crucial (and novel) notion of *basis* is defined so that the whole *infinite* set of solutions can be *finitely* captured for each input typing.
(4) We provide an open-source implementation of the algorithm in OCaml [Arrial 2023]. The implementation supports different verbose modes yielding a trace of the algorithm search.

Related works and Applications. Our work relates to the following theoretical and applied topics:

(1) *Proof-search:* Terms can be seen as proofs by means of the Curry-Howard isomorphism, so that an inhabitation search algorithm can be seen as a proof search method. In both cases, only normal terms/cut-free proofs are constructed as outputs. In this respect, proof-search naturally appeared in the operational semantics of logical languages such as PROLOG, the idea was further extended to more expressive logical languages by Miller et al. [1991]. Closer to our paper, a proof-search method for a fragment of intuitionistic linear logic was introduced by Hodas and Miller [1994], and its implementation reported in [Cervesato et al. 2000]. These works only consider exponential formulae in some restricted positions, which makes the inhabitation problem much easier. The problem was further studied by Hughes and Orchard [2020] in the context of program synthesis for graded modal types, a type system with quantitative features. Thus, our ideas for solving quantitative inhabitation provides a procedure to solve proof-search in future proof-assistants having quantitative features.
(2) *Program synthesis:* Non-idempotent types can be seen as specifications of resource consumption, so that the inhabitation problem can be seen as a particular case of (quantitative) program synthesis. This is for example highlighted by Hughes and Orchard [2020] in the context of graded modal types. Other approaches using combinatory logic with bounded polymorphic idempotent intersection types have shown to have numerous applications [Bessai 2013; Bessai et al. 2014; Düdder 2014; Plate 2013; Vasileva 2013; Wolf 2013]. But the field of program synthesis is much larger than the inhabitation problem. There are many ways to express specifications (not only by means of types) and to search for programs that meet that specification (not only programs in normal form). The inhabitation problem is only a small portion of the vast domain of program synthesis, for which this work modestly contributes.

Aforementioned graded modal types [Orchard et al. 2019] form another resource-aware type system with some similarities and differences with non-idempotent intersection types. On the one hand, graded modal types with nonnegative integer grades (resp. infinite grades) can be seen as a subset of non-idempotent (resp. idempotent) intersection types. There are however several differences. Graded modal types give a specification of how many times a resource can be used with a given fixed type, while a non-idempotent type is much more general, as a term may be typed with different types, each one having a different grade. As a consequence, non-idempotent types can be used to build denotational models, in the sense that they enjoy subject reduction and subject expansion, while graded modal types [Orchard et al. 2019] only enjoy subject reduction. Driven by distinct purposes, these design differences have an important consequence: non-idempotent intersection types are not only sound with respect to strong normalization, but also complete, in the sense that a terminating term necessarily has an associated non-idempotent typing derivation. Such a property can typically be illustrated with the self application term $\lambda x.xx$. Therefore, non-idempotent intersection types provide an undecidable (semantic) framework, in contrast, graded modal types give concrete decidable resource-aware typing systems.

On the other hand, graded modal types are more general than non-idempotent intersection types: grades can be expressed by any pre-ordered semirings (not only nonnegative integers), which do not have a counterpart in non-idempotent intersection types. Also, graded modal types can express full polymorphism, whereas non-idempotent intersection types only capture a *finitary* form of it.

Another interesting remark is that, while it is trivial to erase graded information from graded modal types to recover a simple type (if no polymorphism is used), it is still an open problem how to convert a simply typed derivation into a non-idempotent derivation.

Other related works are discussed in Sect. 7.

Summary. Sec. 2 recalls the $\lambda!$-calculus and its type system $\mathcal{U}$. Sec. 3 defines the finite basis that capture the (potentially infinite) set of solutions to the inhabitation problem. Sec. 4 defines a type relation used to guess types while the algorithm runs. Sec. 5 presents the algorithm and proves its termination, soundness and completeness. In Sec. 6 we propose two concrete applications of the general algorithm to solve the inhabitation problems in CBN in CBV. We conclude in Sec. 7.

2 PRELIMINARIES

In this section we first recall some basic standard notions on tree grammars [Comon et al. 2008]. We then introduce the $\lambda!$-calculus [Bucciarelli et al. 2020] (an extension of the bang calculus [Ehrhard and Guerrieri 2016] which recovers completeness and confluence by introducing reduction rules acting at a distance), which subsumes the λ-calculus by adding some primitives that allow to capture, among others, Call-by-Name (CBN) and Call-by-Value (CBV). We first present syntactic and operational notions of the untyped version of the calculus. We then introduce its quantitative typing system characterizing normalization. Finally, we formally define the inhabitation problem.

2.1 Some Notations About Grammars

A **(regular, first-order) term grammar** G is defined by a tuple $G = (\Sigma, S, R, s)$, where Σ is a *ranked alphabet* (*i.e.* symbols have an associated unique arity), S is a finite set of *nonterminal* symbols (denoted by letters g and n), $s \in S$ is the *start* symbol, and R is a finite set of production rules of the form $A \mapsto T$, where $A \in S$ and T is a term in the associated term algebra $\mathcal{T}_\Sigma(S)$, *i.e.* the set of all terms built up from symbols in $\Sigma \cup S$ according to their arities (nonterminals are considered nullary). We write $A \mapsto T_1 \mid \ldots \mid T_n$ if all production rules $A \mapsto T_i$ $(1 \leq i \leq n)$ start with the same $A \in S$. Given a production rule $A \mapsto T$, if T is also a nonterminal symbol, then the rule is called **silent**, otherwise it is called **non-silent**.

A **context** C is an element of the term algebra $\mathcal{T}_{\Sigma\cup\{\diamond\}}(S)$, where $\diamond$ is a fresh constant of arity 0 occurring exactly once in C. A term $o \in \mathcal{T}_\Sigma(S)$ **is derived** in a single step into a term $p \in \mathcal{T}_\Sigma(S)$, written $o \rightsquigarrow p$, if there is a context $C \in \mathcal{T}_{\Sigma\cup\{\diamond\}}(S)$ and a production rule $A \mapsto T$ such that $o = C\langle A\rangle$ and $p = C\langle T\rangle$, where the generic notation $C\langle U\rangle$ is used to denote the term obtained by replacing the symbol $\diamond$ in C by U. The reflexive-transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow\!\!\!\!\rightsquigarrow$.

We may also make use of *patterns* for specific shapes of the production rules of some grammars. A **pattern** is noted $\mathcal{G} \mapsto \mathcal{H}$, where $\mathcal{G} \in \mathcal{J}$ is a *meta-variable* for nonterminal symbols and $\mathcal{H} \in \mathcal{T}_\Sigma(\mathcal{J})$. *E.g.*, consider a grammar whose alphabet contains two symbols $\mathtt{Lam}(\_)$ and $\mathtt{Bng}(\_)$ of arity 1 and whose productions rules are constrained by the patterns $\mathcal{G} \mapsto \mathtt{Lam}(\mathcal{G}')$ and $\mathcal{G} \mapsto \mathtt{Bng}(\mathcal{G}')$, then the rules of G can only be of the form $n_1 \mapsto \mathtt{Lam}(n_2)$ and $n_3 \mapsto \mathtt{Bng}(n_4)$, where $n_i \in S$ for all i. This notion is inherited from meta-variables in higher-order rewriting [Klop et al. 1993].

2.2 The $\lambda!$-Calculus

Let us first introduce the term syntax of the $\lambda!$-calculus [Bucciarelli et al. 2020]. Given a countably infinite set $\mathcal{X}$ of variables $x, y, z, \ldots$, the set of terms Λ is given by the following inductive definition:

$$\textbf{(Terms)}\quad t, u, s \;::=\; x \in \mathcal{X} \mid tu \mid \lambda x.t \mid !t \mid \mathsf{der}(t) \mid t[x\backslash u]$$

The set Λ includes λ-terms (**variables** x, **abstractions** $\lambda x.t$ and **applications** tu) as well as three new constructors: a **closure** $t[x\backslash u]$ representing a pending **explicit substitution (ES)** $[x\backslash u]$ on a term t, a **bang** $!t$ to freeze the execution of t, and a **dereliction** $\mathsf{der}(t)$ to fire again the frozen term t. From now on, we set $I := \lambda z.z$, $\Delta := \lambda x.x!x$, and $\Omega := \Delta!\Delta$.

Abstractions $\lambda x.t$ and closures $t[x\backslash u]$ bind the variable x in the term t. The notions of **free** and **bound** variables are defined as expected, in particular $\mathsf{fv}(\lambda x.t) := \mathsf{fv}(t) \setminus \{x\}$ and $\mathsf{fv}(t[x\backslash u]) := \mathsf{fv}(u) \cup (\mathsf{fv}(t) \setminus \{x\})$. The usual notion of α-conversion [Barendregt 1984] is extended to the whole set Λ, and terms are identified up to α-conversion. We denote by $t\{x\backslash u\}$ the usual (capture avoiding) meta-level substitution of the term u for all free occurrences of the variable x in the term t.

Full contexts (F), **surface contexts** (S) and **list contexts** (L), which can be seen as terms containing exactly one **hole** $\diamond$, are inductively defined as follows:

$$\begin{array}{lcl} \mathsf{F} & ::= & \diamond \mid \mathsf{F}t \mid t\mathsf{F} \mid \lambda x.\mathsf{F} \mid !\mathsf{F} \mid \mathsf{der}(\mathsf{F}) \mid \mathsf{F}[x\backslash t] \mid t[x\backslash \mathsf{F}] \\ \mathsf{S} & ::= & \diamond \mid \mathsf{S}t \mid t\mathsf{S} \mid \lambda x.\mathsf{S} \mid \mathsf{der}(\mathsf{S}) \mid \mathsf{S}[x\backslash t] \mid t[x\backslash \mathsf{S}] \\ \mathsf{L} & ::= & \diamond \mid \mathsf{L}[x\backslash t] \end{array}$$

L and S are special cases of F. The hole can occur everywhere in F, while in S it cannot occur under a !. We write $\mathsf{F}\langle t\rangle$ for the term obtained by replacing the hole in F with the term t.

The following **rewriting rules** are the base components of our reduction relations. Any term having the shape of the left-hand side of one of these three rules is called a **redex**.

$$\begin{array}{rccl} \textbf{(Distant Beta)} & \mathsf{L}\langle \lambda x.t\rangle\, u & \mapsto_{\mathsf{dB}} & \mathsf{L}\langle t[x\backslash u]\rangle \\ \textbf{(Substitute Bang)} & t[x\backslash \mathsf{L}\langle !u\rangle] & \mapsto_{\mathsf{s!}} & \mathsf{L}\langle t\{x\backslash u\}\rangle \\ \textbf{(Distant Bang)} & \mathsf{der}(\mathsf{L}\langle !t\rangle) & \mapsto_{\mathsf{d!}} & \mathsf{L}\langle t\rangle \end{array}$$

Rule dB (resp. $\mathsf{s!}$) is assumed to be capture free, so no free variable of u (resp. t) is captured by the context L. The rule dB fires a standard β-redex and generates an ES. The rule $\mathsf{s!}$ fires an ES provided that its argument is a bang. The rule $\mapsto_{\mathsf{d!}}$ defrosts a frozen term. In all of these rewrite rules, the reduction acts *at a distance* [Accattoli and Kesner 2010]: the main constructors involved in the rule can be separated by a finite—possibly empty—list L of ES. This mechanism unblocks redexes that otherwise would be stuck, *e.g.* $(\lambda x.x)[y\backslash w]!z \mapsto_{\mathsf{dB}} x[x\backslash !z][y\backslash w]$ fires a β-redex by taking $\mathsf{L} = \diamond[y\backslash w]$ as the list context in between the function $\lambda x.x$ and the argument $!z$.

The **surface reduction** relation $\rightarrow_{\mathsf{S}}$ is the surface closure of any of the three rewrite rules $\mapsto_{\mathsf{dB}}$, $\mapsto_{\mathsf{s!}}$ and $\mapsto_{\mathsf{d!}}$, *i.e.* $\rightarrow_{\mathsf{S}}$ only fires redexes in surface contexts, and not under bang. Similarly, the **full**

reduction relation $\to_{\mathsf{F}}$ is the full closure of any of the rewrite rules, so that $\to_{\mathsf{F}}$ reduces under full contexts and thus the bang looses its freezing behavior. For example,

$$(\lambda x.!\mathsf{der}(!x))!y \;\to_{\mathsf{S}}\; (!\mathsf{der}(!x))[x\backslash !y] \;\to_{\mathsf{S}}\; !(\mathsf{der}(!y)) \;\to_{\mathsf{F}}\; !y$$

Note that the first two steps are also $\to_{\mathsf{F}}$-steps, while the last step is not an $\to_{\mathsf{S}}$-step. More generally, we have $\to_{\mathsf{S}}\subsetneq\to_{\mathsf{F}}$. For $\mathcal{R} \in \{\mathsf{S}, \mathsf{F}\}$, $\twoheadrightarrow_{\mathcal{R}}$ is the reflexive-transitive closure of $\to_{\mathcal{R}}$.

Some terms of the $\lambda!$-calculus do not contain any redex. Terms without any redex for the surface reduction (*i.e.* under surface contexts) are called **surface normal forms**. Terms without any redex for the full reduction (*i.e.* under full contexts) are called **full normal forms**. For example, the term $!(\mathsf{der}(!y))$ is a surface normal form but not a full normal form since $\mathsf{der}(!y)$ is a redex under a bang.

As a matter of fact, some ill-formed terms are not redexes but neither represent a desired computation result. They are called **clashes** and have one of the following forms:

$$\mathsf{L}\langle !s\rangle\, u \qquad s[x\backslash \mathsf{L}\langle \lambda x.u\rangle] \qquad \mathsf{der}(\mathsf{L}\langle \lambda x.u\rangle) \qquad t(\mathsf{L}\langle \lambda x.u\rangle) \text{ if } t \neq \mathsf{L}'\langle \lambda y.s\rangle$$

This previous *static* notion of ill-formed term is lifted to a *dynamic* level. Indeed, a term t is a **surface clash-free** if it does not S-reduce to a term with a clash outside the scope of some bang, *i.e.* if there are no surface context S and clash c such that $t \twoheadrightarrow_{\mathsf{S}} \mathsf{S}\langle c\rangle$. Similarly, a term t is **full clash-free** if there are no full context F and clash c such that $t \twoheadrightarrow_{\mathsf{F}} \mathsf{F}\langle c\rangle$. For example, $x!(y(\lambda z.z))$ is surface clash-free but not full clash-free since it has a clash $y(\lambda z.z)$ under a bang. Both notions are stable under reduction. Finally, some terms contain neither redexes nor clashes. A **surface clash-free normal form** (resp. **full clash-free normal form**) is a surface normal form which is surface clash-free (resp. full normal form which is full clash-free). These are the desired results of the computation, and they can even be characterized by a tree grammar [Bucciarelli et al. 2020].

Given $\mathcal{R} \in \{\mathsf{S}, \mathsf{F}\}$, a term t is said to be $\mathcal{R}$**-normalizing** iff $t \twoheadrightarrow_{\mathcal{R}} p$ for some $\mathcal{R}$-normal form p. As $\to_{\mathsf{S}}\subsetneq\to_{\mathsf{F}}$, some terms may be S-normalizing but not F-normalizing, *e.g.* $x!(\Delta!\Delta)$.

2.3 The Quantitative Typing System

We now present the quantitative typing system $\mathcal{U}$ [Bucciarelli et al. 2020], based on [de Carvalho 2007; Gardner 1994]. It contains functional and intersection types. Intersection is considered to be associative, commutative but *not idempotent*, thus an intersection type is represented by a (possibly empty) *finite multiset* $[\sigma_i]_{i\in I}$. Formally, given a countably infinite set $\mathcal{TV}$ of type variables $\alpha, \beta, \gamma, \ldots$, we inductively define:

$$\begin{array}{rrcl} \textbf{(Types)} & \sigma, \tau, \rho & ::= & \alpha \in \mathcal{TV} \mid \mathcal{M} \mid \mathcal{M} \Rightarrow \sigma \\ \textbf{(Multitypes)} & \mathcal{M}, \mathcal{N} & ::= & [\sigma_i]_{i\in I} \text{ where } I \text{ is a finite set} \end{array}$$

(**Type**) **environments**, noted Γ, Δ, are functions from variables to multitypes, assigning the **empty multitype** $[\,]$ to all the variables except a finite number (possibly zero). The **empty environment**, which maps every variable to $[\,]$, is denoted by $\emptyset$. The *domain* of Γ is $\mathsf{dom}(\Gamma) = \{x \in \mathcal{X} \mid \Gamma(x) \neq [\,]\}$. Given the environments Γ and Δ, $\Gamma+\Delta$ is the environment mapping x to $\Gamma(x) \uplus \Delta(x)$, where $\uplus$ denotes multiset union; and $+_{i\in I}\Delta_i$ is its obvious extension to the non-binary case, in particular $+_{i\in I}\Delta_i = \emptyset$ if $I = \emptyset$. We use Γ, Δ to denote $\Gamma + \Delta$ when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$, thus $x_1 : \mathcal{M}_1, \ldots, x_n : \mathcal{M}_n$ is the environment assigning $\mathcal{M}_i$ to x_i, for $1 \leq i \leq n$, and $[\,]$ to any other variable. We write $\Gamma \mathbin{\backslash\!\!\backslash} x$ for the environment assigning $[\,]$ to x, and acting as Γ otherwise.

A **typing judgment** is a triple of the form $\Gamma \vdash t : \sigma$, where Γ is a typing environment, t is a term (called the **subject** of the typing judgment), and σ is a type. The typing system $\mathcal{U}$ for the $\lambda!$-calculus is defined by the rules in Fig. 1. The axiom rule (ax) is relevant, *i.e.* there is no weakening. Rules (abs), (app) and (es) are standard. Rule (bag) has as many premises as elements in the finite (possibly empty) set of indices I: the conclusion types $!u$ with a multitype *gathering* all the (possibly

$$\frac{}{x:[\sigma]\vdash x:\sigma}\,\text{ax} \qquad \frac{\Gamma_u\vdash u:\mathcal{M}\Rightarrow\sigma \qquad \Gamma_s\vdash s:\mathcal{M}}{\Gamma_u+\Gamma_s\vdash us:\sigma}\,\text{app} \qquad \frac{(\,\Gamma_i\vdash u:\tau_i\,)_{i\in I} \qquad I \text{ finite}}{+_{i\in I}\Gamma_i\vdash\,!u:[\tau_i]_{i\in I}}\,\text{bng}$$

$$\frac{\Gamma\vdash u:\tau}{\Gamma\setminus\!\!\setminus x\vdash\lambda x.u:\Gamma(x)\Rightarrow\tau}\,\text{abs} \qquad \frac{\Gamma_u\vdash u:\sigma \qquad \Gamma_s\vdash s:\Gamma_u(x)}{(\Gamma_u\setminus\!\!\setminus x)+\Gamma_s\vdash u[x\backslash s]:\sigma}\,\text{es} \qquad \frac{\Gamma\vdash u:[\sigma]}{\Gamma\vdash\mathsf{der}(u):\sigma}\,\text{der}$$

Fig. 1. Type System $\mathcal{U}$ for the $\lambda!$-calculus.

different) types in the premises typing u. In particular, when $I=\emptyset$, the rule has no premises, and it types *any* bang $!u$ with the empty multitype, leaving the subterm u *untyped*. Rule (der) forces the argument of a dereliction to be typed by a multitype of cardinality 1.

A **type derivation** in system $\mathcal{U}$ is a tree obtained by successively applying rules in Fig. 1. The typing judgment at the root of the type derivation is the **conclusion** of the derivation. We write $\Pi\triangleright_{\mathcal{U}}\Gamma\vdash t:\sigma$ (or simply $\Pi\triangleright\Gamma\vdash t:\sigma$) if Π is a type derivation in system $\mathcal{U}$ with conclusion $\Gamma\vdash t:\sigma$. A term is called **$\mathcal{U}$-typable** if there exists a derivation $\Pi\triangleright_{\mathcal{U}}\Gamma\vdash t:\sigma$ for some Γ,σ. The **size** $\#(\Pi)$ of a type derivation Π is the number of typing rules different from (bng) in Π. For example, the type derivation Π below has size 2.

$$\Pi=\frac{\dfrac{}{x:[[\,]\Rightarrow\alpha]\vdash x:[\,]\Rightarrow\alpha}\,\text{ax} \qquad \dfrac{}{\emptyset\vdash\,!\Omega:[\,]}\,\text{bng}}{x:[[\,]\Rightarrow\alpha]\vdash x!\Omega:\alpha}\,\text{app} \tag{1}$$

The salient property of system $\mathcal{U}$ is that it characterizes S-normalizing and clash-free terms. Indeed, the term $x!\Omega$ above is S-normalizing (and clash-free) but not F-normalizing.

Theorem 2.1 ([Bucciarelli et al. 2020]).

(1) *Let $\mathcal{R}\in\{\mathsf{S},\mathsf{F}\}$ and $t\to_{\mathcal{R}}t'$. There is $\Pi\triangleright\Gamma\vdash t:\sigma$ iff there is $\Pi'\triangleright\Gamma\vdash t':\sigma$.*
(2) *A term t is $\mathcal{U}$-typable iff t is S-normalizing to a surface clash-free normal form.*

Now that the type system $\mathcal{U}$ for the $\lambda!$-calculus have been presented, we formally define the *inhabitation problem.*

Definition 2.2. A **typing** is a pair $(\Gamma;\sigma)$, where Γ is a type environment and σ is a type. The **inhabitation problem (IP)** on the input typing $(\Gamma;\sigma)$ consists of searching for a term t (if any) typable with $(\Gamma;\sigma)$, *i.e.* such that there is some derivation $\Pi\triangleright\Gamma\vdash t:\sigma$.

The inhabitation problem is not trivial. For instance, typings $(\emptyset;\alpha)$ and $(\emptyset;([\,]\Rightarrow[\,])\Rightarrow[\,])$ are *not inhabited*, *i.e.* there is no derivation Π such that $\Pi\triangleright\emptyset\vdash t:\alpha$ or $\Pi\triangleright\emptyset\vdash t:([\,]\Rightarrow[\,])\Rightarrow[\,]$ for any term t; while the typing $(x:[[\,]\Rightarrow\alpha];\alpha)$ is *inhabited* by $x!\Omega$, as shown by derivation Π in (1).

3 APPROXIMANTS

We aim to provide an algorithm solving the inhabitation problem for any given input typing $(\Gamma;\sigma)$. The algorithm should be able to find *at least* one term typable with $(\Gamma;\sigma)$ in system $\mathcal{U}$, if it exists, and fail otherwise. But we are more ambitious: we aim to recognize *all* (completeness) and *only* (soundness) such terms. Still, given a typing $(\Gamma;\sigma)$, the corresponding *solution set*—the set of terms typable with σ in the environment Γ—is either empty or *infinite*. For example, the term $t_0:=x$ is a solution for the typing $(x:[\tau];\tau)$, and an infinite family $(t_i)_{i\in\mathbb{N}}$ of solutions can be generated by setting $t_{i+1}:=(\lambda z.z)!t_i$. This observation generalizes to any other inhabited typing. Thus, to avoid compromising termination and completeness of the algorithm, our *method* consists in producing for each typing a *finite basis* (if any) from which the whole solution set can be recovered.

A first (naïve) idea would be to consider a basis consisting of *surface* clash-free normal solutions, since for any typing, every solution of the inhabitation problem has a surface clash-free normal form (Thm. 2.1.2) that is typable with the same given typing (Thm. 2.1.1). Unfortunately, this does not solve the issue. Some typings have infinitely many surface clash-free normal solutions, obtained for example by hiding the previous family $(t_i)_{i\in\mathbb{N}}$ under a bang. As redexes under a bang cannot be further S-reduced, surface clash-free normal forms *are not restrictive enough* to obtain a finite basis.

If we instead consider a basis consisting of *full* clash-free normal solutions, we would not be able to recover all solutions: not every typable term has a full clash-free normal form, as witnessed by the typable term $x!\Omega$ in example (1). The problem arises from the fact that typable terms may contain untyped subterms—as the subterm Ω in (1) —that do not necessarily F-normalizes, making full clash-free normal forms *too restrictive* to obtain a finite basis.

By distinguishing between typed and untyped redexes through an appropriate notion of reduction driven by type derivations—and not only by the syntax of terms—we come up with a set of *normal type derivations* that do not contain any typed redex. Terms typed by such derivations—simply called *normal solutions*—do not yet provide a finite basis, however, it suffices to abstract away all their untyped subterms to get the *canonical solutions* that finally provide a finite basis for each possible input for the IP.

3.1 Normal Type Derivations

We introduce the notion of *typed location* distinguishing typed subterms from the untyped ones.

Definition 3.1. The set $\mathsf{l}(t)$ of **locations** of $t \in \Lambda$ is the set of contexts F such that $\mathsf{F}\langle u\rangle = t$ for some u. Given $\Pi_t \triangleright \Gamma \vdash t : \sigma$, the set $\mathsf{tl}(\Pi_t) \subseteq \mathsf{l}(t)$ of **typed locations** of t in Π_t is defined by induction on Π_t (we refer to the names of the rules in Fig. 1):

ax: $\mathsf{tl}(\Pi_t) := \{\diamond\}$;
es: $\mathsf{tl}(\Pi_t) := \{\diamond\} \cup \{\mathsf{F}[x\backslash s] \mid \mathsf{F} \in \mathsf{tl}(\Pi_u)\} \cup \{u[x\backslash \mathsf{F}] \mid \mathsf{F} \in \mathsf{tl}(\Pi_s)\}$ with $t = u[x\backslash s]$, premises Π_u, Π_s;
abs: $\mathsf{tl}(\Pi_t) := \{\diamond\} \cup \{\lambda x.\mathsf{F} \mid \mathsf{F} \in \mathsf{tl}(\Pi_u)\}$ with $t = \lambda x.u$ and premise Π_u;
app: $\mathsf{tl}(\Pi_t) := \{\diamond\} \cup \{\mathsf{F}s \mid \mathsf{F} \in \mathsf{tl}(\Pi_u)\} \cup \{u\mathsf{F} \mid \mathsf{F} \in \mathsf{tl}(\Pi_s)\}$ with $t = us$ and premises Π_u, Π_s;
bng: $\mathsf{tl}(\Pi_t) := \{\diamond\} \cup \bigcup_{i\in I}\{!\mathsf{F} \mid \mathsf{F} \in \mathsf{tl}(\Pi_u^i)\}$ with $t = !u$ and premises $(\Pi_u^i)_{i\in I}$;
der: $\mathsf{tl}(\Pi_t) := \{\diamond\} \cup \{\mathsf{der}(\mathsf{F}) \mid \mathsf{F} \in \mathsf{tl}(\Pi_u)\}$ with $t = \mathsf{der}(u)$ and premise Π_u.

Given a derivation $\Pi \triangleright \Gamma \vdash t : \sigma$, a redex r in t is a **typed redex** in Π if $t = \mathsf{F}\langle r\rangle$ and $\mathsf{F} \in \mathsf{tl}(\Pi)$.

Example 3.2. In the derivation Π in (2) below, the term Ω is not a typed redex in Π (even though Ω is a redex), because $(\lambda x.y)!\Omega = \mathsf{F}\langle\Omega\rangle$ with $\mathsf{F} = (\lambda x.y)!\diamond \notin \mathsf{tl}(\Pi)$. However, the redex $(\lambda x.y)!\Omega$ is a typed redex in Π because $(\lambda x.y)!\Omega = \mathsf{F}\langle(\lambda x.y)!\Omega\rangle$ with $\mathsf{F} = \diamond \in \mathsf{tl}(\Pi)$.

$$\Pi \;=\; \dfrac{\dfrac{\dfrac{}{y : [\sigma] \vdash y : \sigma}\,\mathsf{ax}}{y : [\sigma] \vdash \lambda x.y : [\,] \Rightarrow \sigma}\,\mathsf{abs} \qquad \dfrac{}{\emptyset \vdash !\Omega : [\,]}\,\mathsf{bng}}{y : [\sigma] \vdash (\lambda x.y)!\Omega : \sigma}\,\mathsf{app} \tag{2}$$

Given $\Pi \triangleright \Gamma \vdash t : \sigma$, we say that Π is a **normal (type) derivation**, written $\Pi \triangleright^{\mathsf{nf}} \Gamma \vdash t : \sigma$, if for all $\mathsf{F} \in \mathsf{tl}(\Pi)$, $t = \mathsf{F}\langle u\rangle$ implies that u is not a typed redex in Π. A term s is said to be a **normal solution** for the typing $(\Gamma; \sigma)$ if there exists a normal type derivation Π such that $\Pi \triangleright^{\mathsf{nf}} \Gamma \vdash s : \sigma$. Thus *e.g.* the derivation Π in (1) is normal and the term $x!\Omega$ is a normal solution for $(x : [[\,] \Rightarrow \alpha]\,; \alpha)$. However, Π is not a normal in (2) since $(\lambda x.y)!\Omega$ is a typed redex.

Using Thm. 2.1.1, we now introduce the notion of *typed reduction* on derivations, which only fires *typed* redexes to construct normal derivations. As this is not a syntactic notion, it cannot be defined by case analysis on the term structure. Still, when looking at terms only (*i.e.* the subject of derivations), typed reduction is a special case of full reduction.

Definition 3.3. Let $\Pi \triangleright \Gamma \vdash t : \sigma$ and $t \to_{\mathsf{F}} t'$. Given $\Pi' \triangleright \Gamma \vdash t' : \sigma$ obtained by Thm. 2.1.1, (Π, t) **typed reduces** to (Π', t'), noted $(\Pi, t) \to_{\mathsf{T}} (\Pi', t')$, if there exists a typed location $\mathsf{F} \in \mathsf{tl}(\Pi)$ and two terms u, u' such that $t = \mathsf{F}\langle u\rangle$ and $t' = \mathsf{F}\langle u'\rangle$ with $u \mapsto_{\mathcal{R}} u'$ and $\mathcal{R} \in \{\mathsf{dB}, \mathsf{s!}, \mathsf{d!}\}$.

Example 3.4. Given Π as in (2) and Π', Π'' as below, we have $(\Pi, (\lambda x.y)!\Omega) \rightarrow_{\mathsf{T}} (\Pi', y)$, since $\mathsf{F} = \diamond$ is a typed location in Π and $(\lambda x.y)!\Omega \rightarrow_{\mathsf{F}} y$; but $(\Pi, (\lambda x.y)!\Omega) \not\rightarrow_{\mathsf{T}} (\Pi'', (\lambda x.y)!((zz)[z\backslash!\Delta]))$ because $(\lambda x.y)(!\diamond)$ is not a typed location in Π, even though $(\lambda x.y)!\Omega \rightarrow_{\mathsf{F}} (\lambda x.y)!((zz)[z\backslash!\Delta])$.

$$\Pi' = \dfrac{}{y : [\sigma] \vdash y : \sigma}\,\text{ax} \qquad \Pi'' = \dfrac{\dfrac{\dfrac{}{y : [\sigma] \vdash y : \sigma}\,\text{ax}}{y : [\sigma] \vdash \lambda x.y : [\,] \Rightarrow \sigma}\,\text{abs} \quad \dfrac{}{\emptyset \vdash !((zz)[z\backslash!\Delta]) : [\,]}\,\text{bng}}{y : [\sigma] \vdash (\lambda x.y)!((zz)[z\backslash!\Delta]) : \sigma}\,\text{app}$$

Let us now study the properties of the typed reduction. From Def. 3.3, it follows that $\Pi \triangleright \Gamma \vdash t : \sigma$ is a normal type derivation if and only if there is no (Π', t') such that $(\Pi, t) \rightarrow_{\mathsf{T}} (\Pi', t')$. Moreover, typing reduction decreases the size of the type derivations:

Theorem 3.5 (Weighted Subject Reduction). *If $(\Pi, t) \rightarrow_{\mathsf{T}} (\Pi', t')$, then $\#(\Pi) > \#(\Pi')$. So, $\rightarrow_{\mathsf{T}}$ is strongly normalizing.*

This property yields for any inhabited typing $(\Gamma; \sigma)$ a corresponding normal solution obtained by typed reduction. Thus, normal type derivations already provide an interesting tool to find a basis for each possible input typing. However, we are not completely done yet: a normal solution may still contain *arbitrary* untyped subterms—introduced by the rule (bng) without premises—which can be replaced by any other term, without compromising the normality of the derivation, thus generating *infinitely* many normal solutions. For instance, in the derivation Π in (1), the untyped subterm Ω can be replaced by any other term. The next goal is then to introduce a *canonical* representative for these undesirable subterms in order to obtain a *finite* basis for each typing.

3.2 Approximants of Normal Type Derivations

Since untyped subterms do not carry any typing information, we can represent them by some constants. So, we extend the original term calculus with a set of **constants** $\mathcal{C}$, the resulting set of $\mathcal{C}$**-terms**, denoted by $\Lambda_{\mathcal{C}}$, is inductively defined as follows:

$$a, b ::= \mathsf{c} \in \mathcal{C} \mid x \in \mathcal{X} \mid \lambda x.a \mid ab \mid a[x\backslash b] \mid !a \mid \mathsf{der}(a)$$

In particular, the elements of $\Lambda_\perp := \Lambda_{\{\perp\}}$ are called $\perp$**-terms**. Note that $\Lambda \subsetneq \Lambda_\perp$. The set $\Lambda_\perp$ comes with a **preorder** $\leq$ given by the full contextual closure of $\perp \leq a$ for all $a \in \Lambda_\perp$. For example, $!\perp[z\backslash x\perp] \leq !z[z\backslash x\perp] \leq !z[z\backslash x!y]$. Given a set $\{a_i\}_{i \in I}$ of $\perp$-terms, its **least upper bound**, if any, is denoted by $\bigvee_{i \in I} a_i$ (in particular, $\bigvee_{i \in I} a_i = \perp$ if $I = \emptyset$). We write $\uparrow_{i \in I} a_i$ to state that $\bigvee_{i \in I} a_i$ exists. Thus, *e.g.*, given $a_1 = !\perp[z\backslash x!y]$ and $a_2 = !z[z\backslash x\perp]$, we have $\bigvee_{i \in I} a_i = !z[z\backslash x!y]$ for $I = \{1, 2\}$.

Intuitively, the constant $\perp$ in $\perp$-terms *canonically* represents subterms that are untyped in system $\mathcal{U}$. Formally, we *extend* the typing judgments of system $\mathcal{U}$ (and the notions of normal type derivation and solution) to $\perp$-terms, as expected. So, the rule (bng) used without any premises can introduce a bang $\perp$-term, possibly containing $\perp$ as a subterm, *e.g.* $\dfrac{}{\emptyset \vdash !\perp : [\,]}\,\text{bng}$ or $\dfrac{}{\emptyset \vdash !((\lambda x.\perp)y) : [\,]}\,\text{bng}$.

With each normal derivation Π we can associate a *canonical* representative $\mathcal{A}(\Pi)$, obtained by replacing all maximal untyped subterms of the subject of Π with $\perp$.

Definition 3.6. Given a normal derivation $\Pi \triangleright^{\text{nf}} \Gamma \vdash a : \sigma$, the **approximant** $\mathcal{A}(\Pi)$ of Π is a $\perp$-term defined by induction on Π as follows:

ax :	$\mathcal{A}(\Pi) := x$,	with $a = x$;
es :	$\mathcal{A}(\Pi) := \mathcal{A}(\Pi_b)\,[x\backslash\mathcal{A}(\Pi_c)]$,	with $a = b[x\backslash c]$, and premises Π_b, Π_c;
abs :	$\mathcal{A}(\Pi) := \lambda x.\mathcal{A}(\Pi_b)$,	with $a = \lambda x.b$, and premise Π_b;
app :	$\mathcal{A}(\Pi) := \mathcal{A}(\Pi_b)\,\mathcal{A}(\Pi_c)$,	with $a = bc$, and premises Π_b, Π_c;
bng :	$\mathcal{A}(\Pi) := !\bigvee_{i \in I} \mathcal{A}(\Pi_b^i)$,	with $a = !b$, and premises $(\Pi_b^i)_{i \in I}$;
der :	$\mathcal{A}(\Pi) := \mathsf{der}(\mathcal{A}(\Pi_b))$,	with $a = \mathsf{der}(b)$, and premise Π_b.

Thus, *e.g.*, $\mathcal{A}(\Pi) = x!\bot$ for the normal type derivation Π in (1), as the rule (bng) has no premises. The approximant of normal derivations is always well defined, thanks to the lemma below and since, given $a \in \Lambda_\bot$, $\uparrow_{i\in I} a_i$ holds for every $\{a_i\}_{i\in I} \subseteq \{b \in \Lambda_\bot \mid b \preceq a\}$

Lemma 3.7. *For every $a \in \Lambda_\bot$, if $\Pi \triangleright^{\mathsf{nf}} \Gamma \vdash a : \sigma$ then $\mathcal{A}(\Pi) \preceq a$.*

The set $\Lambda_\bot$ can be produced by a grammar. We thus introduce a general class of grammars with an associated notion of generation.

Definition 3.8. A C**-grammar** $G = (\Sigma_C, S, R, s)$ is a first-order tree grammar whose set of ranked alphabet Σ_C is given by the zero-ary symbols in $C \cup \{\mathsf{Var}\}$, the unary symbols $\mathsf{Der}(\_)$, $\mathsf{Bng}(\_)$ and $\mathsf{Lam}(\_)$, and the binary symbols $\mathsf{App}(\_,\_)$ and $\mathsf{Sub}(\_,\_)$. Such grammars will be used to generate terms with constants. Indeed, a term $a \in \Lambda_C$ is an **instance** of $o \in \mathcal{T}_{\Sigma_C}(S)$, written $c \,\mathcal{I}\, o$, if:

$$\frac{x \in \mathcal{X}}{x \,\mathcal{I}\, \mathsf{Var}} \quad \frac{\mathsf{c} \in C}{\mathsf{c} \,\mathcal{I}\, \mathsf{c}} \quad \frac{a \,\mathcal{I}\, o}{\mathsf{der}(a) \,\mathcal{I}\, \mathsf{Der}(o)} \quad \frac{a \,\mathcal{I}\, o}{!a \,\mathcal{I}\, \mathsf{Bng}(o)} \quad \frac{a \,\mathcal{I}\, o \quad x \in \mathcal{X}}{\lambda x.a \,\mathcal{I}\, \mathsf{Lam}(o)} \quad \frac{a \,\mathcal{I}\, o \quad b \,\mathcal{I}\, p}{ab \,\mathcal{I}\, \mathsf{App}(o,p)} \quad \frac{a \,\mathcal{I}\, o \quad b \,\mathcal{I}\, p \quad x \in \mathcal{X}}{a[x\backslash b] \,\mathcal{I}\, \mathsf{Sub}(o,p)}$$

A C-term $a \in \Lambda_C$ is **produced by a nonterminal symbol** $n \in S$, noted $a \sqsubseteq n$, if there is an $o \in \mathcal{T}_{\Sigma_C}(S)$ such that $n \rightsquigarrow\!\!\!\!\rightsquigarrow o$ and a is an instance of o; and a is **produced** by a C-grammar G, noted $a \sqsubseteq G$, if a is produced by the start symbol s of the grammar G. We set $\mathcal{L}(G) \coloneqq \{a \in \Lambda_C \mid a \sqsubseteq G\}$.

Thus, terms with constants are higher-order terms, seen here as instances of a first-order grammar. From now on, we will only consider (and characterize) $\bot$-terms that are approximants of normal derivations. We define a special grammar B for that.

Definition 3.9. The **grammar** B is a C-grammar where $C = \{\bot\}$, the set of nonterminal symbols is $\{\mathsf{cne}, \mathsf{cna}, \mathsf{cnb}, \mathsf{cno}\}$, the start symbol is cno, and the set of production rules is given below:

$$\begin{array}{ll} \mathsf{cne} \mapsto\!\!\rightsquigarrow \mathsf{Var} \mid \mathsf{App}(\mathsf{cne},\mathsf{cna}) \mid \mathsf{Der}(\mathsf{cne}) \mid \mathsf{Sub}(\mathsf{cne},\mathsf{cne}) & \mathsf{cnb} \mapsto\!\!\rightsquigarrow \mathsf{cne} \mid \mathsf{Lam}(\mathsf{cno}) \mid \mathsf{Sub}(\mathsf{cnb},\mathsf{cne}) \\ \mathsf{cna} \mapsto\!\!\rightsquigarrow \mathsf{cne} \mid \mathsf{Bng}(\mathsf{cno}) \mid \mathsf{Bng}(\bot) \mid \mathsf{Sub}(\mathsf{cna},\mathsf{cne}) & \mathsf{cno} \mapsto\!\!\rightsquigarrow \mathsf{cna} \mid \mathsf{cnb} \end{array}$$

If $a \sqsubseteq \mathsf{cno}$, then a is called a **(clash-free) canonical $\bot$-term**; and if $a \sqsubseteq \mathsf{cne}$, then a is called a **(clash-free) canonical neutral $\bot$-term**.

Example 3.10. We have $zy \sqsubseteq \mathsf{cne}$. Indeed, $zy \,\mathcal{I}\, \mathsf{App}(\mathsf{Var},\mathsf{Var})$ and

$$\mathsf{cne} \rightsquigarrow \mathsf{App}(\mathsf{cne},\mathsf{cna}) \rightsquigarrow \mathsf{App}(\mathsf{cne},\mathsf{cne}) \rightsquigarrow \mathsf{App}(\mathsf{Var},\mathsf{cne}) \rightsquigarrow \mathsf{App}(\mathsf{Var},\mathsf{Var}).$$

Similarly, it can be shown that $(yx)[x\backslash zy] \sqsubseteq \mathsf{cne}$, $(\lambda x.x)[x'\backslash zy] \sqsubseteq \mathsf{cno}$ and $(\lambda x.x)[x'\backslash zy] \sqsubseteq \mathsf{cnb}$.

The grammar B, as well as the notion of production, will be crucial concepts in Sects. 5 and 6.

Proposition 3.11. *A $\bot$-term a is canonical iff a is the approximant of some normal derivation.*

A normal type derivation is a **canonical derivation** if it types its approximant, *i.e.* if it is of the form $\Pi \triangleright^{\mathsf{nf}} \Gamma \vdash a : \sigma$ with $a = \mathcal{A}(\Pi)$; we then say that a is a **canonical solution** for the typing $(\Gamma; \sigma)$. Thus *e.g.* the derivation Π in (1) is normal but not canonical, while the derivation Π' below is canonical, and $\mathcal{A}(\Pi') = x!\bot$ is a canonical solution for the typing $(x : [[\,] \Rightarrow \alpha]\,; \alpha)$.

$$\Pi' = \frac{\dfrac{}{x : [[\,] \Rightarrow \alpha] \vdash x : [\,] \Rightarrow \alpha}\,\mathsf{ax} \qquad \dfrac{}{\emptyset \vdash !\bot : [\,]}\,\mathsf{bng}}{x : [[\,] \Rightarrow \alpha] \vdash x!\bot : \alpha}\,\mathsf{app} \tag{3}$$

The approximant of a normal derivation—deriving a normal solution for some typing—is indeed a canonical solution for such a typing:

Proposition 3.12 (Canonicity of approximants). *Let $\Pi \triangleright^{\mathsf{nf}} \Gamma \vdash a : \sigma$ with $a \in \Lambda_\bot$. Then, $\mathcal{A}(\Pi)$ is a canonical solution for the typing $(\Gamma; \sigma)$.*

Summing up, each solution to the inhabitation problem for a given typing can be represented by a *canonical* solution. Indeed, a **basis** for a typing $(\Gamma; \sigma)$ is the set:

$$\mathtt{Basis}(\Gamma; \sigma) \coloneqq \{b \in \Lambda_\perp \mid b \text{ canonical solution for } (\Gamma; \sigma)\}$$

Next theorem guarantees that $\mathtt{Basis}(\Gamma; \sigma)$ *generates* the whole solution set to the inhabitation problem for the typing $(\Gamma; \sigma)$. To formalize this notion, we define the **span** of a set $S \subseteq \Lambda_\perp$ as the set of terms obtained by full expansion of redexes of terms greater than the elements of S:

$$\mathsf{Span}(S) \coloneqq \{t \in \Lambda \mid \exists\, a \in S,\ \exists\, u \in \Lambda,\ a \leq u \text{ and } t \twoheadrightarrow_{\mathsf{F}} u\}$$

By defining the **solution set** to the IP for the typing input $(\Gamma; \sigma)$ as $\mathtt{Sol}(\Gamma; \sigma) \coloneqq \{t \in \Lambda \mid \exists\, \Pi \triangleright \Gamma \vdash t : \sigma\}$, we conclude with a crucial property of our basis:

Theorem 3.13 (Sound & complete basis). *For every typing* $(\Gamma; \sigma)$, $\mathsf{Span}(\mathtt{Basis}(\Gamma; \sigma)) = \mathtt{Sol}(\Gamma; \sigma)$.

Thm. 3.13 gives the key argument to the *completeness* property of our inhabitation algorithm (Sect. 5). Also, termination of our algorithm (Cor. 5.14) entails *finiteness* of the basis for every typing: this is why it suffices to represent the solutions to the inhabitation problem by *canonical* solutions.

4 SUBTYPE SEARCH

Using the results in Sect. 3, we know now how to restrict the solution set to a *finite* basis without loosing completeness: we must consider (the approximant of) canonical type derivations. We now wish to build an IP algorithm able to find all such normal type derivations, but this is not immediate. In particular, some types needed for the recursive calls cannot be *simply* deduced from the given input typing, since (even normal) derivations may not have the subformula property. Let us take for example the application rule app in Fig. 1. The type $\mathcal{M}$ of the argument u appears in both premises but does not seem, in principle, to be present in the given input typing. A naive algorithm would try to make recursive calls with every possible multitype $\mathcal{M}$, but this would break termination. A (terminating) algorithm requires a subtler mechanism.

In this section we first focus on the *head subtype property*, which links, by means of a subtyping relation, the conclusion type of any type derivation of a canonical neutral $\perp$-term to its typing environment. We then propose a terminating algorithm computing all possible subtypes of a given type. This algorithm is designed so that the search for subtypes can be guided by some *partial knowledge* about their forms. This will notably be used for the application and explicit substitution cases of the inhabitation algorithm.

4.1 Head Subtype Property

Canonical neutral $\perp$-terms (Def. 3.9) are built from variables, applications, derelictions and ES. In the last three corresponding typing rules (app, der and es in Fig. 1), the conclusion type is contained in the type of its left (or unique) premise. By construction, the subject of this left premise is also a canonical neutral term, and thus, by induction, one has that the conclusion type of the associated derivation is contained in the type of its leftmost axiom (typing the leftmost variable occurrence, called the *syntactic head*). By definition of the axiom rule, the type of this variable is also contained in its typing environment. However, the syntactic head of a canonical neutral $\perp$-term may be captured by some ES, so its type may be erased from the environment. Hence, the conclusion type of a derivation does not seem, at first sight, to necessarily appear in its typing environment.

Fortunately, the relation between the type of the conclusion and the typing environment can be restored thanks to the notion of *semantic head variable*: it is the *leftmost free* variable modulo unfolding. We then consider a notion of *semantic head type* of a derivation Π which is the type given by Π to the semantic head variable of its subject. It turns out that the semantic head type appears in the environment of the conclusion, which finally solves our problem.

Definition 4.1 (Semantic head variable/type). Let $a \Subset \mathsf{cne}$ be a canonical neutral $\bot$-term. Its **semantic head variable** $\mathsf{shv}(a) \in \mathsf{fv}(a)$ is defined by induction as follows:

$$\mathsf{shv}(x) := x \quad \mathsf{shv}(\mathsf{der}(u)) := \mathsf{shv}(u) \quad \mathsf{shv}(tu) := \mathsf{shv}(t) \quad \mathsf{shv}(t[x\backslash u]) := \mathsf{shv}(t)\{x\backslash \mathsf{shv}(u)\}.$$

The **semantic head type** $\mathsf{sht}(\Pi)$ of a $\Pi \triangleright \Gamma \vdash a : \sigma$ with $a \Subset \mathsf{cne}$ is defined by induction on Π:

$$\begin{array}{rlll}
\mathsf{ax}: & \mathsf{sht}(\Pi) := \sigma; & & \\
\mathsf{es}: & \mathsf{sht}(\Pi) := \begin{cases} \mathsf{sht}(\Pi_c) & \text{if } \mathsf{shv}(b) = x, \\ \mathsf{sht}(\Pi_b) & \text{otherwise,} \end{cases} & & \text{with } a = b[x\backslash c] \text{ and premises } \Pi_b, \Pi_c; \\
\mathsf{app}: & \mathsf{sht}(\Pi) := \mathsf{sht}(\Pi_b), & & \text{with } a = bc \text{ and premises } \Pi_b, \Pi_c; \\
\mathsf{der}: & \mathsf{sht}(\Pi) := \mathsf{sht}(\Pi_b), & & \text{with } a = \mathsf{der}(b) \text{ and premise } \Pi_b.
\end{array}$$

For instance, if $\Pi \triangleright y : [[\,] \Rightarrow \sigma] \vdash x[x\backslash y]z : \sigma$, then $\mathsf{shv}(t) = y$ and $\mathsf{sht}(\Pi) = [\,] \Rightarrow \sigma$.

The following notion of **subtype** will be used to capture the relation between the type of the conclusion of a derivation and its environment (forthcoming Lem. 4.3).

Definition 4.2. The **subtype relation** $\preceq$ is a binary relation on types defined by the rules below.

$$\frac{\tau = \sigma}{\tau \preceq \sigma}\,\mathsf{refl} \qquad\qquad \frac{\tau \preceq \mathcal{M} \quad or \quad \tau \preceq \rho}{\tau \preceq (\mathcal{M} \Rightarrow \rho)}\,\mathsf{arrow} \qquad\qquad \frac{\exists\, j \in I,\ \tau \preceq \rho_j}{\tau \preceq [\rho_i]_{i \in I}}\,\mathsf{mult}.$$

It can be shown that $\preceq$ is a non-strict partial order. The equality in the rule (refl) is considered modulo associativity and commutativity of the multitype constructor, *e.g.*

$$[\,] \Rightarrow [\rho_2, \rho_1] \ \preceq\ [[\,] \Rightarrow \alpha, [[\,] \Rightarrow [\rho_1, \rho_2]] \Rightarrow \alpha'] \tag{4}$$

Using the notions of semantic head variable and type, as well as the subtype relation, we can now state a fundamental property to make the inhabitation algorithm decidable.

Lemma 4.3 (Head Subtype). *Let* $\Pi \triangleright \Gamma \vdash a : \sigma$ *with* $a \Subset \mathsf{cne}$*, then* $\sigma \preceq \mathsf{sht}(\Pi)$ *and* $\Gamma(\mathsf{shv}(a)) = \mathcal{M} \uplus [\mathsf{sht}(\Pi)]$ *for some* $\mathcal{M}$.

We will see in Sec. 5 that the subtype relation plays a crucial role in the inhabitation algorithm. We thus design a subalgorithm computing all the subtypes of a given type.

4.2 Subtype Search

We now introduce an algorithm to search for all the subtypes of a given type, in a general form. If we seek for a solution having the form of an application tu, we then search for a multitype $\mathcal{M}$ that not only is the type of the right premise typing u, but also appears as a subtype of the type of the left premise typing t (*cf.* rule app in Fig. 1). We will therefore generalize our subtype search problem to the search of subtypes having a *specific shape*. This specific shape is denoted using **partial types**, which are types containing placeholders $\diamond_1, \ldots, \diamond_n$. Given a partial type p with $n \geq 0$ placeholders and a list of types $\tau_1, \ldots, \tau_n$, we write $p\langle \tau_1, \cdots, \tau_n \rangle$ for the type obtained by replacing each placeholder $\diamond_i$ by the type τ_i. For example, if $p = [\diamond_1, \sigma] \Rightarrow \diamond_2$ then $p\langle \tau_1, \tau_2 \rangle = [\tau_1, \sigma] \Rightarrow \tau_2$.

Definition 4.4. Let τ, σ be types and p a partial type. Then τ is a **subtype of** σ **with partial knowledge** p, noted $\tau \preceq_p \sigma$, if $\tau \preceq \sigma$ and $\tau = p\langle \tau_1, \cdots, \tau_n \rangle$ for some types $\tau_1, \ldots, \tau_n$.

Testing for subtype with a specific shape (i.e. partial knowledge) amounts to *matching*. For instance, coming back to example (4), $[\,] \Rightarrow [\rho_2, \rho_1] \preceq_{\diamond \Rightarrow [\rho_1, \rho_2]} [[\,] \Rightarrow \alpha, [[\,] \Rightarrow [\rho_1, \rho_2]] \Rightarrow \alpha']$.

Given a term σ and a partial type p, the **Subtype with Partial Knowledge (SPK) algorithm** yields a type τ such that $\tau \preceq_p \sigma$ if such a type exists, and otherwise fails. When the partial shape is empty, *i.e.* $p = \diamond$, the SPK algorithm just yields a subtype of the input σ. The rules of the SPK algorithm are presented in Fig. 2: the notation $\tau \Vdash S(\sigma, p)$ is formed of a *call* $S(\sigma, p)$ with input

$$\frac{\exists \tau_1, \ldots, \tau_n,\ \ \sigma = p\langle \tau_1 \cdots \tau_n \rangle}{p\langle \tau_1 \cdots \tau_n \rangle \Vdash\!\vdash S(\sigma, p)}\text{match} \quad \frac{\exists i \in I,\ \tau \Vdash\!\vdash S(\sigma_i, p)}{\tau \Vdash\!\vdash S([\sigma_i]_{i \in I}, p)}\text{bag} \quad \frac{\tau \Vdash\!\vdash S(\mathcal{M}, p)}{\tau \Vdash\!\vdash S(\mathcal{M} \Rightarrow \sigma, p)}{\Rightarrow_1} \quad \frac{\tau \Vdash\!\vdash S(\sigma, p)}{\tau \Vdash\!\vdash S(\mathcal{M} \Rightarrow \sigma, p)}{\Rightarrow_2}$$

Fig. 2. Rules of the SPK Algorithm.

(σ, p) and an *answer* τ to this call. In every run of the algorithm, the call of the lower part of each rule generates the recursive calls of the upper part of the corresponding rule. Once a match has been finally done, the answer travels downwards to the final rule.

Rules $(\Rightarrow_1)$, $(\Rightarrow_2)$ and (bag) correspond to an immediate subtype. Rule (match) corresponds to the reflexivity closure of the relation, taking into account the matching of the input type σ with respect to the parameter p. As in the (refl) rule (Def. 4.2), equality is considered modulo commutativity and associativity of the multitype constructor. Coming back to example (4), if $\sigma = [[\,] \Rightarrow \alpha, [[\,] \Rightarrow [\rho_1, \rho_2]] \Rightarrow \alpha']$, one has: $[\,] \Rightarrow [\rho_2, \rho_1] \Vdash\!\vdash S(\sigma, \diamond \Rightarrow [\rho_1, \rho_2])$.

Different runs of the algorithm may yield different solutions, *e.g.* if $\sigma_0 = [\alpha] \Rightarrow \alpha'$, three runs are all possible from the same call: $[\alpha] \Rightarrow \alpha' \Vdash\!\vdash S(\sigma_0, \diamond)$ and $\alpha \Vdash\!\vdash S(\sigma_0, \diamond)$ and $\alpha' \Vdash\!\vdash S(\sigma_0, \diamond)$.

As expected, the SPK algorithm is *sound* and *complete*:

Lemma 4.5 (Soundness and completeness of SPK). *Let τ, σ be types and p be a partial type. Then, $\tau \preceq_p \sigma$ if and only if $\tau \Vdash\!\vdash S(\sigma, p)$.*

The SPK algorithm is non-deterministic. Indeed, to compute all subtypes of a given type, every possible run is executed. We then need to show that there is a finite number of possible immediate runs (*finite degree*), which is straightforward, and that each of these runs terminates (*finite depth*). To prove finite depth, we consider a positive measure on types called **constructor size**, given by:

$$\mathrm{sz}(\alpha) := 1 \qquad \mathrm{sz}([\sigma_i]_{i \in I}) := \textstyle\sum_{i \in I} \mathrm{sz}(\sigma_i) + 1 \qquad \mathrm{sz}(\mathcal{M} \Rightarrow \sigma) := \mathrm{sz}(\mathcal{M}) + \mathrm{sz}(\sigma) + 1.$$

Lemma 4.6 (Termination of SPK). *The SPK algorithm terminates. More precisely:*

(1) *Finite depth: Every run of the algorithm terminates.*
(2) *Finite degree: For any possible input, the set of all possible immediate recursive calls is finite.*

Thus, from a call $S(\sigma, p)$, algorithm SPK can compute all subtypes of σ with partial knowledge p.

5 THE INHABITATION ALGORITHM

This section is devoted to algorithm $\mathbf{Inh}_{\mathcal{U}}$ solving the IP for system $\mathcal{U}$. We first discuss the general form of its rules (Sect. 5.1), then we present $\mathbf{Inh}_{\mathcal{U}}$ and give some execution examples (Sect. 5.2). We state termination, soundness and completeness for $\mathbf{Inh}_{\mathcal{U}}$ (Sect. 5.3). We then introduce a more abstract version of the algorithm (Sect. 5.4), called $\mathbf{Inh}^{G}_{\mathcal{U}}$, whose executions are controlled through a parametric grammar G. Termination, soundness and completeness are preserved by $\mathbf{Inh}^{G}_{\mathcal{U}}$ (Sect. 5.5).

5.1 General Form of the Algorithm Rules

The $\mathbf{Inh}_{\mathcal{U}}$ algorithm aims to build any possible type derivation for a given typing. Due to the properties shown in Sect. 3, $\mathbf{Inh}_{\mathcal{U}}$ only returns canonical $\perp$-terms, since Thm. 3.13 ensures that this is sufficient to span the whole solution set. Our algorithm $\mathbf{Inh}_{\mathcal{U}}$ is then intimately associated with *grammar* $\mathtt{B}$ from Def. 3.9, which produces all possible canonical $\perp$-terms. We now give an informal and introductory presentation of the algorithm, focusing on the following salient features.

Tree Structure. Type derivations are made of typing rules assembled into a tree structure, so the inhabitation algorithm is written in a similar spirit: each rule of the typing system $\mathcal{U}$ has at least a corresponding rule in $\mathbf{Inh}_{\mathcal{U}}$, declined in variants as we will explain. The algorithm builds a tree having the rules of $\mathbf{Inh}_{\mathcal{U}}$ for (hyper) edges and sequents of the form $a \Vdash \mathtt{Call}$ for nodes, where

$$\frac{\begin{array}{l|l}\text{Requirement}_1 & \\ \vdots & \\ \text{Requirement}_n & \quad a_1 \Vdash \texttt{Call}_1 \quad \cdots \quad a_m \Vdash \texttt{Call}_m\end{array}}{\odot(a_1, \ldots, a_m) \Vdash \texttt{Call}}$$

Fig. 3. Pattern of the $\mathbf{Inh}_{\mathcal{U}}$ Algorithm Rules

$\texttt{Call}$ is any kind of **call** (different kinds of calls are detailed later), and a is an *answer* (a canonical $\perp$-term) to the call. Such a tree is built by the algorithm search in two subsequent stages:

- *bottom-up* on the right-hand sides ($\uparrow$ in Fig. 3): from a node of the form $\_ \Vdash \texttt{Call}$, corresponding to the conclusion of some rule of the algorithm (meaning that $\mathbf{Inh}_{\mathcal{U}}$ is searching for an answer for the call $\texttt{Call}$), the algorithm performs *recursive calls* $\_ \Vdash \texttt{Call}_1, \ldots, \_ \Vdash \texttt{Call}_n$ —the premises of the corresponding algorithm rule;
- *top-down* on the left-hand sides ($\downarrow$ in Fig. 3): once the tree of recursive calls is completed, the algorithm computes the answer by going down from the leaves to the root; the conclusion answer $\odot(a_1, \ldots, a_m)$ for the call $\texttt{Call}$, written $\odot(a_1, \ldots, a_m) \Vdash \texttt{Call}$, where $\odot$ is any term constructor of $\lambda!$, is built from the premises answers $a_1 \Vdash \texttt{Call}_1, \ldots, a_m \Vdash \texttt{Call}_n$.

Moreover, at any node, the set of applicable rules is regulated by some preliminary requirements. The general form of the rules of the algorithm $\mathbf{Inh}_{\mathcal{U}}$ (Fig. 4) follows the pattern in Fig. 3, with $n, m \geq 0$ (if $n = 0$ or $m = 0$, no preliminary requirement or recursive call is written in the rule, respectively).

Preliminary Requirements. Each rule of $\mathbf{Inh}_{\mathcal{U}}$ may need some *preliminary requirements*, which are specified by formulas existentially quantifying some elements, marked in red (see Fig. 4). These requirements are independent of the answer and, for each rule, they must be checked *before* any recursive call of the bottom-up stage ($\uparrow$ in Fig. 3). For example, the rule (N_P-H) in Fig. 4 has two requirements, the first one demands the existence of *some* splitting of the environment $\Gamma = \Gamma' + x : [\tau]$, and the last one requires a subtype verification $\sigma \Vvdash S(\tau, \diamond)$ with respect to the type τ found in the first requirement.

Non-Determinism. Given an input for the algorithm, *different* searches are possible. First, it cannot be *uniquely* determined which rule has to be applied at each step of the search. Second, even when the rule to be used is chosen, it is not always possible to *uniquely* decompose the inputs to build the inhabitation subproblems of the recursive searches. Which rule or decomposition to choose is generally unknown and several combinations may yield different answers, so that all of them should be tried. Last but not least, the subtype search of the SPK algorithm (Sect. 4) used by $\mathbf{Inh}_{\mathcal{U}}$ may also produce multiple answers. Just as above, all combinations have to be tested. Producing all possible answers is therefore achieved by constructing all possible searches out of the set of *non-deterministic* rules presented in Fig. 4.

Two Kinds of Recursive Call. The algorithm $\mathbf{Inh}_{\mathcal{U}}$ has two different kinds of **calls**, generically denoted $\texttt{Call}$: they are either N*-calls* of the form $N(\Gamma; \sigma)$, $N_A(\Gamma; \sigma)$, or $N_B(\Gamma; \sigma)$, or H*-calls* of the form $H^{x:[\tau]}(\Gamma; \sigma)$. The three former calls (resp. latter call) correspond to the search for a solution to the IP with input $(\Gamma; \sigma)$ (resp. $(\Gamma + x : [\tau]\,; \sigma)$). In H-calls, a specific variable x and type $[\tau]$ is picked out of the type environment to act as a search hint. Motivated by Sect. 4, these hints will set the semantic head of the solution. In fact, N-calls correspond to the search of solutions generated by the nonterminal symbol cno of grammar B, and similarly for N_A, N_B and H with respect to the symbols cna, cnb and cne, respectively. Some rules are represented parametrically using a generic call N_P with $P \in \{A, B\}$, to denote either N_A or N_B.

$$\frac{\mid}{x \Vdash H^{x:[\sigma]}(\emptyset;\sigma)}\text{VAR} \qquad \frac{[\sigma] \Vdash\!\!\Vdash S(\tau,\diamond) \mid a \Vdash H^{x:[\tau]}(\Gamma;[\sigma])}{\text{der}(a) \Vdash H^{x:[\tau]}(\Gamma;\sigma)}\text{DR} \qquad \frac{\Gamma = \Gamma' + x:[\tau] \;\; \sigma \Vdash\!\!\Vdash S(\tau,\diamond) \mid a \Vdash H^{x:[\tau]}(\Gamma';\sigma)}{a \Vdash N_P(\Gamma;\sigma)}\text{N}_P\text{-H}$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b \;\; \mathcal{M} \Rightarrow \sigma \Vdash\!\!\Vdash S(\tau, \diamond \Rightarrow \sigma) \mid a \Vdash H^{x:[\tau]}(\Gamma_a; \mathcal{M} \Rightarrow \sigma) \quad b \Vdash N_A(\Gamma_b; \mathcal{M})}{ab \Vdash H^{x:[\tau]}(\Gamma;\sigma)}\text{APP} \qquad \frac{\mid \; a \Vdash N_P(\Gamma;\sigma)}{a \Vdash N(\Gamma;\sigma)}\text{N-N}_P$$

$$\frac{\texttt{fix}\ x \notin \text{dom}(\Gamma) \mid a \Vdash N(\Gamma, x:\mathcal{M};\sigma)}{\lambda x.a \Vdash N_B(\Gamma;\mathcal{M} \Rightarrow \sigma)}\text{ABS} \qquad \frac{I \neq \emptyset \;\; \Gamma = +_{i\in I}\Gamma_i \mid (a_i \Vdash N(\Gamma_i;\tau_i))_{i\in I} \quad \uparrow_{i\in I} a_i}{!\bigvee_{i\in I} a_i \Vdash_g N_A(\Gamma;[\tau_i]_{i\in I})}\text{BG} \qquad \frac{\mid}{!\bot \Vdash_g N_A(\emptyset;[\,])}\text{BG}_\bot$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b + z:[\rho], \;\; \texttt{fix}\ y \notin \text{dom}(\Gamma) \cup \{x\}, \;\; n \in [\![0, \text{sz}(\rho)]\!], \;\; \mathcal{M} \Vdash\!\!\Vdash S(\rho, [\diamond_1,\ldots,\diamond_n]) \mid a \Vdash H^{x:[\tau]}(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash H^{z:[\rho]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash H^{x:[\tau]}(\Gamma;\sigma)}\text{ES-H}$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b, \;\; \texttt{fix}\ y \notin \text{dom}(\Gamma) \cup \{x\} \;\; n \in [\![1, \text{sz}(\tau)]\!], \; [\rho_i]_{i\in[\![1,n]\!]} \Vdash\!\!\Vdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \;\; j \in [\![1,n]\!], \; \sigma \Vdash\!\!\Vdash S(\rho_j,\diamond) \mid a \Vdash H^{y:[\rho_j]}(\Gamma_a, y:[\rho_i]_{i\in[\![1,n]\!]\backslash j};\sigma) \quad b \Vdash H^{x:[\tau]}(\Gamma_b;[\rho_i]_{i\in[\![1,n]\!]})}{a[y\backslash b] \Vdash H^{x:[\tau]}(\Gamma;\sigma)}\text{ES-CH}$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b + z:[\tau], \;\; \texttt{fix}\ y \notin \text{dom}(\Gamma), \;\; n \in [\![0, \text{sz}(\tau)]\!], \;\; \mathcal{M} \Vdash\!\!\Vdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \mid a \Vdash N_P(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash H^{z:[\tau]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash N_P(\Gamma;\sigma)}\text{ES-N}_P$$

Fig. 4. Rules of the Algorithm $\mathbf{Inh}_{\mathcal{U}}$ for System $\mathcal{U}$
($P \in \{A, B\}$ in rules (N_P-H), (N-N_P) and (ES-N_P))

Run. Given an input typing $(\Gamma;\sigma)$, a **run** of $\mathbf{Inh}_{\mathcal{U}}$ is a tree of recursive calls starting from the root $N(\Gamma;\sigma)$; its (hyper) edges are the rules in Fig. 4. A run is built bottom-up by making a particular choice among the non-deterministic ones discussed above (including the existentially quantified elements in red). When no more recursive calls are possible, the following conditions have to be fulfilled to obtain a valid run and compute the answer (at the top-down stage):

- all the leaves of the tree correspond to rule (VAR) or ($\text{BG}_\bot$), the only ones with no premises,
- in all the instances of the rule (BG), the least upper bound of all recursive answers ($\uparrow_{i\in I} a_i$) exists (this is necessarily checked at the top-down stage, *after* all recursive calls).

Otherwise, the run is considered failed. Notice that for a same input, there may be different runs generated by different non-deterministic choices. Some of these runs may fail while others may succeed. Several examples are given in Example 5.1.

5.2 The Algorithm

The rules of the $\mathbf{Inh}_{\mathcal{U}}$ algorithm for the type system $\mathcal{U}$ are presented in Fig. 4, where we use the notation $[\![m,n]\!] := \{m, m+1, \ldots, n\}$ for all $m, n \in \mathbb{N}$ with $m \le n$.

We chose H-calls $H^{x:[\tau]}(\Gamma;\sigma)$ to represent searches for solutions with a specific semantic head $x : [\tau]$. The rule (N_P-H) converts an N-call into an H-call and its requirements are therefore justified by Lem. 4.3: the recursive call is done with a chosen head which must contain as subtype the input type σ. First, a variable x must be isolated in the environment Γ—this is done by the splitting requirement $\Gamma = \Gamma' + x : [\tau]$—so that in a second step the subtype check is performed on the type of this variable x.

Rule (ABS) searches for an inhabitant that is an abstraction. By α-conversion, the bound variable chosen in the answer is arbitrary fixed once and for all outside the domain of its environment Γ.

Rules (ES-H) and (ES-CH) correspond to the search for an inhabitant $a[y\backslash b]$ with a given semantic head $x : [\tau]$. Here again, the bound variable y is arbitrary fixed once and for all. By construction, the semantic head is either in a (left premise) or b (right premise). The rule is therefore duplicated to handle both possibilities.

More precisely, rule (ES-H) treats the case where the left semantic head is not captured by the substitution. The left premise therefore shares the semantic head with that of the main call (in the conclusion) and a new semantic head is required for the right premise. Again by Lem. 4.3, this new semantic head has to be contained in the input type environment. This is done by the splitting requirement $\Gamma = \Gamma_a + \Gamma_b + z : [\rho]$, where the variable z may also appear in the premises environments Γ_a and Γ_b The type $\mathcal{M}$ of the right premise is however unknown and has to be deduced: it is a—possibly empty—multitype which, by Lem. 4.3, is contained in the newly chosen semantic head z. We can find it using the requirement $n \in [\![0, \mathrm{sz}(\rho)]\!]$, $\mathcal{M} \Vdash S(\rho, [\diamond_1, \ldots, \diamond_n])$. Note that if $n = 0$, the requirement becomes $\mathcal{M} \Vdash S(\rho, [\,])$.

Rule (ES-CH) treats the case where the semantic head of the left premise is captured by the substitution. Thus, the right premise shares the semantic head with the main call (in the conclusion). Similarly to (ES-H), the multitype $[\rho_i]_{i\in[\![1,n]\!]}$ in the right premise has to be deduced before the recursive call. However, since the head is captured, we seek a *non-empty* multitype. So, the search is implemented by the requirement $n \in [\![1, \mathrm{sz}(\tau)]\!]$, $[\rho_i]_{i\in[\![1,n]\!]} \Vdash S(\tau, [\diamond_1, \ldots, \diamond_n])$. The head of the left premise is then selected as one of the elements of the newly obtained multiset. The last requirement checks the head subtype property on the chosen type $j \in [\![1, n]\!]$, $\sigma \Vdash S(\rho_j, \diamond)$.

The typing (bng) rule of system $\mathcal{U}$ has two corresponding rules in $\mathbf{Inh}_{\mathcal{U}}$: a special rule ($\mathrm{BG}_\perp$) handling the case where the constant $\perp$ is introduced, and another rule (BG) dealing with the non-empty cases. Note that in the (bng) rule, the least upper bound of the approximants of all premises always exists (see page 10), whereas in $\mathbf{Inh}_{\mathcal{U}}$ the recursive calls do not guarantee this property. This is why (BG) explicitly requires $\uparrow_{i\in I} a_i$. The other requirement splits the environment as expected.

Example 5.1. Let us see an example of the IP in $\lambda!$ using the algorithm $\mathbf{Inh}_{\mathcal{U}}$ for the input typing $(x : [[[\alpha]]]\,;\alpha)$. All the solutions given by the algorithm are:

$$\mathrm{der}(\mathrm{der}(x)) \quad \mathrm{der}(y)[y\backslash x] \quad \mathrm{der}(y[y\backslash x]) \quad z[z\backslash y][y\backslash x] \quad y[y\backslash \mathrm{der}(x)] \quad z[z\backslash y[y\backslash x]].$$

We describe a particular run of the algorithm finding the solution $\mathrm{der}(\mathrm{der}(x))$ from the starting call $N(x : [[[\alpha]]]\,;\alpha)$. The run starts by applying rule (N-$\mathrm{N_A}$), followed by rule ($\mathrm{N_A}$-H). In the latter, the head $x : [[[\alpha]]]$ is chosen from the environment and the subtype inequation $\alpha \preceq [[[\alpha]]]$ is checked, which yields the call $H^{x:[[[\alpha]]]}(\emptyset;\alpha)$. Then, rule (DR) is applied twice, with the corresponding subtype verifications $[\alpha] \preceq [[[\alpha]]]$ and $[[\alpha]] \preceq [[[\alpha]]]$. Finally, rule (VAR) is applied and the solution $\mathrm{der}(\mathrm{der}(x))$ is built, by composing the answers of the previous rules in a top-down way.

Notice that if the second use of the (DR) rule were replaced by the (APP) rule, its second requirement would have failed, since one cannot find a type $\mathcal{M}$ such that $\mathcal{M} \Rightarrow [\alpha] \preceq [[[\alpha]]]$, and therefore the entire run would have failed.

We now describe another particular run of the algorithm finding the solution $y[y\backslash \mathrm{der}(x)]$ from the starting call $N(x : [[[\alpha]]]\,;\alpha)$. The run starts by applying rule (N-$\mathrm{N_A}$), followed by rule (ES-N_A). In the latter, the environment is split into two empty environments, plus a head $x : [[[\alpha]]]$. The subtype inequation $\mathcal{M} \preceq [[[\alpha]]]$ is solved by taking $\mathcal{M} = [\alpha]$. Two calls are then carried out separately: on the one hand, a fresh variable y is fixed and a call $N_A(y : [\alpha]\,;\alpha)$ is carried out. By applying rules ($\mathrm{N_A}$-H) and (VAR), the algorithm yields y as a solution to this first recursive call. On the other hand, the call $H^{x:[[[\alpha]]]}(\emptyset;[\alpha])$ is performed. By applying rules (DR) and (VAR), the algorithm yields $\mathrm{der}(x)$ as a solution to this second recursive call. Finally, both solutions are assembled using an explicit substitution to build the solution $y[y\backslash \mathrm{der}(x)]$ for the initial call.

5.3 Properties of the Algorithm $\mathbf{Inh}_{\mathcal{U}}$

The first crucial property of algorithm $\mathbf{Inh}_{\mathcal{U}}$ is the following one:

Theorem 5.2 (Termination of $\mathbf{Inh}_{\mathcal{U}}$). *The algorithm $\mathbf{Inh}_{\mathcal{U}}$ is terminating.*

We postpone however the proof to Sect. 5.5, as Thm. 5.2 is a consequence of Cor. 5.14, stating termination for the parametric algorithm $\mathbf{Inh}^{G}_{\mathcal{U}}$. As for now, we discuss soundness and completeness.

The *actual answers* (outputs) of the $\mathbf{Inh}_{\mathcal{U}}$ algorithm for a typing input $(\Gamma;\sigma)$, written $\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)$, are sound and complete for the IP in a twofold sense: with respect to the *expected answers* of $\mathbf{Inh}_{\mathcal{U}}$ on $(\Gamma;\sigma)$, which is given by the set $\mathtt{Basis}(\Gamma;\sigma)$, and with respect to the whole *solution set* to the IP on $(\Gamma;\sigma)$, given by $\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)$, both notions coming from Sect. 3.2.

We prove that actual and expected answers coincide:

Theorem 5.3 ($\mathbf{Inh}_{\mathcal{U}}$ is sound & complete). *For any typing $(\Gamma;\sigma)$, $\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma) = \mathtt{Basis}(\Gamma;\sigma)$.*

The outputs of $\mathbf{Inh}_{\mathcal{U}}$ "span" the solution set for the IP on a typing input.

Corollary 5.4 (Sound & complete solution to IP). *For any typing $(\Gamma;\sigma)$, $\mathtt{Span}(\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)) = \mathtt{Sol}(\Gamma;\sigma)$.*

Proof. We have $\mathtt{Span}(\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)) =_{\mathit{Thm.}\ 5.3} \mathtt{Span}(\mathtt{Basis}(\Gamma;\sigma)) =_{\mathit{Thm.}\ 3.13} \mathtt{Sol}(\Gamma;\sigma)$. □

To summarize, the output of the algorithm $\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)$ yields a compact representation of all possible terms typable with the given input $(\Gamma;\sigma)$. Thus, $\mathbf{Inh}_{\mathcal{U}}(\Gamma;\sigma)$ can also be queried for emptiness, and moreover, one can check whether a given term is among those represented.

5.4 A (Parametric) Abstract Approach

The $\mathbf{Inh}_{\mathcal{U}}$ algorithm for system $\mathcal{U}$ is intimately linked to grammar $\mathtt{B}$ (Def. 3.9) of canonical solutions: different calls $N(\Gamma;\sigma)$, $N_A(\Gamma;\sigma)$, $N_B(\Gamma;\sigma)$ and $H^{x:[\tau]}(\Gamma;\sigma)$ are introduced to account for the search in different subgrammars ($\mathtt{cno}$, $\mathtt{cna}$, $\mathtt{cnb}$ and $\mathtt{cne}$, respectively). The grammar $\mathtt{B}$ then *guides* the search by hardcoding its production rules in the algorithm rules. A more general algorithm is then naturally suggested by this first one: it is constructed by processing separately the *typing* requirements from the *grammatical* ones. As a consequence, it allows one to target different solution sets using a unique algorithm. Indeed, it turns out that the rules of $\mathbf{Inh}_{\mathcal{U}}$ cannot only be guided by the grammar $\mathtt{B}$, but also by a more general *class* of grammars, called $\mathtt{NH}$-*grammars*, which can be used as a *parameter* for the algorithm. Their set of nonterminal symbols is bi-partitioned in two disjoint subsets, production rules follow a peculiar kind of patterns, and an upward closure condition must be satisfied.

Definition 5.5. An $\mathtt{NH}$-**grammar** is an $\{\bot\}$-grammar (see Def. 3.8) $G = (\Sigma_{\{\bot\}}, S_N \cup S_H, R, s)$ such that $S_N \cap S_H = \emptyset$, the start symbol $s \in S_N$ and the production rules in R follow the patterns below:

$$
\begin{array}{llllll}
\mathcal{G} \rightsquigarrow \mathcal{G}' & \mathcal{G} \rightsquigarrow \mathtt{Bng}(\bot) & \mathcal{G} \rightsquigarrow \mathtt{Bng}(\mathcal{G}') & \mathcal{G} \rightsquigarrow \mathtt{Lam}(\mathcal{G}') & \mathcal{G} \rightsquigarrow \mathtt{Sub}(\mathcal{G}_1,\mathcal{G}_2) & \mathcal{G} \rightsquigarrow \mathcal{G}' \\
\mathcal{G} \rightsquigarrow \mathcal{G}' & \mathcal{G} \rightsquigarrow \mathtt{Var} & \mathcal{G} \rightsquigarrow \mathtt{Der}(\mathcal{G}') & \mathcal{G} \rightsquigarrow \mathtt{App}(\mathcal{G}_1,\mathcal{G}_2) & \mathcal{G} \rightsquigarrow \mathtt{Sub}(\mathcal{G}_1,\mathcal{G}_2) &
\end{array}
$$

where $\mathcal{G}, \mathcal{G}', \mathcal{G}_1, \mathcal{G}_2$ are meta-variables for arbitrary nonterminal symbols, written in green or purple depending on whether they must be instantiated by nonterminal symbols which belong to S_N or S_H, respectively. Moreover, two additional conditions are required:

(1) For all $n \in S_N \cup S_H$, $n \rightsquigarrow\!\!\!\!\!\rightarrow n$ does not hold;
(2) For all rule $n \rightsquigarrow \mathtt{Bng}(n')$ in R with $n' \in S_N$, for all $\{a_i\}_{i\in I} \subseteq \Lambda_\bot$, such that $\uparrow_{i\in I} a_i$, if $a_i \sqsubseteq n'$ for all $i \in I$ then $!\bigvee_{i\in I} a_i \sqsubseteq n$.

$$\frac{g \rightsquigarrow \mathsf{Var} \;\mid}{x \Vdash_g H^{x:[\sigma]}(\emptyset;\sigma)}\;(\text{VAR}) \qquad \frac{\begin{array}{c} g \rightsquigarrow \mathsf{Der}(g') \\ {[\sigma]} \Vvdash S(\tau,\diamond) \end{array} \;\Big|\; a \Vdash_{g'} H^{x:[\tau]}(\Gamma;[\sigma])}{\mathsf{der}(a) \Vdash_g H^{x:[\tau]}(\Gamma;\sigma)}\;(\text{DR})$$

$$\frac{\begin{array}{c} g \rightsquigarrow \mathsf{App}(g_a,g_b) \\ \Gamma = \Gamma_a + \Gamma_b \\ \mathcal{M} \Rightarrow \sigma \Vvdash S(\tau, \diamond \Rightarrow \sigma) \end{array} \;\Big|\; a \Vdash_{g_a} H^{x:[\tau]}(\Gamma_a; \mathcal{M} \Rightarrow \sigma) \quad b \Vdash_{g_b} N(\Gamma_b;\mathcal{M})}{ab \Vdash_g H^{x:[\tau]}(\Gamma;\sigma)}\;(\text{APP})$$

$$\frac{g \rightsquigarrow g' \;\mid\; a \Vdash_{g'} H^{x:[\tau]}(\Gamma;\sigma)}{a \Vdash_g H^{x:[\tau]}(\Gamma;\sigma)}\;(\text{H-H}) \qquad \frac{\begin{array}{c} g \rightsquigarrow g' \\ \Gamma = \Gamma' + x:[\tau] \\ \sigma \Vvdash S(\tau,\diamond) \end{array} \;\Big|\; a \Vdash_{g'} H^{x:[\tau]}(\Gamma';\sigma)}{a \Vdash_g N(\Gamma;\sigma)}\;(\text{N-H}) \qquad \frac{g \rightsquigarrow g' \;\mid\; a \Vdash_{g'} N(\Gamma;\sigma)}{a \Vdash_g N(\Gamma;\sigma)}\;(\text{N-N})$$

$$\frac{\begin{array}{c} g \rightsquigarrow \mathsf{Lam}(g') \\ \mathtt{fix}\ x \notin \mathsf{dom}(\Gamma) \end{array} \;\Big|\; a \Vdash_{g'} N(\Gamma, x:\mathcal{M};\sigma)}{\lambda x.a \Vdash_g N(\Gamma;\mathcal{M} \Rightarrow \sigma)}\;(\text{ABS}) \qquad \frac{\begin{array}{c} g \rightsquigarrow \mathsf{Bng}(g') \\ I \neq \emptyset \\ \Gamma = +_{i\in I}\Gamma_i \end{array} \;\Big|\; (a_i \Vdash_{g'} N(\Gamma_i;\tau_i))_{i\in I} \quad \uparrow_{i\in I} a_i}{!\bigvee_{i\in I} a_i \Vdash_g N(\Gamma;[\tau_i]_{i\in I})}\;(\text{BG}) \qquad \frac{g \rightsquigarrow \mathsf{Bng}(\bot) \;\mid}{!\bot \Vdash_g N(\emptyset;[\,])}\;(\text{BG}_\bot)$$

$$\frac{\begin{array}{c} g \rightsquigarrow \mathsf{Sub}(g_a,g_b) \\ \Gamma = \Gamma_a + \Gamma_b + z:[\rho], \qquad \mathtt{fix}\ y \notin \mathsf{dom}(\Gamma) \cup \{x\} \\ n \in \llbracket 0, \mathsf{sz}(\rho) \rrbracket,\ \mathcal{M} \Vvdash S(\rho,[\diamond_1,\ldots,\diamond_n]) \end{array} \;\Big|\; a \Vdash_{g_a} H^{x:[\tau]}(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash_{g_b} H^{z:[\rho]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash_g H^{x:[\tau]}(\Gamma;\sigma)}\;(\text{ES-H})$$

$$\frac{\begin{array}{c} g \rightsquigarrow \mathsf{Sub}(g_a,g_b) \\ \Gamma = \Gamma_a + \Gamma_b, \qquad \mathtt{fix}\ y \notin \mathsf{dom}(\Gamma) \cup \{x\} \\ n \in \llbracket 1, \mathsf{sz}(\tau) \rrbracket,\ [\rho_i]_{i\in\llbracket 1,n\rrbracket} \Vvdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \\ j \in \llbracket 1,n \rrbracket,\ \sigma \Vvdash S(\rho_j,\diamond) \end{array} \;\Big|\; a \Vdash_{g_a} H^{y:[\rho_j]}(\Gamma_a, y:[\rho_i]_{i\in\llbracket 1,n\rrbracket\setminus j};\sigma) \quad b \Vdash_{g_b} H^{x:[\tau]}(\Gamma_b;[\rho_i]_{i\in\llbracket 1,n\rrbracket})}{a[y\backslash b] \Vdash_g H^{x:[\tau]}(\Gamma;\sigma)}\;(\text{ES-CH})$$

$$\frac{\begin{array}{c} g \rightsquigarrow \mathsf{Sub}(g_a,g_b) \\ \Gamma = \Gamma_a + \Gamma_b + z:[\tau], \qquad \mathtt{fix}\ y \notin \mathsf{dom}(\Gamma) \\ n \in \llbracket 0, \mathsf{sz}(\tau) \rrbracket,\ \mathcal{M} \Vvdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \end{array} \;\Big|\; a \Vdash_{g_a} N(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash_{g_b} H^{z:[\tau]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash_g N(\Gamma;\sigma)}\;(\text{ES-N})$$

Fig. 5. Rules of the Parametric Algorithm $\mathbf{Inh}^G_{\mathcal{U}}$ for System $\mathcal{U}$

Example 5.6. Grammar B (Def. 3.9) is a special case of NH-grammar, where $S_N = \{\mathsf{cno}, \mathsf{cna}, \mathsf{cnb}\}$, $S_H = \{\mathsf{cne}\}$ and the start symbol is cno. In particular, the productions rules cno $\rightsquigarrow$ cna | cnb are of the form $G \rightsquigarrow G'$, while the production rules cna $\rightsquigarrow$ cne and cnb $\rightsquigarrow$ cne are of the form $G \rightsquigarrow G'$.

Lemma 5.7. *The grammar* B *is an* NH*-grammar.*

From now on, the notation $\mathbf{Inh}^G_{\mathcal{U}}$ will be used to emphasize the particular NH-grammar G that equips the set of rules of the new parametric algorithm in Fig. 5. In the new algorithm $\mathbf{Inh}^G_{\mathcal{U}}$, the search is now guided by the NH-grammar G. Instead of hardcoding the productions using a number of different names for the calls, the rules of the parametric algorithm are built to explicitly navigate any NH-grammar G using only two kinds of calls, N and H, and the following features:

- Sequents are now of the form $a \Vdash_g$ `Call`, where g is a nonterminal symbol of the NH-grammar G, considered as the **parameter** of `Call`. This parameter specifies which nonterminal symbol guides the search and thus in which subgrammar the search must be conducted.
- The preliminary requirements of each algorithm rule are now enriched with grammar requirements; thus *e.g.* in rule (N-H) of Fig. 5 it is also required the existence of *some* rule of the form $g \rightsquigarrow g'$ in the NH-grammar G.

Thus for example, in rule (VAR) of Fig. 5, a call $H^{x:[\sigma]}(\emptyset;\sigma)$ with parameter g only succeeds if a production rule of the form $g \rightsquigarrow$ `Var` belongs to the associated grammar G. In the case of grammar $G = \mathsf{B}$, this only happens if $g = \mathsf{cne}$. A more interesting example is rule (N-N), where a call $N(\Gamma;\sigma)$

with parameter g may occur with any kind of silent rule $g \mapsto g'$ of the grammar G (there are four of such silent rules in the particular case $G = \mathsf{B}$: $\mathsf{cno} \mapsto \mathsf{cna} \mid \mathsf{cnb}$ and $\mathsf{cnb} \mapsto \mathsf{cne}$ and $\mathsf{cna} \mapsto \mathsf{cne}$).

Given an input typing $(\Gamma; \sigma)$, a **run** of $\mathbf{Inh}^{G}_{\mathcal{U}}$ is a tree of recursive parametrized calls starting from the root $N(\Gamma; \sigma)$ with the start nonterminal symbol of the associated grammar G. Notice that now three different rules (N-N), (H-H) and (N-H) are associated with silent production rules $g \mapsto g'$ in the grammar G. They are called **silent rules**, while the others are called **non-silent rules**.

Example 5.8. Let us see now some examples of the IP in $\lambda!$ using the parametric algorithm $\mathbf{Inh}^{\mathsf{B}}_{\mathcal{U}}$, that is, $\mathbf{Inh}^{G}_{\mathcal{U}}$ where G is instantiated by the particular NH-grammar B. We show the answers given by our implementation [Arrial 2023] in the mode `verbose 0`. Different levels of verbose mode are available. In particular, this allows us to visualize the different type derivations that are constructed by the algorithm when searching for the basis. The first example has already been discussed in Example 5.1 using the non-parametric algorithm $\mathbf{Inh}_{\mathcal{U}}$.

```
--- Example 1 ---
Inh N(x:[[[α]]]; α)
Sol>  z[z:=y[y:=x]],  y[y:=der(x)],  z[z:=y][y:=x],
der(y)[y:=x],  der(y[y:=x]),  der(der(x))

--- Example 2 ---
Inh N(∅; [[α]->α]->[α]->α)
Sol>  λx.x, λx.λy.x!y

--- Example 3 ---
Inh N(∅; [[[α]->[α]]->[[α]->[α]]])
Sol>  !λx.!x, !λx.!λy.x!y, !λy.!λz.(!x)[x:=y!z],
!λy.!λz.!x[x:=y!z], !λx.!λy.!der(x!y)

--- Example 4 ---
Inh N(∅; ([]->[])->[])
Sol>  ∅

--- Example 5 ---
Inh N(x:[[]->α]; α)
Sol>  x!⊥

--- Example 6 ---
Inh N(∅; [[α]->[α]])
Sol>  !λx.!x
```

We briefly revisit the run from Example 5.1 which finds the solution $\mathsf{der}(\mathsf{der}(x))$ for the typing $(x : [[[\alpha]]]\,; \alpha)$. In the *parametric* algorithm $\mathbf{Inh}^{\mathsf{B}}_{\mathcal{U}}$, the run starts from the call $N(x : [[[\alpha]]]\,; \alpha)$ with parameter cno, and rule (N-N) is applied. Since $\mathsf{cno} \mapsto \mathsf{cna}$ is a production rule of the NH-grammar B, then the next recursive call is $N(x : [[[\alpha]]]\,; \alpha)$ with parameter cna. Note that taking cnb as parameter also works. Afterwards, rule (N-H) is applied, and $\mathsf{cna} \mapsto \mathsf{cne}$ fulfills the grammar requirement. Rule (DR) follows twice, where the grammar requirement is instantiated with $\mathsf{cne} \mapsto \mathtt{Der}(\mathsf{cne})$. Finally, rule (VAR) is applied since $\mathsf{cne} \mapsto \mathtt{Var}$ is a production rule of B.

A comparison between Examples 5.1 and 5.8 shows—with an example—that algorithm $\mathbf{Inh}^{\mathsf{B}}_{\mathcal{U}}$ behaves like $\mathbf{Inh}_{\mathcal{U}}$, and in particular they return the same output, despite the former uses a different approach that separates the typing requirements from the grammatical ones. This holds in general.

PROPOSITION 5.9. *For every typing* $(\Gamma; \sigma)$, $\mathbf{Inh}_{\mathcal{U}}(\Gamma; \sigma) = \mathbf{Inh}^{\mathsf{B}}_{\mathcal{U}}(\Gamma; \sigma)$.

5.5 Properties of the Parametric Algorithm $\mathbf{Inh}^{G}_{\mathcal{U}}$

All our work introduced in Sects. 4 and 5 generalizes to arbitrary NH-grammars. In particular, statements referring to cne are generalized to subsets produced by nonterminal symbols in S_H.

Termination. The $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm is non-deterministic, and generating a finite basis for a given input means generating all possible runs. So, we focus on the proof of termination, which relies on two properties: every run of the algorithm terminates (*finite depth*, the hardest and subtlest part of the termination proof), and there is a finite number of runs for every possible input (*finite degree*). Termination of $\mathbf{Inh}_{\mathcal{U}}$ (Thm. 5.2) turns out to be a corollary of termination of $\mathbf{Inh}^{G}_{\mathcal{U}}$ (Cor. 5.14).

Finite Depth. We prove that every run of the algorithm $\mathbf{Inh}^{G}_{\mathcal{U}}$ terminates by building a decreasing measure along recursive calls, which is defined in terms of the input of these calls. Some rules of $\mathbf{Inh}^G_{\mathcal{U}}$ clearly invoke recursive calls with smaller inputs, for example (BG) and (ABS). Other rules invoke recursive calls with subtypes in the input provided by the SPK algorithm, for example, (DR), (APP), (ES-H), (ES-CH) and (ES-N); these subtypes are extracted from the input of the main call, so that the inputs of the recursive calls turn out to be smaller than the one of the main call. However, building such a decreasing measure along the recursive calls is not straightforward. First, we extend the **constructor size** for types defined before Lem. 4.6 to environments and calls:

$$\mathsf{sz}(\Gamma) := \sum_{x \in \mathsf{dom}(\Gamma)} \mathsf{sz}(\Gamma(x)) \quad \mathsf{sz}(H^{x:[\tau]}(\Gamma;\sigma)) := \mathsf{sz}(\Gamma) + \mathsf{sz}([\tau]) - \mathsf{sz}(\sigma) \quad \mathsf{sz}(N(\Gamma;\sigma)) := \mathsf{sz}(\Gamma) + \mathsf{sz}(\sigma).$$

The constructor size is clearly decreasing for the recursive calls of rules (VAR), (BG), ($\mathsf{BG}_\perp$), (DR), (APP) and (ABS). However, this measure is not still decreasing for rules (ES-H), (ES-CH) and (ES-N). We then introduce another measure which decreases in these cases. The **depth** $\mathsf{dpt}(\tau)$ of a type τ is defined as (with $n \geq 0$):

$$\mathsf{dpt}(\alpha) := 0 \qquad \mathsf{dpt}(\mathcal{M} \Rightarrow \sigma) := \mathsf{dpt}(\mathcal{M}) + \mathsf{dpt}(\sigma) \qquad \mathsf{dpt}([\sigma_i]_{i \in [\![1,n]\!]}) := n + \sum_{i \in [\![1,n]\!]} \mathsf{dpt}(\sigma_i).$$

We extend the notion of depth to environments and calls as follows:

$$\mathsf{dpt}(\Gamma) := \sum_{x \in \mathsf{dom}(\Gamma)} \mathsf{dpt}(\Gamma(x)) \qquad \mathsf{dpt}(H^{x:[\tau]}(\Gamma;\sigma)) := \mathsf{dpt}(\Gamma) + \mathsf{dpt}([\tau]) \qquad \mathsf{dpt}(N(\Gamma;\sigma)) := \mathsf{dpt}(\Gamma).$$

For example, $\mathsf{dpt}(x : [\alpha, [\beta]]) = 3$ and $\mathsf{dpt}(x : [\alpha], y : [\beta]) = 2$. Finally, the **composite measure** $\mathsf{m}(\mathtt{Call})$ of a call (either a H-call or a N-call) is given by:

$$\mathsf{m}(\mathtt{Call}) := (\mathsf{sz}(\mathtt{Call}), \mathsf{dpt}(\mathtt{Call}))$$

We can use the lexicographic order $>_{\mathtt{lex}}$ to compare the measures of two *related* calls. In particular, given two calls c, c' such that c calls c', then $\mathsf{m}(c) \geq_{\mathtt{lex}} \mathsf{m}(c')$. Formally,

Lemma 5.10. *The measure is strictly decreasing on non-silent rules and decreasing on silent rules.*

Silent rules are used to navigate the grammar without tampering the typing, thus leaving the measure possibly constant. Condition 1 in the definition of NH-grammar (Def. 5.5) is therefore required to prove that silent rules cannot be used infinitely often. This is crucial to prove termination.

Lemma 5.11. *The number of consecutive non-silent production rules in any* NH*-grammar is bounded.*

Combining these two lemmas gives the first termination argument.

Lemma 5.12 (Run termination). *Every run of the* $\mathbf{Inh}^G_{\mathcal{U}}$ *algorithm terminates.*

Finite Degree. We now focus on bounding the number of non-deterministic possible calls for a given input, so that the algorithm results to be finitely branching. Indeed, for every rule of $\mathbf{Inh}^G_{\mathcal{U}}$, several non-deterministic recursive calls are possible. The bound on non-determinism is due to three reasons: (1) a finite number of production rules; (2) a finite number of splittings for a given environment; (3) a finite number of subtypes generated by the SPK algorithm (Lem. 4.6).

Lemma 5.13 (Finite branching). *The set of possible immediate recursive calls generated by* $\mathbf{Inh}^G_{\mathcal{U}}$ *for any input and any parameter is finite.*

Using finite depth (Lem. 5.12), finite degree (Lem. 5.13) and König's Lemma, we conclude:

Corollary 5.14 (Termination of $\mathbf{Inh}^G_{\mathcal{U}}$). *The* $\mathbf{Inh}^G_{\mathcal{U}}$ *algorithm terminates.*

Soundness and completeness of $\mathbf{Inh}_{\mathcal{U}}$ (Thm. 5.3) can be generalized to $\mathbf{Inh}^G_{\mathcal{U}}$ for *any* NH-grammar G. The idea is that nonterminal symbols in S_N (resp. S_H) are used in $\mathbf{Inh}^G_{\mathcal{U}}$ to drive N-calls (resp. H-calls). Condition 2 in the definition of NH-grammar (Def. 5.5) is crucial to prove completeness.

Theorem 5.15 (The parametric algorithm is sound and complete). *Let G be an* NH*-grammar. For any typing* $(\Gamma;\sigma)$, $\mathbf{Inh}^G_{\mathcal{U}}(\Gamma;\sigma) = \mathtt{Basis}(\Gamma;\sigma) \cap \mathcal{L}(G)$.

To summarize, a major consequence of the parametrization of the algorithm $\mathbf{Inh}_{\mathcal{U}}$ by any NH-grammar G is that we can now use the *same* algorithm $\mathbf{Inh}^G_{\mathcal{U}}$ for all languages encodable into $\lambda!$. In other words, using the (same) algorithm for different languages is made possible by the grammar parameter indicating to the algorithm which kind of syntax should be followed to construct an inhabitant. We think that this feature is an important contribution of our approach, which gives a *unified* treatment of inhabitation for different models of computation.

6 CBN/CBV INHABITATION

This section discusses a direct application of our work, consisting in restricting the algorithm $\mathbf{Inh}^G_{\mathcal{U}}$ to solve the inhabitation problem for Call-by-Value (CBV) and Call-by-Name (CBN). We first give a general presentation of the operational semantics of both CBN and CBV in a simple framework with ES. Then, we define new grammars BN and BV to solve the inhabitation problems for the CBN and CBV cases by instantiating the generalized $\mathbf{Inh}^G_{\mathcal{U}}$ algorithm (Sect. 5.2). Indeed, both grammars BN and BV are instances of the general class of NH-grammars (Sect. 5.4). This methodology results in particular in two different inhabitation algorithms, one for CBV and another for CBN, both inheriting the properties of the general one for $\lambda!$. While the resulting CBV inhabitation algorithm is an original contribution of this paper (Sect. 6.3), our resulting CBN inhabitation algorithm is in some sense similar to the original CBN algorithm in the literature [Bucciarelli et al. 2018].

CBN *and* CBV *calculi.* Both CBN and CBV settings are specified using λ-calculi with ES, as in [Accattoli and Paolini 2012]. For both calculi, the set Λ_λ of **terms** is inductively defined as follows (note that der and ! are absent):

$$\textbf{(Terms)}\quad t,u ::= v \mid tu \mid t[x\backslash u] \qquad\qquad \textbf{(Values)}\quad v ::= x \mid \lambda x.t$$

Full contexts F are terms with exactly one occurrence of the symbol $\diamond$. Reductions for CBN and CBV are driven by the following notion of contexts, which allow actions at a *distance*:

$$\begin{array}{rrcl} \textbf{(List Contexts)} & \mathsf{L} & ::= & \diamond \mid \mathsf{L}[x\backslash t] \\ \textbf{(CBN Contexts)} & \mathsf{N} & ::= & \diamond \mid \mathsf{N}u \mid \lambda x.\mathsf{N} \mid \mathsf{N}[x\backslash u] \\ \textbf{(CBV Contexts)} & \mathsf{V} & ::= & \diamond \mid \mathsf{V}u \mid t\mathsf{V} \mid \mathsf{V}[x\backslash u] \mid t[x\backslash \mathsf{V}] \end{array}$$

The **CBN reduction relation** $\rightarrow_{\mathsf{N}}$ is defined as the closure of the rules dB and NS below under contexts N, while the **CBV reduction relation** $\rightarrow_{\mathsf{V}}$ is defined as the closure of the rules dB and VS below under contexts V.

$$\mathsf{L}\langle \lambda x.t\rangle\, u \;\mapsto_{\mathsf{dB}}\; \mathsf{L}\langle t[x\backslash u]\rangle \qquad t[x\backslash u] \;\mapsto_{\mathsf{NS}}\; t\{x\backslash u\} \qquad t[x\backslash \mathsf{L}\langle v\rangle] \;\mapsto_{\mathsf{VS}}\; \mathsf{L}\langle t\{x\backslash v\}\rangle$$

Rule dB (resp. VS) is assumed to be capture free, thus no free variable of u (resp. t) is captured by context L. CBN and CBV differ in that CBN can always fire an ES, while CBV can only if the ES argument is a value, possibly wrapped by a finite list of ES. So, *e.g.* $(\lambda x.yxx)(II) \rightarrow_{\mathsf{N}} (yxx)[x\backslash II] \rightarrow_{\mathsf{N}} y(II)(II) \twoheadrightarrow_{\mathsf{N}} yI(II) \twoheadrightarrow_{\mathsf{N}} yII$ and $(\lambda x.yxx)(II) \rightarrow_{\mathsf{V}} (yxx)[x\backslash II] \twoheadrightarrow_{\mathsf{V}} (yxx)[x\backslash I] \twoheadrightarrow_{\mathsf{V}} yII$. Notice how reduction $\rightarrow_{\mathsf{V}}$ unblocks redexes, *e.g.* given $\delta := \lambda z.zz$, the term $t := (\lambda y.\delta)(xx)\delta$, which is a normal form in Plotkin's CBV [Plotkin 1975], is now non-terminating $t \rightarrow_{\mathsf{V}} \delta[y\backslash xx]\delta \rightarrow_{\mathsf{V}} (zz)[z\backslash\delta][y\backslash xx] \rightarrow_{\mathsf{V}} (\delta\delta)[y\backslash xx] \twoheadrightarrow_{\mathsf{V}} (\delta\delta)[y\backslash xx]$, as one would expect, since it is also denotationally non-terminating [Carraro and Guerrieri 2014; Paolini and Ronchi Della Rocca 1999].

In order to define appropriate notions of typed reduction, we also need to introduce **full CBN** reduction, written $\rightarrow_{\mathsf{FN}}$, (resp. **full CBV** reduction, written $\rightarrow_{\mathsf{FV}}$), given as the closure of the rewriting rules dB and NS (resp. dB and VS) under *full* contexts .

$$\frac{}{x : [\sigma] \vdash x : \sigma}\text{ax} \qquad \frac{\Gamma, x : [\tau_i]_{i\in I} \vdash t : \sigma \quad (\Gamma_i \vdash u : \tau_i)_{i\in I} \quad I \text{ finite}}{\Gamma +_{i\in I} \Gamma_i \vdash t[x\backslash u] : \sigma}\text{es}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \mathbin{\backslash\!\!\backslash} x \vdash \lambda x.t : \Gamma(x) \Rightarrow \sigma}\text{abs} \qquad \frac{\Gamma \vdash t : [\tau_i]_{i\in I} \Rightarrow \sigma \quad (\Gamma_i \vdash u : \tau_i)_{i\in I} \quad I \text{ finite}}{\Gamma +_{i\in I} \Gamma_i \vdash tu : \sigma}\text{app}$$

Fig. 6. Type System $\mathcal{N}$ for the CBN λ-calculus.

$$\frac{}{x : \mathcal{M} \vdash x : \mathcal{M}}\text{ax} \qquad \frac{\Gamma \vdash t : \sigma \quad \Gamma' \vdash u : \Gamma(x)}{(\Gamma \mathbin{\backslash\!\!\backslash} x) + \Gamma' \vdash t[x\backslash u] : \sigma}\text{es}$$

$$\frac{(\Gamma_i \vdash t : \sigma_i)_{i\in I} \quad I \text{ finite}}{+_{i\in I}\Gamma_i \mathbin{\backslash\!\!\backslash} x \vdash \lambda x.t : [\Gamma_i(x) \Rightarrow \sigma_i]_{i\in I}}\text{abs} \qquad \frac{\Gamma \vdash t : [\mathcal{M} \Rightarrow \sigma] \quad \Gamma' \vdash u : \mathcal{M}}{\Gamma + \Gamma' \vdash tu : \sigma}\text{app}$$

Fig. 7. Type System $\mathcal{V}$ for the CBV λ-calculus.

CBN *and* CBV *type systems.* We now present the quantitative type systems $\mathcal{N}$ and $\mathcal{V}$ for CBN and CBV, respectively, already studied in [Bucciarelli et al. 2020]. **Types** and **judgments** are the same as for system $\mathcal{U}$. The typing rules of systems $\mathcal{N}$ and $\mathcal{V}$ are in Figs. 6 and 7, respectively. A derivation Π in system $\mathcal{N}$ with conclusion $\Gamma \vdash t : \sigma$ is noted $\Pi \triangleright_{\mathcal{N}} \Gamma \vdash t : \sigma$, and similarly with $\triangleright_{\mathcal{V}}$ for system $\mathcal{V}$.

Notice how typed terms may contain untyped subterms in both systems. Indeed, in system $\mathcal{N}$, untyped subterms are introduced by rules (es) and (app) with an empty set I; in system $\mathcal{V}$, untyped subterms are introduced by rule (ax) with an empty multiset, or rule (abs) with an empty set I.

The salient property of type systems $\mathcal{N}$ and $\mathcal{V}$ is the characterization of normalization in CBN and CBV, respectively.

Theorem 6.1 (Characterization of normalization, [Bucciarelli et al. 2020]). *Let t in Λ_λ.*

(1) CBN *normalization: t is $\mathcal{N}$-typable if and only if it is $\to_{\mathsf{N}}$-normalizing.*
(2) CBV *normalization: t is $\mathcal{V}$-typable if and only if it is $\to_{\mathsf{V}}$-normalizing.*

Both CBN and CBV can be embedded into the $\lambda!$-calculus by preserving the typing. This makes it possible to decide the CBN/CBV inhabitation problem using the original $\mathbf{Inh}_{\mathcal{U}}$ algorithm.

6.1 Call-by-Name Inhabitation

We now consider the inhabitation problem for CBN equipped with the typing system $\mathcal{N}$ (Fig. 6). First, we specify which are the new tools and notions needed to face the IP for CBN, which are in fact adapted from those in Sect. 3. Since we have already discussed those notions at length in the previous sections, the new inherited definitions for CBN are now just briefly mentioned, or presented in a sober way. Then, instead of building a new algorithm from scratch, we show that there is an embedding of CBN into $\lambda!$ (Def. 6.3) which does not only preserve typing, but also the crucial notion of basis, thus allowing to restrict the search of the $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm on a new grammar called BN, in order to solve the CBN inhabitation (Lem. 6.5). Finally, we compare the resulting algorithm obtained by this method, with the original one in the literature [Bucciarelli et al. 2018].

Soundness and Completeness of the Basis. Appropriate notions of $\mathcal{N}$**-typed locations**, **normal** $\mathcal{N}$**-derivations**, and $\mathcal{N}$**-typed reductions** are introduced as expected for the type system $\mathcal{N}$ and the CBN reduction relation, following these same concepts for the $\lambda!$-calculus (Sect. 3), but taking into account how untyped subterms may now occur in $\mathcal{N}$-typed terms. Indeed, untyped subterms can be now introduced by using an empty set I in the typing rules (es) and (app). As in $\lambda!$, one constant $\perp$ is added to the set Λ_λ to canonically represent untyped subterms of typed terms. The resulting set $\Lambda_\perp^{\mathcal{N}}$ of **⊥-terms** of CBN is given by the inductive definition below:

$$a, b \ ::= \ \perp \mid x \mid \lambda x.a \mid ab \mid a[x\backslash b]$$

A **preorder** $\preceq_{\mathsf{N}}$ on $\Lambda_\perp^{\mathcal{N}}$ is given by the full contextual closure of $\perp \preceq_{\mathsf{N}} a$ for any ⊥-terms a. Then **$\mathcal{N}$-approximants** of normal $\mathcal{N}$-derivations Π, noted $\mathcal{A}_{\mathcal{N}}(\Pi)$, as well as **canonical $\mathcal{N}$-derivations**, and **canonical $\mathcal{N}$-solutions**, are defined following the same concepts as in Sect. 3. Canonical $\mathcal{N}$-solutions can be produced by the **C-grammar** N below, where $C = \{\perp\}$ and c is the start symbol:

$$(\mathsf{N}) \qquad \mathsf{a} \leadsto \mathsf{Var} \mid \mathsf{App}(\mathsf{a}, \mathsf{b}) \qquad \mathsf{b} \leadsto \mathsf{c} \mid \perp \qquad \mathsf{c} \leadsto \mathsf{Lam}(\mathsf{c}) \mid \mathsf{a}.$$

Grammar N will play a major role to develop our CBN inhabitation algorithm. Indeed, in a new grammar, so that the same $\mathbf{Inh}_{\mathcal{U}}^{G}$ algorithm presented in Sect. 5, and not a new one, will be used on this new grammar to generate the CBN basis.

The key notions of **basis** and **span** for the CBN case are also defined as expected:

$$\mathtt{Basis}_{\mathcal{N}}(\Gamma;\sigma) := \{b \in \Lambda_\perp^{\mathcal{N}} \mid b \text{ canonical } \mathcal{N}\text{-solution for } (\Gamma;\sigma)\}$$
$$\mathtt{Span}_{\mathcal{N}}(S) := \{t \in \Lambda_\lambda \mid \exists\, a \in S,\ \exists\, u \in \Lambda_\lambda,\ a \preceq_{\mathsf{N}} u \text{ and } t \twoheadrightarrow_{\mathsf{FN}} u\}$$

The solution set of the CBN IP for the typing $(\Gamma;\sigma)$ is $\mathtt{Sol}_{\mathcal{N}}(\Gamma;\sigma) := \{t \in \Lambda_\lambda \mid \exists\, \Pi \triangleright_{\mathcal{N}} \Gamma \vdash t : \sigma\}$. It is generated by the basis as expected:

Lemma 6.2 (Sound & Complete Basis). *For any typing* $(\Gamma;\sigma)$, $\mathtt{Span}_{\mathcal{N}}(\mathtt{Basis}_{\mathcal{N}}(\Gamma;\sigma)) = \mathtt{Sol}_{\mathcal{N}}(\Gamma;\sigma)$.

Embedding in $\lambda!$. Now that the basis of solutions for CBN inhabitation have been presented, we focus on relating CBV to $\lambda!$. We use the embedding below introduced by [Bucciarelli et al. 2020]:

Definition 6.3. The **CBN embedding** $(\_)^{\mathsf{cbn}} : \Lambda_\lambda \to \Lambda$ is defined as follows:

$$x^{\mathsf{cbn}} := x \qquad (\lambda x.t)^{\mathsf{cbn}} := \lambda x.t^{\mathsf{cbn}} \qquad (tu)^{\mathsf{cbn}} := t^{\mathsf{cbn}}!u^{\mathsf{cbn}} \qquad (t[x\backslash u])^{\mathsf{cbn}} := t^{\mathsf{cbn}}[x\backslash !u^{\mathsf{cbn}}].$$

Example 6.4. Let $I := \lambda x.x$ and $\Delta := \lambda x.x!x$ and $\delta := \lambda z.zz$. We have $I^{\mathsf{cbn}} = I$ and $\delta^{\mathsf{cbn}} = \Delta$ and $(\delta\delta)^{\mathsf{cbn}} = \Delta\,!\Delta$. Moreover, $((xy)z)^{\mathsf{cbn}} = (x!y)!z$.

An intuitive way to understand this encoding is given by the following fact: while any argument (right-hand side of application or substitution) can be erased/duplicated in CBN, only bang terms can be erased/duplicated in the $\lambda!$-calculus, so that arguments must be translated to bang terms.

The embedding $\_^{\mathsf{cbn}}$ translates $\to_{\mathsf{FN}}$-reduction steps (in CBN) into $\to_{\mathsf{F}}$-reduction steps (in $\lambda!$) [Bucciarelli et al. 2018]. Beyond this untyped result, we need to show that the embedding also preserves typing and canonicity, which are used to decide the IP. For that, we first extend the previous embedding on terms to ⊥-terms by setting in particular $\perp^{\mathsf{cbn}} := \perp$. Note that the embedding is *injective*, and that its **preimage**, denoted $(\_)^{-\mathsf{cbn}}$, can be described by a simple erasure of bangs (!).

From now on, we annotate with an index $\mathcal{U}$ all the previously defined functions/predicates related to the $\lambda!$-calculus and system $\mathcal{U}$, to avoid confusion with the corresponding notions for CBN. For example, we now denote by $\mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma)$ the set $\mathtt{Basis}(\Gamma;\sigma)$ (Sect. 3.2). Next lemma states that typing, normal derivations and approximants of the two systems are related by the CBN embedding.

Lemma 6.5 (Bridge).

(1) (CBN $\to \lambda!$) *Let* $\Pi \triangleright_{\mathcal{N}} \Gamma \vdash a : \sigma$. *Then there exists* $\Pi' \triangleright_{\mathcal{U}} \Gamma \vdash a^{\mathsf{cbn}} : \sigma$. *Moreover, if* Π *is normal then* Π' *is also normal and* $\mathcal{A}_{\mathcal{U}}(\Pi') = \mathcal{A}_{\mathcal{N}}(\Pi)^{\mathsf{cbn}}$ *when defined.*

(2) ($\lambda! \to$ CBN) *Let* $\Pi \triangleright_{\mathcal{U}} \Gamma \vdash a : \sigma$. *Then for the unique* $b \in a^{-\mathsf{cbn}}$, *there exists* $\Pi' \triangleright_{\mathcal{N}} \Gamma \vdash b : \sigma$. *Moreover, if* Π *is normal then* Π' *is also normal and* $\mathcal{A}_{\mathcal{U}}(\Pi) = \mathcal{A}_{\mathcal{N}}(\Pi')^{\mathsf{cbn}}$ *when defined.*

Thus, the translation of an element of $\mathtt{Basis}_{\mathcal{N}}(\Gamma;\sigma)$ is an element of $\mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma)$. Conversely, any ⊥-term that translates to an element of $\mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma)$ is an element of $\mathtt{Basis}_{\mathcal{N}}(\Gamma;\sigma)$.

CBN *Inhabitation.* Having all these results in mind, one could now build a direct new algorithm solving the inhabitation problem for CBN. However, since the CBN basis is translated to the $\lambda!$ basis, we can easily exploit the original $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm to also decide the IP for CBN, where the driving grammar G is no longer B, but another NH-grammar. For that, let us focus on the image of the CBN basis by considering the C-**grammar** BN below, where $C = \{\bot\}$ and nno is the start symbol:

$$(\mathsf{BN}) \qquad \mathsf{nne} \rightsquigarrow \mathsf{Var} \mid \mathsf{App}(\mathsf{nne}, \mathsf{nna}) \qquad \mathsf{nna} \rightsquigarrow \mathsf{Bng}(\mathsf{nno}) \mid \mathsf{Bng}(\bot) \qquad \mathsf{nno} \rightsquigarrow \mathsf{Lam}(\mathsf{nno}) \mid \mathsf{nne}$$

Grammar BN can also be seen as the image grammar N via the CBV embedding, as well as the intersection of the canonicals $\bot$-terms of $\lambda!$ with the image of the CBN embedding. Formally,

Lemma 6.6. *Grammar* BN *is an* NH*-grammar. Moreover, for every $a \in \Lambda_\bot$*

$$a \sqsubseteq \mathsf{BN} \quad \Leftrightarrow \quad \exists\, b \sqsubseteq \mathsf{N},\ a = b^{\mathsf{cbn}} \quad \Leftrightarrow \quad \exists\, b \in \Lambda_\bot^{\mathcal{N}},\ a = b^{\mathsf{cbn}} \text{ and } a \sqsubseteq \mathsf{B}.$$

Using Cor. 5.14 and Thm. 5.15, we deduce that $\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}$ terminates and computes the image of the CBN basis through the embedding. A simple erasure of bangs on the results of the algorithm $\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}(\Gamma;\sigma)$ allows us to decide the IP for CBN.

Theorem 6.7. *For any typing* $(\Gamma;\sigma)$, $\mathsf{Span}_{\mathcal{N}}(\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbn}}) = \mathtt{Sol}_{\mathcal{N}}(\Gamma;\sigma)$.

Proof. By soundness and completeness of the parametric algorithm (Thm. 5.15) applied to the NH-grammar BN (Lem. 6.6), one has $\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}(\Gamma;\sigma) = \mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma) \cap \mathcal{L}(\mathsf{N})^{\mathsf{cbn}}$. Using the bridge (Lem. 6.5), one obtains $\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbn}} = \mathtt{Basis}_{\mathcal{N}}(\Gamma;\sigma)$ and by soundness and completeness of the basis (Lem. 6.2), one concludes that $\mathsf{Span}_{\mathcal{N}}(\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbn}}) = \mathtt{Sol}_{\mathcal{N}}(\Gamma;\sigma)$. □

Example 6.8. Let us see some examples of the IP in CBN, that is, some examples of runs of $\mathbf{Inh}^{G}_{\mathcal{U}}$ using the grammar $G = \mathsf{BN}$, for the same input typings as in Example 5.1. We show the answers given by our implementation [Arrial 2023] in the mode `verbose 0`.

```
--- Example 1 ---
Inh N(x:[[[α]]]; α)
Sol>  ∅

--- Example 2 ---
Inh N(∅; [[α]->α]->[α]->α)
Sol>  λx.x, λx.λy.xy

--- Example 3 ---
Inh N(∅; [[[α]->[α]]->[[α]->[α]]])
Sol>  ∅

--- Example 4 ---
Inh N(∅; ([]->[])->[])
Sol>  ∅

--- Example 5 ---
Inh N(x:[[]->α]; α)
Sol>  x⊥

--- Example 6 ---
Inh N(∅; [[α]->[α]])
Sol>  ∅
```

6.2 CBN Inhabitation Direct Method

By instantiating the $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm with the grammar BN, one can see that our (*indirect*) algorithm $\mathbf{Inh}^{\mathsf{BN}}_{\mathcal{U}}$, working on the type system $\mathcal{U}$ and being driven by BN, *encodes* a (*direct*) algorithm solving the IP for CBN, working on system $\mathcal{N}$ and grammar N. Indeed, grammar BN is the image of grammar N by the CBN encoding. The *direct* algorithm for CBN, called $\mathbf{Inh}_{\mathcal{N}}$, can be obtained by an operation of "preimage extraction", it decides the IP for CBN but without passing through $\lambda!$: its answers are *directly* canonical $\mathcal{N}$-solutions, which (canonically) represent type derivations in system $\mathcal{N}$.

We obtain the set of rules presented in Fig. 8 which is perfectly isomorphic to the original one [Bucciarelli et al. 2014]. It is worth highlighting that sz(_) alone is sufficient to show termination of these two (direct) isomorphic algorithms, where the answers do not contain neither derelictions, nor ES. This a clue that the IP for CBN is considerably easier than the one for $\lambda!$.

$$\frac{\mid}{x \Vdash H^{x:[\sigma]}(\emptyset;\sigma)}\ \text{VAR} \qquad \frac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma_2 \\ [\mathcal{M} \Rightarrow \sigma] \Vdash S(\tau, [\diamond \Rightarrow \sigma])\end{array} \;\Big|\; a_1 \Vdash H^{x:[\tau]}(\Gamma_1; [\mathcal{M} \Rightarrow \sigma]) \quad a_2 \Vdash N_A(\Gamma_2; \mathcal{M})}{a_1 a_2 \Vdash H^{x:[\tau]}(\Gamma;\sigma)}\ \text{APP}$$

$$\frac{\begin{array}{c} I \neq \emptyset \\ \Gamma = +_{i \in I}\Gamma_i\end{array} \;\Big|\; (\ a_i \Vdash N(\Gamma_i;\sigma_i)\ )_{i \in I} \quad \uparrow_{i \in I} a_i}{\bigvee_{i \in I} a_i \Vdash N_A(\Gamma; [\sigma_i]_{i \in I})}\ \text{SUP} \qquad \frac{\mid}{\bot \Vdash N_A(\emptyset; [\,])}\ \text{BOT}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma' + x : [\tau] \\ \sigma \Vdash S(\tau, \diamond)\end{array} \;\Big|\; a \Vdash H^{x:[\tau]}(\Gamma';\sigma)}{a \Vdash N(\Gamma;\sigma)}\ \text{N-H} \qquad \frac{\texttt{fix}\ x \in \mathrm{dom}(\Gamma) \;\mid\; a' \Vdash N(\Gamma, x : \mathcal{M}; \sigma)}{\lambda x.a' \Vdash N(\Gamma; \mathcal{M} \Rightarrow \sigma)}\ \text{ABS}$$

Fig. 8. Rules of the Direct Algorithm $\mathbf{Inh}_{\mathcal{N}}$ for System $\mathcal{N}$

6.3 Call-by-Value Inhabitation

As in the CBN case, we start by briefly outlining the tools needed to face the IP for CBV, although highlighting some important differences. Again, instead of building a new inhabitation algorithm for CBV from scratch, we show that there is an embedding of CBV into $\lambda!$ (Def. 6.10) which preserves the crucial notions of typing and basis, thus allowing once again to exploit the $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm on a new grammar, this time called BV, in order to solve the CBV inhabitation (Lem. 6.12). We also provide the expected completeness and correction properties.

Soundness and Completeness of the Basis. Appropriate notions of **$\mathcal{V}$-typed locations**, **normal $\mathcal{V}$-derivations**, and **$\mathcal{V}$-typed reductions** are introduced as expected for the type system $\mathcal{V}$ and the CBV reduction relation, following these same concepts for the $\lambda!$-calculus (Sect. 3), but taking into account how untyped subterms may now occur in $\mathcal{V}$-typed terms (see p. 22, after Fig. 7).

As in $\lambda!$, constants are introduced to canonically represent untyped subterms. However, in CBV, two distinct constants are used: indeed, untyped subterms introduced by the (ax) rule are always variables, whereas those introduced by the (abs) rule are arbitrary terms. The former are represented by the constant $\bot_{\mathsf{v}}$ and the latter by the constant $\bot$. The resulting set $\Lambda^{\mathcal{V}}_{\bot,\bot_{\mathsf{v}}}$ of **$(\bot, \bot_{\mathsf{v}})$-terms** of CBV is given by the inductive definition below:

$$a, b \ ::= \ \bot \mid \bot_{\mathsf{v}} \mid x \mid \lambda x.a \mid ab \mid a[x\backslash b]$$

A **preorder** $\preceq_{\mathsf{V}}$ on $\Lambda^{\mathcal{V}}_{\bot,\bot_{\mathsf{v}}}$ is given by the $\mathcal{V}$-contextual closure of $\bot_{\mathsf{v}} \preceq_{\mathsf{V}} x$ for any variable x and $\bot \preceq_{\mathsf{V}} a$ for any $\bot_{\mathsf{v}}$-term a. Thus *e.g.* $\bot_{\mathsf{v}}(y\bot) \preceq_{\mathsf{V}} x(y(zw))$. Then **$\mathcal{V}$-approximants** of normal $\mathcal{V}$-derivations Π, noted $\mathscr{A}_{\mathcal{V}}(\Pi)$, as well as **canonical $\mathcal{V}$-derivations**, and **canonical $\mathcal{V}$-solutions**, are defined following the same concepts as in Sect. 3. Canonical $\mathcal{V}$-solutions can be generated by the following **C-grammar** V, where $C = \{\bot, \bot_{\mathsf{v}}\}$ and c is the start nonterminal symbol:

(V) $\quad \mathsf{a} \leadsto \mathsf{Var} \mid \mathsf{Sub}(\mathsf{a}, \mathsf{b}) \qquad \mathsf{c} \leadsto \mathsf{Lam}(\bot) \mid \mathsf{Lam}(\mathsf{c}) \mid \bot_{\mathsf{v}} \mid \mathsf{Var} \mid \mathsf{b} \mid \mathsf{Sub}(\mathsf{c}, \mathsf{b})$
$\quad \mathsf{b} \leadsto \mathsf{App}(\mathsf{a}, \mathsf{c}) \mid \mathsf{App}(\mathsf{b}, \mathsf{c}) \mid \mathsf{Sub}(\mathsf{b}, \mathsf{b})$

Notice that the grammar V characterizing canonical $\mathcal{V}$-solutions is significantly more complex than the grammar N of canonical $\mathcal{N}$-solutions. In particular, it makes use of the binary symbol $\mathsf{Sub}(\_,\_)$ absent in grammar N: such a symbol produces $\bot$-terms with explicit substitutions (ES) that are necessary in to denote CBV normal forms (*i.e.* $x[x\backslash yz]$), in contrast to the CBN case.

The key notions of **basis** and **span** for the CBN case are also defined as expected:

$$\mathsf{Basis}_{\mathcal{V}}(\Gamma;\sigma) := \{b \in \Lambda^{\mathcal{V}}_{\bot,\bot_{\mathsf{v}}} \mid b \text{ canonical } \mathcal{V}\text{-solution for } (\Gamma;\sigma)\}$$
$$\mathsf{Span}_{\mathcal{V}}(S) := \{t \in \Lambda_{\lambda} \mid \exists\, a \in S,\ \exists\, u \in \Lambda_{\lambda},\ a \preceq_{\mathsf{V}} u \text{ and } t \twoheadrightarrow_{\mathsf{FV}} u\}$$

The solution set of the CBV IP for the typing $(\Gamma;\sigma)$ is $\mathsf{Sol}_{\mathcal{V}}(\Gamma;\sigma) := \{t \in \Lambda_{\lambda} \mid \exists\, \Pi \triangleright_{\mathcal{V}} \Gamma \vdash t : \sigma\}$. As expected, the solution set is generated by the basis, as stated below.

Lemma 6.9 (Sound & Complete Basis). *For any typing* $(\Gamma;\sigma)$, $\mathtt{Span}_{\mathcal{V}}(\mathtt{Basis}_{\mathcal{V}}(\Gamma;\sigma)) = \mathtt{Sol}_{\mathcal{V}}(\Gamma;\sigma)$.

Embedding in $\lambda!$. Now we focus on relating CBV to $\lambda!$. This is done by means of the following embedding introduced by [Bucciarelli et al. 2020]:

Definition 6.10. The **CBV embedding** $(\_)^{\mathsf{cbv}}: \Lambda_\lambda \to \Lambda$ is defined as follows:

$$x^{\mathsf{cbv}} := !x \qquad\qquad (\lambda x.t)^{\mathsf{cbv}} := !\lambda x.t^{\mathsf{cbv}}$$

$$(tu)^{\mathsf{cbv}} := \begin{cases} \mathsf{L}\langle s\rangle\, u^{\mathsf{cbv}} & \text{if } t^{\mathsf{cbv}} = \mathsf{L}\langle !s\rangle \\ \mathsf{der}(t^{\mathsf{cbv}})u^{\mathsf{cbv}} & \text{otherwise} \end{cases} \qquad (t[x\backslash u])^{\mathsf{cbv}} := t^{\mathsf{cbv}}[x\backslash u^{\mathsf{cbv}}]$$

Example 6.11. As in Example 6.4, let $I := \lambda x.x$ and $\Delta := \lambda x.x!x$ and $\delta := \lambda z.zz$. We have $I^{\mathsf{cbv}} = !\lambda x.!x$ and $\delta^{\mathsf{cbv}} = !\Delta = !(\delta^{\mathsf{cbn}})$ and $(\delta\delta)^{\mathsf{cbv}} = \Delta\,!\Delta = (\delta\delta)^{\mathsf{cbn}}$. Moreover, $((xy)z)^{\mathsf{cbv}} = \mathsf{der}(x!y)\,!z$.

While any value can be erased/duplicated in CBV, only bang terms can be erased/duplicated in the $\lambda!$-calculus, so that values must be translated to bang terms. However, this remark alone is not sufficient to achieve a CBV embedding enjoying good properties, and in particular to translate CBV-*normal forms* into $\lambda!$-*normal forms*. The translation of applications is precisely designed in order to guarantee this property.

The embedding $\_^{\mathsf{cbv}}$ translates $\rightarrow_{\mathsf{FN}}$-reduction steps (in CBV) into $\rightarrow_{\mathsf{F}}$-reduction steps (in $\lambda!$) [Bucciarelli et al. 2018]. Beyond this untyped result, we now show that the embedding preserves typing, canonical solutions, and the basis, which are needed to decide the IP. For that, we first extend the previous embedding on terms to $(\bot, \bot_{\mathsf{v}})$-terms, by setting in particular:

$$\bot^{\mathsf{cbv}} := \bot \qquad\qquad {\bot_{\mathsf{v}}}^{\mathsf{cbv}} := !\bot \qquad\qquad (\lambda x.\bot)^{\mathsf{cbv}} := !\bot$$

Notice that this extension is no longer injective: it identifies two canonical ways of introducing an untyped subterm. However, as we shall see, the typing and canonicity are preserved on the entire preimage $(\_)^{-\mathsf{cbv}}$ of the embedding. Moreover, the preimage of this embedding does not only correspond to erasing bangs and derelictions, but also to replacing each constant $!\bot$ in the $\lambda!$ side by two possible terms $\lambda x.\bot$ and $\bot_{\mathsf{v}}$ in the CBV side. Next lemma states that typing, normal derivations and approximants are related by the CBV embedding.

Lemma 6.12 (Bridge).

(1) (CBV $\to \lambda!$) *Let* $\Pi \triangleright_{\mathcal{V}} \Gamma \vdash a : \sigma$. *Then there exists* $\Pi' \triangleright_{\mathcal{U}} \Gamma \vdash a^{\mathsf{cbv}} : \sigma$. *Moreover, if* Π *is normal then* Π *is also normal and* $\mathcal{A}_{\mathcal{U}}(\Pi') = \mathcal{A}_{\mathcal{V}}(\Pi)^{\mathsf{cbv}}$ *when defined;*

(2) ($\lambda! \to$ CBV) *Let* $\Pi \triangleright_{\mathcal{U}} \Gamma \vdash a : \sigma$. *Then for any* $b \in a^{-\mathsf{cbv}}$, *there exists* $\Pi' \triangleright_{\mathcal{V}} \Gamma \vdash b : \sigma$. *Moreover, if* Π *is normal then* Π' *is also normal and* $\mathcal{A}_{\mathcal{U}}(\Pi) = \mathcal{A}_{\mathcal{V}}(\Pi')^{\mathsf{cbv}}$ *with* $\mathcal{A}_{\mathcal{V}}(\Pi') = b$ *if* $a = \mathcal{A}_{\mathcal{U}}(\Pi)$ *when defined.*

As with CBN, the translation of an element of $\mathtt{Basis}_{\mathcal{V}}(\Gamma;\sigma)$ is also an element of $\mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma)$. Conversely, any $\bot$-term in the preimage of an element of $\mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma)$ is an element of $\mathtt{Basis}_{\mathcal{V}}(\Gamma;\sigma)$.

CBV *Inhabitation.* Again, we use the restriction of the $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm on a new NH-grammar to decide the IP for CBV. For that, let us focus on the image of the CBV basis. Consider the following C**-grammar** BV, where $C = \{\bot\}$ and vno is the start symbol:

$$\text{(BV)}\quad \begin{array}{llll} \mathsf{vne_v} \rightsquigarrow \mathsf{Var} & & \mathsf{vne_f} \rightsquigarrow \mathsf{Var} \mid \mathsf{Sub}(\mathsf{vne_f}, \mathsf{vne_a}) \\ \mathsf{vne_d} \rightsquigarrow \mathsf{Der}(\mathsf{vne_a}) & & \mathsf{vne_a} \rightsquigarrow \mathsf{App}(\mathsf{vne_f}, \mathsf{vno}) \mid \mathsf{App}(\mathsf{vne_d}, \mathsf{vno}) \mid \mathsf{Sub}(\mathsf{vne_a}, \mathsf{vne_a}) \\ \mathsf{vnb} \rightsquigarrow \mathsf{Lam}(\mathsf{vno}) \mid \mathsf{vne_v} & & \mathsf{vno} \rightsquigarrow \mathsf{Bng}(\mathsf{vnb}) \mid \mathsf{Bng}(\bot) \mid \mathsf{vne_a} \mid \mathsf{Sub}(\mathsf{vno}, \mathsf{vne_a}) \end{array}$$

Grammar BV can also be seen as the image grammar V via the CBV embedding, as well as the intersection of the canonicals $\bot$-terms of $\lambda!$ with the image of the CBV embedding. Formally, Lemma 6.13. *Grammar* BV *is an* NH-*grammar. Moreover, for every* $a \in \Lambda_\perp$

$$a \sqsubseteq \mathsf{BV} \quad \Leftrightarrow \quad \exists\, b \sqsubseteq \mathsf{V},\ a = b^{\mathsf{cbv}} \quad \Leftrightarrow \quad \exists\, b \in \Lambda^{\mathcal{V}}_{\perp,\perp_{\mathsf{v}}},\ a = b^{\mathsf{cbv}} \text{ and } a \sqsubseteq \mathsf{B}.$$

Using Cor. 5.14 and Thm. 5.15, we deduce that $\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}$ terminates and computes the image of the CBV basis through the embedding. A simple erasure on the results allows us to decide the IP for CBV.

Theorem 6.14. *For every typing* $(\Gamma;\sigma)$, $\mathsf{Span}_{\mathcal{V}}(\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbv}}) = \mathtt{Sol}_{\mathcal{V}}(\Gamma;\sigma)$.

Proof. By soundness and completeness of the parametric algorithm (Thm. 5.15) applied to the NH-grammar BV (Lem. 6.13), one has $\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}(\Gamma;\sigma) = \mathtt{Basis}_{\mathcal{U}}(\Gamma;\sigma) \cap \mathcal{L}(\mathsf{V})^{\mathsf{cbv}}$. By the bridge (Lem. 6.12), $\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbv}} = \mathtt{Basis}_{\mathcal{V}}(\Gamma;\sigma)$ and so, by soundness and completeness of the basis (Lem. 6.9), $\mathsf{Span}_{\mathcal{V}}(\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}(\Gamma;\sigma)^{\text{-cbv}}) = \mathtt{Sol}_{\mathcal{V}}(\Gamma;\sigma)$. □

Example 6.15. Let us see some examples of the IP in CBV, *i.e.*, some runs of $\mathbf{Inh}^{G}_{\mathcal{U}}$ using the grammar $G = \mathsf{BV}$, for the same input typings as in Examples 5.1 and 6.8. We show the answers given by our implementation [Arrial 2023] in the mode `verbose 0`.

```
--- Example 1 ---
Inh N(x:[[[α]]]; α)
Sol>  ∅

--- Example 2 ---
Inh N(∅; [[α]->α]->[α]->α)
Sol>  ∅

--- Example 3 ---
Inh N(∅; [[[α]->[α]]->[[α]->[α]]])
Sol>  λx.x, λx.λy.xy, λy.λz.(x)[x:=yz]

--- Example 4 ---
Inh N(∅; ([]->[])->[])
Sol>  ∅

--- Example 5 ---
Inh N(x:[[]->α]; α)
Sol>  x_, x(λ_.⊥)

--- Example 6 ---
Inh N(∅; [[α]->[α]])
Sol>  λx.x
```

6.4 CBV Inhabitation Direct Method

For the sake of completeness, we also briefly present a *direct* algorithm solving the IP problem for CBV which does not pass through $\lambda!$. For that, we follow the same ideas used to obtain the direct algorithm $\mathbf{Inh}_{\mathcal{N}}$ for CBN in Sect. 6.2. Indeed, the (*indirect*) algorithm $\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}$, working on system $\mathcal{U}$ and grammar BV, was obtained by instantiating the general $\mathbf{Inh}^{G}_{\mathcal{U}}$ algorithm with grammar BV. Grammar BV is the image of grammar V by the CBV encoding so that $\mathbf{Inh}^{\mathsf{BV}}_{\mathcal{U}}$ *encodes* some (*direct*) algorithm solving the IP for CBV and working on system $\mathcal{V}$ and grammar V. This *direct* algorithm, called $\mathbf{Inh}_{\mathcal{V}}$, can also be obtained by the "preimage extraction", and does not pass through $\lambda!$. The rules of $\mathbf{Inh}_{\mathcal{V}}$ (Fig. 9) appear quite similar to the ones of $\mathbf{Inh}_{\mathcal{U}}$ (Fig. 4), and much more complex than the ones of the inhabitation algorithm for CBN in [Bucciarelli et al. 2014]. This suggests that the IP in CBV has the same level of complexity as in $\lambda!$, and is considerably harder than in CBN.

7 CONCLUSION

This paper solves the challenging problem of inhabitation for $\lambda!$ in the framework of *quantitative* type systems: we present an algorithm deciding the inhabitation problem for the $\lambda!$-calculus and the quantitative type system $\mathcal{U}$. For each given typing, the algorithm does not only search for one inhabitant, but generates a finite basis representing, and being able to generate, *all and only all* the possible solutions. This makes our method very powerful.

A several-for-one deal! One of the most original points of our work is the use of a grammar to parametrize the inhabitation algorithm for the $\lambda!$-calculus in order to also solve the IP for other models of computation encodable inside $\lambda!$. To the best of our knowledge, this is the first time that inhabitation is solved in such a generic way. This means in particular that the *same* algorithm can

$$\frac{\mid}{x \Vdash H_F^{x:[\sigma]}(\emptyset;\sigma)}\text{VAR-FUN} \qquad \frac{I \neq \emptyset \quad \Gamma = x:[\sigma_i]_{i\in I} \mid}{x \Vdash N(\Gamma;[\sigma]_{i\in I})}\text{VAR-VAL} \qquad \frac{\mid}{\bot_v \Vdash N(\emptyset;[\,])}\text{VAR}_\bot$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad [\mathcal{M} \Rightarrow \sigma] \Vvdash S(\tau,[\diamond \Rightarrow \sigma]) \mid a_1 \Vdash H_Q^{x:[\tau]}(\Gamma_1;[\mathcal{M}\Rightarrow\sigma]) \quad a_2 \Vdash N(\Gamma_2;\mathcal{M})}{a_1 a_2 \Vdash H_A^{x:[\tau]}(\Gamma;\sigma)}\text{APP}_Q \qquad \frac{\mid}{\lambda x.\bot \Vdash N(\emptyset;[\,])}\text{ABS}_\bot$$

$$\frac{\Gamma = \Gamma' + x:[\tau] \quad \sigma \Vvdash S(\tau,\diamond) \mid a \Vdash H_A^{x:[\tau]}(\Gamma';\sigma)}{a \Vdash N(\Gamma;\sigma)}\text{N-H}_A \qquad \frac{I \neq \emptyset \quad \Gamma = +_{i\in I}\Gamma_i \quad \texttt{fix}\ x \in \mathrm{dom}(\Gamma) \mid (a_i \Vdash N(\Gamma_i, x:\mathcal{M}_i;\sigma_i))_{i\in I} \quad \uparrow_{i\in I} a_i}{\lambda x.\bigvee_{i\in I} a_i \Vdash N(\Gamma;[\mathcal{M}_i \Rightarrow \sigma_i]_{i\in i})}\text{ABS}$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b + z:[\rho], \quad \texttt{fix}\ y \notin \mathrm{dom}(\Gamma)\cup\{x\} \quad n \in [\![0,\mathrm{sz}(\rho)]\!],\ \mathcal{M} \Vvdash S(\rho,[\diamond_1,\ldots,\diamond_n]) \mid a \Vdash H_Q^{x:[\tau]}(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash H_A^{z:[\rho]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash H_Q^{x:[\tau]}(\Gamma;\sigma)}\text{ES-H}_Q$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b, \quad \texttt{fix}\ y \notin \mathrm{dom}(\Gamma)\cup\{x\} \quad n \in [\![1,\mathrm{sz}(\tau)]\!],\ [\rho_i]_{i\in[\![1,n]\!]} \Vvdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \quad j \in [\![1,n]\!],\ \sigma \Vvdash S(\rho_j,\diamond) \mid a \Vdash H_Q^{y:[\rho_j]}(\Gamma_a, y:[\rho_i]_{i\in[\![1,n]\!]\backslash j};\sigma) \quad b \Vdash H_A^{x:[\tau]}(\Gamma_b;[\rho_i]_{i\in[\![1,n]\!]})}{a[y\backslash b] \Vdash H_Q^{x:[\tau]}(\Gamma;\sigma)}\text{ES-CH}_Q$$

$$\frac{\Gamma = \Gamma_a + \Gamma_b + z:[\tau], \quad \texttt{fix}\ y \notin \mathrm{dom}(\Gamma) \quad n \in [\![0,\mathrm{sz}(\tau)]\!],\ \mathcal{M} \Vvdash S(\tau,[\diamond_1,\ldots,\diamond_n]) \mid a \Vdash N(\Gamma_a, y:\mathcal{M};\sigma) \quad b \Vdash H_A^{z:[\tau]}(\Gamma_b;\mathcal{M})}{a[y\backslash b] \Vdash N(\Gamma;\sigma)}\text{ES-N}$$

Fig. 9. Rules of the Direct Algorithm $\mathbf{Inh}_{\mathcal{V}}$ for System $\mathcal{V}$
($Q \in \{F, A\}$ in rules (APP_Q), (ES-H_Q), (ES-CH_Q))

be used not only to search for inhabitants in $\lambda!$ but also in other languages encodable within it: this is done by just changing the parameter of the algorithm to another tree grammar. In particular, we propose two restrictions of our algorithm so as to naturally derive inhabitation algorithms for CBN and CBV λ-calculi in a quantitative framework. This is done by using (untyped and typed) appropriate embeddings of CBN/CBV into the $\lambda!$-calculus. In the first case, we use the CBN embedding by Girard, and the resulting CBN inhabitation algorithm is isomorphic to the one in [Bucciarelli et al. 2014, 2018]. In the second one, we use the CBV embedding in [Bucciarelli et al. 2020], and the resulting CBV inhabitation algorithm is *new*: we provide the first proof of decidability of the IP for a non-idempotent intersection type system that characterizes CBV normalization. Indeed, some preliminary ideas towards a possible CBV algorithm appear in [Kerinec et al. 2021], but their type system does not validate (*i.e.* subject reduction fails) permutation rules to unblock redexes [Accattoli and Guerrieri 2022a,b], and their calculus without permutations contains normal forms that are untypable. Therefore, soundness and completeness of their inhabitation algorithm are not guaranteed.

It is worth noticing that, although our method *encodes* the inhabitation problems of CBN/CBV into the one for the $\lambda!$-calculus, *direct* algorithms (Sect. 6.2 for CBN, Sect. 6.4 for CBV) can also be derived from the encoded ones. This means that the $\lambda!$ technology could also be forgotten at the end. Although we have not explored a general methodology to derive direct inhabitation algorithms for calculi encodable in $\lambda!$, we give two concrete examples, leaving the topic for future work.

A non-trivial problem. What makes difficult our algorithm with respect to the quantitative CBN case [Bucciarelli et al. 2014, 2018] is the presence of special built-in constructors in the $\lambda!$-calculus, notably dereliction and explicit substitutions (ES), which are precisely needed to subsume a reasonable version of CBV. For example, rule (der) in system $\mathcal{U}$—typing derelictions—does not only break the *subformula property* (the type in the premise is not a subtype of the conclusion), but the type in the conclusion is a strict subtype of that of the premise, a phenomenon which jeopardizes the termination of the algorithm. However, we are able to highlight a measure defined on the input parameters of the algorithm which is strictly decreasing along each recursive call. This gives a non-trivial argument for the termination property of our algorithm. Also, we aim to find *all* the solutions for a given typing, and some of these solutions may contain (nested) ES: they are neither trivial, nor easy to find.

One may think that ES are syntactically redundant, since a closure of the form $t[x\backslash s]$ can be seen as syntactic sugar for a β-like redex $(\lambda x.t)s$. But the problem would be still there. Indeed, the real issue is that in calculi like CBV and $\lambda!$ some (normal) terms may contain ES (or β-like redexes) that cannot be fired because the argument is not of the right form (a value in CBV, a bang in $\lambda!$). This problem is absent in CBN, where there is no restriction to fire a β-redex (a normal form cannot contain ES). This is one of the reasons that makes the IP for CBV or $\lambda!$ much harder than in CBN.

It is worth mentioning the existence of an inhabitation algorithm for a pattern matching language with ES appearing in [Bucciarelli et al. 2021]. However, the algorithm does not search for approximants with nested substitutions, so that it is less expressive from a program synthesis point of view, *i.e.* the algorithm does not construct a complete basis generating *all* possible answers.

More generally, the $\lambda!$-calculus not only subsumes CBN and CBV λ-calculi, but is a richer and more expressive language than CBN and CBV. This causes another difficulty to design our algorithm that does not appear in CBN and CBV: the presence of clashes, which are ill-formed terms that are meaningless even though normal. Type system $\mathcal{U}$ and our inhabitation algorithm are conceived to exclude clashes from the search space.

Future work. Using the tools developed in this paper, we are currently considering an encoding of the approximation theorems for CBN and CBV into the approximation theorem for $\lambda!$. We would like also to investigate how the natural notion of level in $\lambda!$ captures new notions of reductions for CBN and CBV. Last, but not least, we conjecture that our inhabitation algorithm will play a key role to characterize solvability in $\lambda!$. Indeed, in languages with call-by-value flavor, as for example in λ-calculus with pattern matching [Bucciarelli et al. 2021], solvability was shown to be equivalent to typability *and* inhabitation. We conjecture the same will happen in the $\lambda!$-calculus, so that the problem solved in this paper would be a crucial step forward solvability in $\lambda!$.

REFERENCES

Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. 2020. Tight typings and split bounds, fully developed. *J. Funct. Program.* 30 (2020), e14. https://doi.org/10.1017/S095679682000012X

Beniamino Accattoli and Giulio Guerrieri. 2022a. Call-by-Value Solvability and Multi Types. *CoRR* abs/2202.03079 (2022). arXiv:2202.03079 https://arxiv.org/abs/2202.03079

Beniamino Accattoli and Giulio Guerrieri. 2022b. The theory of call-by-value solvability. *Proc. ACM Program. Lang.* 6, ICFP (2022), 855–885. https://doi.org/10.1145/3547652

Beniamino Accattoli and Delia Kesner. 2010. The structural *lambda*-calculus. In *Proceedings of 24th EACSL Conference on Computer Science Logic (LNCS, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 381–395.

Beniamino Accattoli and Luca Paolini. 2012. Call-by-Value Solvability, Revisited. In *FLOPS (LNCS, Vol. 7294)*, Tom Schrijvers and Peter Thiemann (Eds.). Springer, 4–16.

Victor Arrial. 2023. *An Implementation of the Quantitative Inhabitation for Different Lambda Calculi in a Unifying Framework.* https://github.com/ArrialVictor/InhabitationLambdaBang

Henk Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics* (revised ed.). Studies in logic and the foundation of mathematics, Vol. 103. North-Holland, Amsterdam.

Jan Bessai. 2013. *Synthesizing Dependency Injection Configurations for the Spring Framework.* Master's thesis. Technical University of Dortmund.

Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro, and Jakob Rehof. 2018. Mixin Composition Synthesis based on Intersection Types. *Log. Methods Comput. Sci.* 14, 1 (2018). https://doi.org/10.23638/LMCS-14(1:18)2018

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. 2014. Combinatory Logic Synthesizer. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, Tiziana Margaria and Bernhard Steffen (Eds.). Vol. 8802. Springer Berlin Heidelberg, Berlin, Heidelberg, 26–40. https://doi.org/10.1007/978-3-662-45234-9_3 Series Title: Lecture Notes in Computer Science.

Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. 2020. The Bang Calculus Revisited. In *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12073)*, Keisuke Nakano and Konstantinos Sagonas (Eds.). Springer, 13–32. https://doi.org/10.1007/978-3-030-59025-3

Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. 2014. The Inhabitation Problem for Non-idempotent Intersection Types. In *Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS 2014, Rome, Italy, September 1-3, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8705)*, Josep Díaz, Ivan Lanese, and Davide Sangiorgi (Eds.). Springer, 341–354. https://doi.org/10.1007/978-3-662-44602-7_26

Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. 2021. Solvability = Typability + Inhabitation. *Logical Methods in Computer Science* Volume 17, Issue 1 (Jan. 2021). https://doi.org/10.23638/LMCS-17(1:7)2021

Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. 2018. Inhabitation for Non-idempotent Intersection Types. *Logical Methods in Computer Science* 14, 3 (2018). https://doi.org/10.23638/LMCS-14(3:7)2018

Alberto Carraro and Giulio Guerrieri. 2014. A Semantical and Operational Account of Call-by-Value Solvability. In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8412)*, Anca Muscholl (Ed.). Springer, 103–118. https://doi.org/10.1007/978-3-642-54830-7_7

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. 2000. Efficient resource management for linear logic proof search. *Theoretical Computer Science* 232, 1 (Feb. 2000), 133–163. https://doi.org/10.1016/S0304-3975(99)00173-5

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. *Tree Automata Techniques and Applications*. 262 pages. https://hal.inria.fr/hal-03367725

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung* 19, 1 (1978), 139–156. https://doi.org/10.1007/BF02011875

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1980. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Log.* 21, 4 (1980), 685–693. https://doi.org/10.1305/ndjfl/1093883253

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

Daniel de Carvalho. 2007. *Sémantiques de la logique linéaire et temps de calcul*. Ph. D. Dissertation. Université Aix-Marseille II.

Daniel de Carvalho. 2018. Execution time of λ-terms via denotational semantics and intersection types. *Math. Struct. Comput. Sci.* 28, 7 (2018), 1169–1203. https://doi.org/10.1017/S0960129516000396

Boris Düdder. 2014. *Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic*. Ph. D. Dissertation. https://doi.org/10.17877/DE290R-6528

Thomas Ehrhard. 2016. Call-By-Push-Value from a Linear Logic Point of View. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 202–228. https://doi.org/10.1007/978-3-662-49498-1_9

Thomas Ehrhard and Giulio Guerrieri. 2016. The Bang Calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 174–187. https://doi.org/10.1145/2967973.2968608

Claudia Faggian and Giulio Guerrieri. 2021. Factorization in Call-by-Name and Call-by-Value Calculi via Linear Logic. In *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12650)*, Stefan Kiefer and Christine Tasson (Eds.). Springer, 205–225. https://doi.org/10.1007/978-3-030-71995-1_11

Philippa Gardner. 1994. Discovering needed reductions using type theory. In *Theoretical Aspects of Computer Software*, Masami Hagiya and John C. Mitchell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 555–574.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Giulio Guerrieri and Giulio Manzonetto. 2018. The Bang Calculus and the Two Girard's Translations. In *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018 (EPTCS, Vol. 292)*. 15–30. https://doi.org/10.4204/EPTCS.292.2

Giulio Guerrieri and Federico Olimpieri. 2021. Categorifying Non-Idempotent Intersection Types. In *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference) (LIPIcs, Vol. 183)*, Christel Baier and Jean Goubault-Larrecq (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:24. https://doi.org/10.4230/LIPIcs.CSL.2021.25

Joshua S. Hodas and Dale Miller. 1994. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation* 110, 2 (May 1994), 327–365. https://doi.org/10.1006/inco.1994.1036

Jack Hughes and Dominic Orchard. 2020. Resourceful Program Synthesis from Graded Linear Types. In *Logic-Based Program Synthesis and Transformation: 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7–9, 2020, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 151–170. https://doi.org/10.1007/978-3-030-68446-4_8

Axel Kerinec, Giulio Manzonetto, and Simona Ronchi Della Rocca. 2021. Call-By-Value, Again!. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:18. https://doi.org/10.4230/LIPIcs.FSCD.2021.7

Delia Kesner and Andrés Viso. 2021. Encoding Tight Typing in a Unified Framework. *CoRR* abs/2105.00564 (2021). arXiv:2105.00564 https://arxiv.org/abs/2105.00564

Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. 1993. Combinatory Reduction Systems: Introduction and Survey. *Theor. Comput. Sci.* 121, 1&2 (1993), 279–308. https://doi.org/10.1016/0304-3975(93)90091-7

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–243. https://doi.org/10.1007/3-540-48959-2_17

Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. https://doi.org/10.1145/357084.357090

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1991. Uniform Proofs as a Foundation for Logic Programming. *Ann. Pure Appl. Log.* 51, 1-2 (1991), 125–157. https://doi.org/10.1016/0168-0072(91)90068-W

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Luca Paolini and Simona Ronchi Della Rocca. 1999. Call-By-Value Solvability. *RAIRO Theoretical Informatics and Applications* 33, 6 (1999), 507–534. https://doi.org/10.1051/ita:1999130

S Plate. 2013. *Automatische Generierung einer Konfiguration fur virtuelle Maschinen unter Zuhilfenahme eines Inhabitationsalgorithmus.* Bachelor's Thesis. Universität Dortmund, Department of Computer Science.

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism.* Academic Press, 561–577.

Pawel Urzyczyn. 1999. The Emptiness Problem for Intersection Types. *Journal of Symbolic Logic* 64, 3 (1999), 1195–1215. https://doi.org/10.2307/2586625

Anna Vasileva. 2013. *Synthese von Orchestrationscode für Cloud-basierte Dienste.* Ph. D. Dissertation. Universität Dortmund, Fakultät Informatik.

P Wolf. 2013. *Entwicklung einer Adapters mit VI Scripting (LabVIEW) zur Synthese von LEGO(R) NXT-VIs aus einem Repository.* Bachelor's Thesis. Universität Dortmund, Department of Computer Science.

Received 2022-07-07; accepted 2022-11-07