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 $\lambda!$.

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:

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 $X$ is written $\Gamma \vdash t : \sigma$, where $t$ is a term, $\sigma$ is the type assigned to $t$, and $\Gamma$ 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 $\Gamma$, a term $t$ and a type $\sigma$, decide whether $\Gamma \vdash t : \sigma$; (2) typability: given a term $t$, decide whether there exists an environment $\Gamma$ and a type $\sigma$ such that $\Gamma \vdash t : \sigma$; (3) inhabitation: given an environment $\Gamma$, and a type $\sigma$, decide whether there is a term such that $\Gamma \vdash t : \sigma$. Inhabitation corresponds to decide the existence of a program (term $t$) that satisfies the given specification (type $\sigma$) under some assumptions (environment $\Gamma$). 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.
Quantitative Type Systems. Intersection type systems [Coppo and Dezani-Ciancaglini 1978, 1980] were introduced for the \( \lambda \)-calculus to increase the typability power of simple types by introducing a new intersection type constructor \( \land \), which is, in principle, associative, commutative and idempotent (i.e. \( \sigma \land \sigma = \sigma \)). Intersection types allow terms to have different types simultaneously, e.g. a term \( t \) has type \( \sigma \land \tau \) whenever \( t \) has both the type \( \sigma \) and the type \( \tau \). 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 [Coppe 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 \( \sigma \land \sigma \land \tau \) can be seen as a resource that, during execution, can be used once as a data of type \( \tau \) and twice as a data of type \( \sigma \). The resulting non-idempotent type systems for the \( \lambda \)-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 it 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 \( \lambda \)-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 CBN 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 \( \lambda \)!-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 \( \Gamma \) and type \( \sigma \), 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 \textit{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 $\lambda$-calculus.

(2) Our approach is a \textit{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, \textit{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 \textit{all} the solutions for a given typing. This is made by a fine analysis of the completeness property: a crucial (and novel) notion of \textit{basis} is defined so that the whole infinite set of solutions can be \textit{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.

\textbf{Related works and Applications.} Our work relates to the following theoretical and applied topics:

(1) \textit{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) \textit{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 \( \lambda \)-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 \( \Sigma \) 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 \( A \rightarrow T \), where \( A \in S \) and \( T \) is a term in the associated term algebra \( T_\Sigma(S) \), 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 \( A \rightarrow T_1 \mid \ldots \mid T_n \) if all production rules \( A \rightarrow T_i \) \((1 \leq i \leq n)\) start with the same \( A \in S \). Given a production rule \( A \rightarrow T \), 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 \( A \rightsquigarrow T \) such that \( o = C(A) \) and \( p = C(T) \), where the generic notation \( C(U) \) 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 \( \rightsquigarrow^* \).

We may also make use of patterns for specific shapes of the production rules of some grammars. A pattern is noted \( G \rightsquigarrow \mathcal{H} \), where \( G \in \mathcal{J} \) 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 \( \text{Lam}(\_\) and \( \text{Bng}(\_\) of arity 1 and whose productions rules are constrained by the patterns \( G \rightsquigarrow \text{Lam}(G') \) and \( G \rightsquigarrow \text{Bng}(G') \), then the rules of \( G \) can only be of the form \( n_1 \rightsquigarrow \text{Lam}(n_2) \) and \( n_3 \rightsquigarrow \text{Bng}(n_4) \), where \( n_i \in S \) for all \( i \). This notion is inherited from meta-variables in higher-order rewriting [Klopp 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 \( X \) of variables \( x, y, z, \ldots \), the set of terms \( \Lambda \) is given by the following inductive definition:

\[
\begin{align*}
\text{(Terms)} \; t, u, s & : = \; x \in X \mid tu \mid \lambda x.t \mid !t \mid \text{der}(t) \mid t[x/u]
\end{align*}
\]

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

Abstractions \( \lambda x.t \) and closures \( t[x/u] \) bind the variable \( x \) in the term \( t \). The notions of free and bound variables are defined as expected, in particular \( \text{fv}(\lambda x.t) := \text{fv}(t) \setminus \{x\} \) and \( \text{fv}(t[x/u]) := \text{fv}(u) \cup (\text{fv}(t) \setminus \{x\}) \). The usual notion of \( \alpha \)-conversion [Barendregt 1984] is extended to the whole set \( \Lambda \), and terms are identified up to \( \alpha \)-conversion. We denote by \( t[x/u] \) 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:

\[
\begin{align*}
F & := \; \Diamond \mid Ft \mid tF \mid \lambda x.F \mid !F \mid \text{der}(F) \mid F[x/t] \mid t[x/F] \\
S & := \; \Diamond \mid St \mid tS \mid \lambda x.S \mid \text{der}(S) \mid S[x/t] \mid t[x/S] \\
L & := \; \Diamond \mid L[x/t] 
\end{align*}
\]

\( L \) and \( S \) are special cases of \( F \). The hole can occur everywhere in \( F \), while in \( S \) it cannot occur under a \( ! \). We write \( F(t) \) 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**.

\[
\begin{align*}
\text{(Distant Beta)} \; & \quad L(\lambda x.t)u \quad \mapsto_{\text{db}} \quad L(t[x/u]) \\
\text{(Substitute Bang)} \; & \quad t[x/L(\langle u \rangle)] \quad \mapsto_{\text{s!}} \quad L(t[x/u]) \\
\text{(Distant Bang)} \; & \quad \text{der}(L(\langle t \rangle)) \quad \mapsto_{\text{d!}} \quad L(t) 
\end{align*}
\]

Rule \( \text{db} \) (resp. \( \text{s!} \)) is assumed to be capture free, so no free variable of \( u \) (resp. \( t \)) is captured by the context \( L \). The rule \( \text{db} \) fires a standard \( \beta \)-redex and generates an ES. The rule \( \text{s!} \) fires an ES provided that its argument is a bang. The rule \( \mapsto_{\text{d!}} \) 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 of ES. This mechanism unblocks redexes that otherwise would be stuck, e.g. \( (\lambda x.x)[y/w]!z \mapsto_{\text{db}} x[x/z][y/w] \) fires a \( \beta \)-redex by taking \( L = \Diamond[y/w] \) as the list context in between the function \( \lambda x.x \) and the argument \( !z \).

The **surface reduction** relation \( \mapsto_S \) is the surface closure of any of the three rewrite rules \( \mapsto_{\text{db}} \), \( \mapsto_{\text{s!}} \) and \( \mapsto_{\text{d!}} \), i.e. \( \mapsto_S \) only fires redexes in surface contexts, and not under bang. Similarly, the **full**
reduction relation $\rightarrow_f$ is the full closure of any of the rewrite rules, so that $\rightarrow_f$ reduces under full contexts and thus the bang loses its freezing behavior. For example,

$$\lambda x. \text{der}(!x)!y \rightarrow_s (\text{!der}(!x))[x!/y] \rightarrow_s !\text{der}(!y) \rightarrow_f !y$$

Note that the first two steps are also $\rightarrow_f$-steps, while the last step is not an $\rightarrow_s$-step. More generally, we have $\rightarrow_s \subseteq \rightarrow_f$. For $R \in \{S, F\}$, $\rightarrow_R$ is the reflexive-transitive closure of $\rightarrow_R$.

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 $!(\text{der}(!y))$ is a surface normal form but not a full normal form since $\text{der}(!y)$ 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:

$$L(!s)u \quad s[x\langle L(\lambda x.u) \rangle \quad \text{der}(L(\lambda x.u)) \quad t(L(\lambda x.u)) \text{ if } t \neq L'(\langle \lambda y.s \rangle)$$

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 $t \rightarrow_s S(c)$. Similarly, a term $t$ is full clash-free if there are no full context $F$ and clash $c$ such that $t \rightarrow_f F(c)$. 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 $R \in \{S, F\}$, a term $t$ is said to be $R$-normalizing iff $t \rightarrow_R p$ for some $R$-normal form $p$. As $\rightarrow_s \subseteq \rightarrow_f$, 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 $\mathcal{T}V$ of type variables $\alpha, \beta, \gamma, \ldots$, we inductively define:

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

(Type) environments, noted $\Gamma, \Delta$, 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 $\Gamma$ is $\text{dom}(\Gamma) = \{x \in \mathcal{X} \mid \Gamma(x) \neq []\}$. Given the environments $\Gamma$ and $\Delta$, $\Gamma + \Delta$ is the environment mapping $x$ to $\Gamma(x) \cup \Delta(x)$, where $\cup$ denotes multiset union; and $+_i I\Delta_i$ is its obvious extension to the non-binary case, in particular $+_i I\Delta_i = \emptyset$ if $I = \emptyset$. We use $\Gamma, \Delta$ to denote $\Gamma + \Delta$ when $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$, thus $x_1 : M_1, \ldots, x_n : M_n$ is the environment assigning $M_i$ to $x_i$ for $1 \leq i \leq n$, and $[]$ to any other variable. We write $\Gamma\downharpoonright x$ for the environment assigning $[]$ to $x$, and acting as $\Gamma$ otherwise.

A typing judgment is a triple of the form $\Gamma + t : \sigma$, where $\Gamma$ is a typing environment, $t$ is a term (called the subject of the typing judgment), and $\sigma$ 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...
We aim to provide an algorithm solving the inhabitation problem for any given input typing \(\Gamma\). Still, given a typing \(\Gamma\), the corresponding solution set—the set of terms typable with \(\sigma\) in the environment \(\Gamma\)—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 (naive) 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 \(\Omega\) 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 \(l(t)\) of locations of \(t \in \Lambda\) is the set of contexts \(F\) such that \(F(u) = t\) for some \(u\). Given \(\Pi_t \vdash \Gamma \vdash t : \sigma\), the set \(tl(\Pi_t) \subseteq l(t)\) of typed locations of \(t\) in \(\Pi_t\) is defined by induction on \(\Pi_t\) (we refer to the names of the rules in Fig. 1):

- **ax:** \(tl(\Pi_t) := \{\Diamond\};\)
- **es:** \(tl(\Pi_t) := \{\Diamond\} \cup \{F[x/s] | F \in tl(\Pi_u)\} \cup \{u[x/F] | F \in tl(\Pi_u)\} \quad \text{with } t = u[x/s], \text{premises } \Pi_u, \Pi_s;\)
- **abs:** \(tl(\Pi_t) := \{\Diamond\} \cup \{\lambda x. F | F \in tl(\Pi_u)\} \quad \text{with } t = \lambda x. u \text{ and premise } \Pi_u;\)
- **app:** \(tl(\Pi_t) := \{\Diamond\} \cup \{Fs | F \in tl(\Pi_u)\} \cup \{uF | F \in tl(\Pi_u)\} \quad \text{with } t = us \text{ and premises } \Pi_u, \Pi_s;\)
- **bng:** \(tl(\Pi_t) := \{\Diamond\} \cup \{\Pi_{i \in I} !F | F \in tl(\Pi_u)\} \quad \text{with } t = !u \text{ and premises } \Pi_{i \in I};\)
- **der:** \(tl(\Pi_t) := \{\Diamond\} \cup \{\Pi_{i \in I} \} \quad \text{with } t = \Pi_{i \in I} \text{ and premise } \Pi_u.\)

Given a derivation \(\Pi \vdash \Gamma \vdash t : \sigma\), a redex \(r\) in \(t\) is a typed redex in \(\Pi\) if \(t = F(r) \text{ and } F \in tl(\Pi)\).

**Example 3.2.** In the derivation in (2) below, the term \(\Omega\) is not a typed redex in \(\Pi\) (even though \(\Omega\) is a redex), because \((\lambda x. y)!\Omega = F(\Omega)\) with \(F = (\lambda x. y)!\Diamond \notin tl(\Pi)\). However, the redex \((\lambda x. y)!\Omega\) is a typed redex in \(\Pi\) because \((\lambda x. y)!\Omega = F((\lambda x. y)!\Omega)\) with \(F = \Diamond \in tl(\Pi)\).

\[
\Pi = \begin{array}{c}
y : [\sigma] \vdash y : \sigma \\
\frac{y : [\sigma] \vdash \lambda x. y : [\text{l } \Rightarrow \sigma]}{y : [\sigma] \vdash (\lambda x. y)!\Omega : \sigma} \quad \text{ax}\end{array}^{\text{app}} \quad \text{bng}\]

(2)

Given \(\Pi \vdash \Gamma \vdash t : \sigma\), we say that \(\Pi\) is a normal (type) derivation, written \(\Pi^n\Gamma \vdash t : \sigma\), if for all \(F \in tl(\Pi), t = F(u)\) implies that \(u\) is not a typed redex in \(\Pi\). A term \(s\) is said to be a normal solution for the typing \((\Gamma; \sigma)\) if there exists a normal type derivation \(\Pi\) such that \(\Pi^n\Gamma \vdash s : \sigma\). Thus e.g. the derivation \(\Pi\) in (1) is normal and the term \(x!\Omega\) is a normal solution for \((x : [\lfloor x \rfloor \Rightarrow \sigma]; \sigma)\). However, \(\Pi\) 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 \(\Pi \vdash \Gamma \vdash t : \sigma\) and \(t \rightarrow_{F} t'\). Given \(\Pi' \vdash \Gamma \vdash t' : \sigma\) obtained by Thm. 2.1.1, \((\Pi, t)\) typed reduces to \((\Pi', t')\), noted \((\Pi, t) \rightarrow_{T} (\Pi', t')\), if there exists a typed location \(F \in tl(\Pi)\) and two terms \(u, u'\) such that \(t = F(u)\) and \(t' = F(u')\) with \(u \rightarrow_{R} u'\) and \(R \in \{dB, s', !d\}\).
Example 3.4. Given \( \Pi \) as in (2) and \( \Pi' \), \( \Pi'' \) as below, we have \( (\Pi, (\lambda x.y)!\Omega) \rightarrow (\Pi', y) \), since \( F = \diamond \) is a typed location in \( \Pi \) and \( (\lambda x.y)!\Omega \rightarrow y \); but \( (\Pi, (\lambda x.y)!\Omega) \not\rightarrow (\Pi'', (\lambda x.y)!((z)\underbrace{[z]_{\upharpoonright \Delta}}_\text{bang})) \) because \( (\lambda x.y)!\diamond \) is not a typed location in \( \Pi \), even though \( (\lambda x.y)!\Omega \rightarrow (\lambda x.y)!((z)\underbrace{[z]_{\upharpoonright \Delta}}_\text{bang}) \).

\[
\Pi' = \frac{y : [\sigma] \triangleright y : \sigma}{\Pi'' = \frac{y : [\sigma] \triangleright \lambda x.y : [\ ] \rightarrow \sigma}{\frac{\emptyset \triangleright !((z)\underbrace{[z]_{\upharpoonright \Delta}}_\text{bang}) : [\ ]}{}}^{\text{bng}}
\]

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

**Theorem 3.5 (Weighted Subject Reduction).** If \( (\Pi, t) \rightarrow (\Pi', t'), then \#(\Pi) > \#(\Pi'). \) So, \( \rightarrow_t \) 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 \( \Pi \) in (1), the untyped subterm \( \Omega \) 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 type calculus with a set of constants \( C \), the resulting set of \( C \)-terms, denoted by \( \Lambda_C \), is inductively defined as follows:

\[
a, b ::= c \in C \mid x \in X \mid \lambda x.a \mid ab \mid a[x/b] \mid !a \mid \text{der}(a)
\]

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

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

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

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

- \( \text{ax} : \mathcal{A}(\Pi) := x \), with \( a = x \);
- \( \text{es} : \mathcal{A}(\Pi) := \mathcal{A}(\Pi_b) \setminus \mathcal{A}(\Pi_c) \), with \( a = b[x/c] \), and premises \( \Pi_b, \Pi_c \);
- \( \text{abs} : \mathcal{A}(\Pi) := \lambda x.\mathcal{A}(\Pi_b) \), with \( a = \lambda x.b \), and premise \( \Pi_b \);
- \( \text{app} : \mathcal{A}(\Pi) := \mathcal{A}(\Pi_b) \setminus \mathcal{A}(\Pi_c) \), with \( a = bc \), and premises \( \Pi_b, \Pi_c \);
- \( \text{bng} : \mathcal{A}(\Pi) := !\bigvee_{i \in I} \mathcal{A}(\Pi^b_i) \), with \( a = !b \), and premises \( \Pi^b_i \in I \);
- \( \text{der} : \mathcal{A}(\Pi) := \text{der}(\mathcal{A}(\Pi_b)) \), with \( a = \text{der}(b) \), and premise \( \Pi_b \).
Thus, e.g., $\mathcal{A}(\Pi) = x!\perp$ for the normal type derivation $\Pi$ 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 $a \in \Lambda_\perp$, $\uparrow_{i\in I} a_i$ holds for every $\{a_i\}_{i\in I} \subseteq \{b \in \Lambda_\perp \mid b \leq a\}$

**Lemma 3.7.** For every $a \in \Lambda_\perp$, if $\Pi \nvdash a : \sigma$ then $\mathcal{A}(\Pi) \leq a$.

The set $\Lambda_\perp$ 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 $\Sigma_C$ is given by the zero-ary symbols in $C \cup \{\text{Var}\}$, the unary symbols $\text{Der}(\_)$, $\text{Bng}(\_)$ and $\text{Lam}(\_)$, and the binary symbols $\text{App}(\_\_, \_)$ and $\text{Sub}(\_\_, \_\_)$. Such grammars will be used to generate terms with constants. Indeed, a term $a \in \Lambda_C$ is an instance of $o \in T_C(S)$, written $c I o$, if:

\[
\begin{align*}
\begin{array}{llllllll}
 x \in X & c \in C & \text{der}(a) I \text{Der} & a I o & a I o \ x \in X & \lambda x.a I \text{Lam} & a I o \ b I p & a I o \ b I p \ x \in X
\end{array}
\end{align*}
\]

A C-term $a \in \Lambda_C$ is produced by a nonterminal symbol $n \in S$, noted $a \in n$, if there is an $o \in T_C(S)$ such that $n \rightsquigarrow o$ and $a$ is an instance of $o$; and $a$ is produced by the start symbol $s$ of the grammar $G$. We set $L(G) := \{a \in \Lambda_C \mid a \in G\}$.

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) $\perp$-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 $C = \{\perp\}$, the set of nonterminal symbols is $\{\text{cne}, \text{cna}, \text{cnb}, \text{cno}\}$, the start symbol is cno, and the set of production rules is given below:

\[
\begin{align*}
\text{cne} \rightsquigarrow \text{Var} & \mid \text{App}\text{(cne, cna)} \mid \text{Der}\text{(cne)} \mid \text{Sub}\text{(cne, cne)} \mid \text{Bng}\text{(cno)} \mid \text{Sub}\text{(cna, cne)} \mid \text{Bng}\text{(\perp)} \\
\text{cna} \rightsquigarrow \text{cne} & \mid \text{Bng}\text{(cno)} \mid \text{Bng}\text{(\perp)} \mid \text{Sub}\text{(cna, cne)} \mid \text{cno} \rightsquigarrow \text{cna} \mid \text{cnb}
\end{align*}
\]

If $a \in \text{cno}$, then $a$ is called a (clash-free) canonical $\perp$-term; and if $a \in \text{cne}$, then $a$ is called a (clash-free) canonical neutral $\perp$-term.

**Example 3.10.** We have $zy \in \text{cne}$. Indeed, $zy \ I \text{App(Var, Var)}$ and $\text{cne} \rightsquigarrow \text{App}\text{(cne, cna)} \rightsquigarrow \text{App}\text{(cne, cne)} \rightsquigarrow \text{App}\text{(Var, cne)} \rightsquigarrow \text{App}\text{(Var, Var)}$.

Similarly, it can be shown that $(yx)[x^\prime z y] \in \text{cne}$, $\perp \text{(\lambda x.x)}[x^\prime z y] \in \text{cno}$ and $\perp \text{(\lambda x.x)[x^\prime \ \text{zy}]} \in \text{cnb}$.

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

**Proposition 3.11.** A $\perp$-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 $\Pi \nvdash a : \sigma$ with $a = A(\Pi)$; we then say that $a$ is a canonical solution for the typing $(\Gamma; \sigma)$. Thus e.g. the derivation $\Pi$ in (1) is normal but not canonical, while the derivation $\Pi'$ below is canonical, and $A(\Pi') = x!\perp$ is a canonical solution for the typing $(x : \! x \Rightarrow a : a)$. $\Pi'$ is given by the zero-ary symbols in $\{\text{cnb}\}$, written $c I o$, if:

\[
\begin{align*}
\begin{array}{llllll}
 x : \! x & \Rightarrow a & \Rightarrow a^x & \emptyset & \vdash \perp : \! x \Rightarrow a \\
 x : \! x & \Rightarrow a & \vdash x ! \perp : a
\end{array}
\end{align*}
\]

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 $\Pi \nvdash a : \sigma$ with $a \in \Lambda_\perp$. Then, $A(\Pi)$ 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:

\[
\text{Basis}(\Gamma; \sigma) := \{ b \in \Lambda_{\bot} \mid b \text{ canonical solution for }(\Gamma; \sigma)\}
\]

Next theorem guarantees that \(\text{Basis}(\Gamma; \sigma)\) 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 \(S \subseteq \Lambda_{\bot}\) as the set of terms obtained by full expansion of redexes of terms greater than the elements of \(S\):

\[
\text{Span}(S) := \{ t \in \Lambda \mid \exists a \in S, \exists u \in \Lambda, a \leq u \text{ and } t \rightarrow_f u \}
\]

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

**Theorem 3.13 (Sound & Complete Basis).** For every typing \((\Gamma; \sigma)\), \(\text{Span}(\text{Basis}(\Gamma; \sigma)) = \text{Sol}(\Gamma; \sigma)\).

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 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 \(\text{app}\) in Fig. 1. The type \(M\) 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 \(M\), 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 \(\bot\)-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 \(\bot\)-terms (Def. 3.9) are built from variables, applications, derelictions and ES. In the last three corresponding typing rules (\(\text{app}\), \(\text{der}\) and \(\text{es}\) 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 \(\bot\)-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 \(\Pi\) which is the type given by \(\Pi\) 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.
4. The subtype relation \( \preceq \) is a binary relation on types defined by the rules below.

\[
\begin{align*}
\frac{\tau = \sigma}{\tau \preceq \sigma}^{\text{refl}} & \quad \frac{\tau \preceq M \text{ or } \tau \preceq \rho}{\tau \preceq (M \Rightarrow \rho)}^{\text{arrow}} & \quad \frac{\exists j \in I, \tau \preceq \rho_j}{\tau \preceq [\rho_j]_{i \in I}}^{\text{mult}}.
\end{align*}
\]

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.,

\[ [[]] \Rightarrow [\rho_2, \rho_1] \preceq [[]] \Rightarrow \alpha, [[]] \Rightarrow [\rho_1, \rho_2] \Rightarrow \alpha' \]  

(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 \( \Pi \vdash \Gamma \vdash a : \sigma \) with \( a \in \text{cne} \), then \( \sigma \preceq \text{sht}(\Pi) \) and \( \Gamma(\text{shv}(a)) = M \uplus [\text{sht}(\Pi)] \) for some \( M \).

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 \( M \) 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 \( p(\tau_1, \cdots, \tau_n) \) for the type obtained by replacing each placeholder \( \Diamond_i \) by the type \( \tau_i \). For example, if \( p = [\Diamond_1, \sigma] \Rightarrow \Diamond_2 \) then \( p(\tau_1, \tau_2) = [\tau_1, \sigma] \Rightarrow \tau_2 \).

**Definition 4.4.** Let \( \tau, \sigma \) be types and \( \rho \) a partial type. Then \( \tau \) is a **subtype of \( \sigma \) with partial knowledge \( \rho \)**, noted \( \tau \preceq_\rho \sigma \), if \( \tau \preceq \sigma \) and \( \tau = p(\tau_1, \cdots, \tau_n) \) 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 = [\rho_1, \rho_2] \Rightarrow [[]] \Rightarrow \alpha, [[]] \Rightarrow [\rho_1, \rho_2] \Rightarrow \alpha' \].

Given a term \( \sigma \) and a partial type \( \rho \), the **Subtype with Partial Knowledge (SPK) algorithm** yields a type \( \tau \) such that \( \tau \preceq_\rho \sigma \) if such a type exists, and otherwise fails. When the partial shape is empty, i.e. \( \rho = \Diamond \), the SPK algorithm just yields a subtype of the input \( \sigma \). The rules of the SPK algorithm are presented in Fig. 2: the notation \( \tau \parallel S(\sigma, p) \) is formed of a call \( S(\sigma, p) \) with input.
\[
\exists r_1, \ldots, r_n. \quad \sigma = p(r_1 \cdots r_n) \quad \frac{\text{match}}{p(r_1 \cdots r_n) \Vdash S(\sigma, p)}
\]

\[
\exists i \in I. \quad \tau \Vdash S(\sigma_i, p) \quad \frac{\text{bag}}{\tau \Vdash S([\sigma_i]_{i \in I}, p)}
\]

\[
\tau \Vdash S(M, p) \quad \frac{\text{refl}}{\tau \Vdash S(M \Rightarrow \sigma, p)}
\]

\[
\tau \Vdash S(M) \Rightarrow \sigma, p \quad \frac{\Rightarrow_1}{\tau \Vdash S(\sigma, p)}
\]

\[
\tau \Vdash S(\sigma, p) \quad \frac{\Rightarrow_2}{\tau \Vdash S(M \Rightarrow \sigma, p)}
\]

Fig. 2. Rules of the SPK Algorithm.

(\sigma, p) and an answer \(\tau\) 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 (\text{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 \(\sigma\) 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: \([\_ \Rightarrow [\rho_2, \rho_1] \Vdash S(\alpha, \Diamond \Rightarrow [\rho_1, \rho_2])\).

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

As expected, the SPK algorithm is sound and complete:

**Lemma 4.5 (Soundness and Completeness of SPK).** Let \(\tau, \sigma\) be types and \(p\) be a partial type. Then, \(\tau \preceq_p \sigma\) if and only if \(\tau \Vdash S(\sigma, p)\).

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:

\[
sz(\alpha) := 1
\]

\[
sz([\sigma_i]_{i \in I}) := \sum_{i \in I} sz(\sigma_i) + 1
\]

\[
sz(M \Rightarrow \sigma) := sz(M) + sz(\sigma) + 1
\]

**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 \(\sigma\) with partial knowledge \(p\).

5 THE INHABITATION ALGORITHM

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

5.1 General Form of the Algorithm Rules

The \(\text{Inh}_U\) algorithm aims to build any possible type derivation for a given typing. Due to the properties shown in Sect. 3, \(\text{Inh}_U\) only returns canonical \(\bot\)-terms, since Thm. 3.13 ensures that this is sufficient to span the whole solution set. Our algorithm \(\text{Inh}_U\) is then intimately associated with grammar \(B\) from Def. 3.9, which produces all possible canonical \(\bot\)-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 \(U\) has at least a corresponding rule in \(\text{Inh}_U\), declined in variants as we will explain. The algorithm builds a tree having the rules of \(\text{Inh}_U\) for (hyper) edges and sequents of the form \(a \vdash \Diamond \text{Call}\) for nodes, where
Call is any kind of call (different kinds of calls are detailed later), and \( a \) is an answer (a canonical \( \bot \)-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 \( _{\bot} \vdash \text{Call} \), corresponding to the conclusion of some rule of the algorithm (meaning that \( \text{Inh}_U \) is searching for an answer for the call \( \text{Call} \)), the algorithm performs recursive calls \( _{\bot} \vdash \text{Call}_1, \ldots, _{\bot} \vdash \text{Call}_n \) —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 \( \text{Call} \), written \( \odot(a_1, \ldots, a_m) \vdash \text{Call} \), where \( \odot \) is any term constructor of \( \lambda! \), is built from the premises answers \( a_1 \vdash \text{Call}_1, \ldots, a_m \vdash \text{Call}_n \).

Moreover, at any node, the set of applicable rules is regulated by some preliminary requirements. The general form of the rules of the algorithm \( \text{Inh}_U \) (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 \( \text{Inh}_U \) 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 \( \sigma \vdash S(\tau, \odot) \) with respect to the type \( \tau \) 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 \( \text{Inh}_U \) 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 \( \text{Inh}_U \) has two different kinds of calls, generically denoted \( \text{Call} \): 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 \( H^{\leq n}[\tau](\Gamma; \sigma) \). 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 \( \text{cno} \) of grammar \( B \), and similarly for \( N_A \), \( N_B \) and \( H \) with respect to the symbols \( \text{cna}, \text{cnb} \) and \( \text{cne} \), respectively. Some rules are represented parametrically using a generic call \( N_P \) with \( P \in \{A, B\} \), to denote either \( N_A \) or \( N_B \).
Run. Given an input typing \((\Gamma; \sigma)\), a run of \(\text{Inh}_U\) 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 (BG), 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, 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 \(\text{Inh}_U\) algorithm for the type system \(U\) 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 \(m \leq n\).

We chose \(H\)-calls \(H^x[r](\Gamma; \sigma)\) to represent searches for solutions with a specific semantic head \(x : [r]\). 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 \(\sigma\). First, a variable \(x\) must be isolated in the environment \(\Gamma\)—this is done by the splitting requirement \(\Gamma = \Gamma' + x : [r]\)—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 \(\alpha\)-conversion, the bound variable chosen in the answer is arbitrary fixed once and for all outside the domain of its environment \(\Gamma\).

Rules (ES-H) and (ES-CH) correspond to the search for an inhabitant \(a[y \backslash b]\) with a given semantic head \(x : [r]\). 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 \( \Gamma_a \) and \( \Gamma_b \). The type \( M \) 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 \( n \in \mathbb{I} \left[ 0, \text{sz}(\rho) \right] \), \( M \vdash S(\rho, [\Diamond_1, \ldots, \Diamond_n]) \). Note that if \( n = 0 \), the requirement becomes \( M \vdash S(\rho, [\Diamond]) \).

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 \( [\rho_i]_{i \in \llbracket 1, n \rrbracket} \) 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 \( n \in \mathbb{I} \left[ 1, \text{sz}(\tau) \right] \), \( [\rho_i]_{i \in \llbracket 1, n \rrbracket} \vdash S(\tau, [\Diamond_1, \ldots, \Diamond_n]) \). 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 \( j \in \mathbb{I} \left[ 1, n \right] \), \( \sigma \vdash S(\rho_j, \Diamond) \).

The typing (bng) rule of system \( U \) has two corresponding rules in \( \text{Inh}_U \): a special rule (BG⊥) handling the case where the constant \( \bot \) 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 \( \text{Inh}_U \) the recursive calls do not guarantee this property. This is why (BG) explicitly requires \( \lceil \in \mathbb{I} \ a_i \). The other requirement splits the environment as expected.

**Example 5.1.** Let us see an example of the IP in \( \lambda \) using the algorithm \( \text{Inh}_U \) for the input typing \( (x : [[[\alpha]]]; \alpha) \). All the solutions given by the algorithm are:

\[
\text{der} (\text{der} (x)) \quad \text{der} (y) [y \set x] \quad \text{der} (y [y \set x]) \quad z [z \set y] [y \set x] \quad y [y \set \text{der} (x)] \quad z [z \set y [y \set x]].
\]

We describe a particular run of the algorithm finding the solution \( \text{der} (\text{der} (x)) \) from the starting call \( N(x : [[[\alpha]]]; \alpha) \). The run starts by applying rule (N-N,\( \alpha \)), followed by rule (N,\( \alpha \)-H). In the latter, the head \( x : [[[\alpha]]] \) is chosen from the environment and the subtype inequation \( \alpha \leq [[[\alpha]]] \) is checked, which yields the call \( H^{x: [[[\alpha]]]}(\emptyset; \alpha) \). Then, rule (DR) is applied twice, with the corresponding subtype verifications \( [\alpha] \leq [[[\alpha]]] \) and \( [[[\alpha]]] \leq [[[\alpha]]] \). Finally, rule (VAR) is applied and the solution \( \text{der} (\text{der} (x)) \) 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 \( M \) such that \( M \Rightarrow [\alpha] \leq [[[\alpha]]] \), and therefore the entire run would have failed.

We now describe another particular run of the algorithm finding the solution \( y [y \set \text{der} (x)] \) from the starting call \( N(x : [[[\alpha]]]; \alpha) \). The run starts by applying rule (N-N,\( \alpha \)), followed by rule (ES-N,\( \alpha \)). In the latter, the environment is split into two empty environments, plus a head \( x : [[[\alpha]]] \). The subtype inequation \( M \leq [[[\alpha]]] \) is solved by taking \( M = [\alpha] \). Two calls are then carried out separately: on the one hand, a fresh variable \( y \) is fixed and a call \( N,\alpha (y : [\alpha]; \alpha) \) is carried out. By applying rules (N,\( \alpha \)+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 \( \text{der} (x) \) as a solution to this second recursive call. Finally, both solutions are assembled using an explicit substitution to build the solution \( y [y \set \text{der} (x)] \) for the initial call.
5.3 Properties of the Algorithm $\text{Inh}_U$

The first crucial property of algorithm $\text{Inh}_U$ is the following one:

**Theorem 5.2 (Termination of $\text{Inh}_U$).** The algorithm $\text{Inh}_U$ 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 $\text{Inh}^G_U$. As for now, we discuss soundness and completeness.

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

We prove that actual and expected answers coincide:

**Theorem 5.3 ($\text{Inh}_U$ is sound & complete).** For any typing $(\Gamma; \sigma)$, $\text{Inh}_U(\Gamma; \sigma) = \text{Basis}(\Gamma; \sigma)$.

The outputs of $\text{Inh}_U$ “span” the solution set for the IP on a typing input.

**Corollary 5.4 (Sound & complete solution to IP).** For any typing $(\Gamma; \sigma)$, $\text{Span}(\text{Inh}_U(\Gamma; \sigma)) = \text{Sol}(\Gamma; \sigma)$.

**Proof.** We have $\text{Span}(\text{Inh}_U(\Gamma; \sigma)) = \text{Thm. 5.3 Span}(\text{Basis}(\Gamma; \sigma)) = \text{Thm. 3.13 Sol}(\Gamma; \sigma)$. 

To summarize, the output of the algorithm $\text{Inh}_U(\Gamma; \sigma)$ yields a compact representation of all possible terms typable with the given input $(\Gamma; \sigma)$. Thus, $\text{Inh}_U(\Gamma; \sigma)$ 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 $\text{Inh}_U$ algorithm for system $\mathcal{U}$ is intimately linked to grammar $B$ (Def. 3.9) of canonical solutions: different calls $N(\Gamma; \sigma), N_0(\Gamma; \sigma), N_b(\Gamma; \sigma)$ and $H^{\mathsf{in}(\ell)}(\Gamma; \sigma)$ are introduced to account for the search in different subgrammars (cno, cna, cnb and cne, respectively). The grammar $B$ 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 $\text{Inh}_U$ cannot only be guided by the grammar $B$, but also by a more general class of grammars, called $\mathsf{NH}$-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 $\mathsf{NH}$-grammar is an $\{\bot\}$-grammar (see Def. 3.8) $G = (\Sigma, S, S, \mathcal{G}, \mathcal{H}, R, s)$ such that $S \cap \mathcal{H} = \emptyset$, the start symbol $s \in S$, and the production rules in $R$ follow the patterns below:

$\mathcal{G} \rightsquigarrow \mathcal{G}'$ \quad $\mathcal{G} \rightsquigarrow \mathcal{B}(\bot)$ \quad $\mathcal{G} \rightsquigarrow \mathcal{B}(\mathcal{G}')$ \quad $\mathcal{G} \rightsquigarrow \mathcal{L}(\mathcal{G}')$ \quad $\mathcal{G} \rightsquigarrow \mathcal{S}(\mathcal{G}_1, \mathcal{G}_2)$ \quad $\mathcal{G} \rightsquigarrow \mathcal{G}'$

$\mathcal{G} \rightsquigarrow \mathcal{G}'$ \quad $\mathcal{G} \rightsquigarrow \mathcal{V}(\mathcal{G}')$ \quad $\mathcal{G} \rightsquigarrow \mathcal{D}(\mathcal{G}')$ \quad $\mathcal{G} \rightsquigarrow \mathcal{A}(\mathcal{G}_1, \mathcal{G}_2)$ \quad $\mathcal{G} \rightsquigarrow \mathcal{S}(\mathcal{G}_1, \mathcal{G}_2)$

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 $n \in S_N \cup S_H$, $n \rightsquigarrow n$ does not hold;
2. For all rule $n \rightsquigarrow \mathcal{B}(n')$ in $R$ with $n' \in S_N$, for all $\{a_i\}_{i \in I} \subseteq \Lambda$, such that $\mathcal{B}_i \in a_i$, if $a_i \in n'$ for all $i \in I$ then $\mathcal{B}_i \in n$. 

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

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

From now on, the notation $\text{Inh}_H^G$ 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 $\text{Inh}_H^G$, 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 $a \models^g \text{Call}$, where $g$ is a nonterminal symbol of the NH-grammar $G$, considered as the parameter of $\text{Call}$. 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 $g \mapsto g'$ 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 $g \mapsto \text{Var}$ belongs to the associated grammar $G$. In the case of grammar $G = B$, this only happens if $g = \text{cne}$. A more interesting example is rule (N-N), where a call $N(\Gamma; \sigma)$

---

Fig. 5. Rules of the Parametric Algorithm $\text{Inh}_H^G$ for System $U$
with parameter $g$ may occur with any kind of silent rule $g \rightsquigarrow g'$ of the grammar $G$ (there are four of such silent rules in the particular case $G = B$: $\text{cno} \rightsquigarrow \text{cna}$ | $\text{cnb}$ and $\text{cnb} \rightsquigarrow \text{cne}$ and $\text{cna} \rightsquigarrow \text{cne}$).

Given an input typing $(\Gamma; \sigma)$, a run of $\text{Inh}^G_u$ 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 $g \rightsquigarrow g'$ in the grammar $G$. They are called \textit{silent rules}, while the others are called \textit{non-silent rules}.

\textbf{Example 5.8.} Let us see now some examples of the IP in $\lambda!$ using the parametric algorithm $\text{Inh}^B_u$, that is, $\text{Inh}^G_u$ where $G$ is instantiated by the particular NH-grammar $B$. We show the answers given by our implementation \cite{Arrial23} in the mode \texttt{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 $\text{Inh}^B_u$.

\begin{verbatim}
--- Example 1 ---
Inh N(x: [[a]]): \sigma
Sol > \{z: y=x, y:=\text{der}(x), z:=y[y=x],
\text{der}(y[y=x]), \text{der}(\text{der}(x))\}
--- Example 2 ---
Inh N(\emptyset; \[[\alpha] \rightarrow \alpha\]): \sigma
Sol > \lambda x . x , \lambda x . \lambda y . x y
--- Example 3 ---
Inh N(\emptyset; \[[\alpha] \rightarrow \alpha\]): \sigma
Sol > \lambda x . x , \lambda x . \lambda y . x y , \lambda y . \lambda z . (\{x:=y[z]\},
\lambda y . \lambda z . \{x:=y[z]\}, \lambda x . \lambda y . \text{der}(x[y]))

--- Example 4 ---
Inh N(\emptyset; \{[\alpha] \rightarrow \alpha\}): \sigma
Sol > x!\perp
--- Example 5 ---
Inh N(x: [[\alpha] \rightarrow \alpha\]): \sigma
Sol > \lambda x . x

--- Example 6 ---
Inh N(\emptyset; \{[\alpha] \rightarrow \alpha\}): \sigma
Sol > \lambda x . x
\end{verbatim}

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

A comparison between Examples 5.1 and 5.8 shows—with an example—that algorithm $\text{Inh}^B_u$ behaves like $\text{Inh}^B$, 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.

\textbf{Proposition 5.9.} For every typing $(\Gamma; \sigma)$, $\text{Inh}^B_u(\Gamma; \sigma) = \text{Inh}^B(\Gamma; \sigma)$.

\subsection{5.5 Properties of the Parametric Algorithm $\text{Inh}^B_u$}

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

\textit{Termination.} The $\text{Inh}_u$ 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 $\text{Inh}_u$ (Thm. 5.2) turns out to be a corollary of termination of $\text{Inh}^B_u$ (Cor. 5.14).

\textit{Finite Depth.} We prove that every run of the algorithm $\text{Inh}^B_u$ terminates by building a decreasing measure along recursive calls, which is defined in terms of the input of these calls. Some rules...
of \( \text{Inh}_U^G \) clearly invoke recursive calls with smaller inputs, for example \((\text{BG})\) and \((\text{ABS})\). Other rules invoke recursive calls with subtypes in the input provided by the SPK algorithm, for example, \((\text{DR})\), \((\text{APP})\), \((\text{ES-H})\), \((\text{ES-CH})\) and \((\text{ES-N})\); 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:

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

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

\[
\text{dpt}(\alpha) := 0 \quad \text{dpt}(M \Rightarrow \sigma) := \text{dpt}(M) + \text{dpt}(\sigma) \quad \text{dpt}([\sigma_i]_{i \in \{1, \ldots, n\}}) := n + \sum_{i \in \{1, \ldots, n\}} \text{dpt}(\sigma_i).
\]

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

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

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

\[
\text{m(Call)} := (\text{sz(Call)}, \text{dpt(Call)}).
\]

We can use the lexicographic order \( >_{1\text{ex}} \) to compare the measures of two related calls. In particular, given two calls \( c, c' \) such that \( c \) calls \( c' \), then \( \text{m}(c) \geq_{1\text{ex}} \text{m}(c') \). 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 \( \text{NH-grammar} \) (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 \( \text{NH-grammar} \) is bounded.*

Combining these two lemmas gives the first termination argument.

**Lemma 5.12 (Run Termination).** *Every run of the \( \text{Inh}_U^G \) 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 \( \text{Inh}_U^G \), 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 \( \text{Inh}_U^G \) 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 \( \text{Inh}_U^G \)).** *The \( \text{Inh}_U^G \) algorithm terminates.*

Soundness and completeness of \( \text{Inh}_U \) (Thm. 5.3) can be generalized to \( \text{Inh}_U^G \) for any \( \text{NH-grammar} \) \( G \). The idea is that nonterminal symbols in \( S_N \) (resp. \( S_H \)) are used in \( \text{Inh}_U^G \) to drive \( N \)-calls (resp. \( H \)-calls). Condition 2 in the definition of \( \text{NH-grammar} \) (Def. 5.5) is crucial to prove completeness.
Theorem 5.15 (The parametric algorithm is sound and complete). Let \( G \) be an \( \mathcal{NH} \)-grammar. For any typing \( (\Gamma;\sigma) \), \( \text{Inh}^G_{\mathcal{U}}(\Gamma;\sigma) = \text{Basis}(\Gamma;\sigma) \cap \mathcal{L}(G) \).

To summarize, a major consequence of the parametrization of the algorithm \( \text{Inh}^G_{\mathcal{U}} \) by any \( \mathcal{NH} \)-grammar \( G \) is that we can now use the same algorithm \( \text{Inh}^G_{\mathcal{U}} \) 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 \( \text{Inh}^G_{\mathcal{U}} \) 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 \( \mathcal{BN} \) and \( \mathcal{BV} \) to solve the inhabitation problems for the CBN and CBV cases by instantiating the generalized \( \text{Inh}^G_{\mathcal{U}} \) algorithm (Sect. 5.2). Indeed, both grammars \( \mathcal{BN} \) and \( \mathcal{BV} \) are instances of the general class of \( \mathcal{NH} \)-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 \( \lambda \)-calculi with ES, as in [Accattoli and Paolini 2012]. For both calculi, the set \( \Lambda_{\lambda} \) of terms is inductively defined as follows (note that \( \mathbf{der} \) and \( \mathbf{!} \) are absent):

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

**Full contexts** \( \mathcal{F} \) 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:

\[
(\text{List Contexts}) \quad L ::= \Diamond \mid L[x\backslash t] \\
(\text{CBN Contexts}) \quad N ::= \Diamond \mid Nu \mid \lambda x.N \mid N[x\backslash u] \\
(\text{CBV Contexts}) \quad V ::= \Diamond \mid Vu \mid \iota V \mid V[x\backslash u] \mid t[x\backslash V]
\]

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

\[
L(\lambda x.t)u \rightarrow_{\mathbf{dB}} L(t[x\backslash u]) \quad t[x\backslash u] \rightarrow_{\mathbf{NS}} t(x\backslash u) \quad t[x\backslash L(\mathbf{v})] \rightarrow_{\mathbf{VS}} L(t[x\backslash \mathbf{v}])
\]

Rule \( \mathbf{dB} \) (resp. \( \mathbf{VS} \)) 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, \( e.g. \quad \lambda x.y(xx)(\mathbf{II}) \rightarrow_{\mathcal{N}} (yxx)[x\backslash \mathbf{II}] \rightarrow_{\mathcal{N}} y(\mathbf{II})(\mathbf{II}) \rightarrow_{\mathcal{N}} y\mathbf{I} \rightarrow_{\mathcal{N}} y\mathbf{I} \rightarrow_{\mathbf{NS}} \mathbf{II} \rightarrow_{\mathbf{VN}} y\mathbf{II} \rightarrow_{\mathbf{V}} (yxx)[x\backslash \mathbf{II}] \rightarrow_{\mathbf{V}} (yxx)[x\backslash \mathbf{I}] \rightarrow_{\mathbf{V}} y\mathbf{I} \).

Notice how reduction \( \rightarrow_{\mathbf{V}} \) 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 \( t \rightarrow_{\mathbf{V}} \delta[y\backslash xx]\delta \rightarrow_{\mathbf{V}} (zz)[z\backslash \delta][y\backslash xx] \rightarrow_{\mathbf{V}} (\delta\delta)[y\backslash xx] \rightarrow_{\mathbf{V}} (\delta\delta)[y\backslash xx] \), 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 \( \rightarrow_{\mathbf{FN}} \) (resp. **full CBV** reduction, written \( \rightarrow_{\mathbf{FV}} \)), given as the closure of the rewriting rules \( \mathbf{dB} \) and \( \mathcal{NS} \) (resp. \( \mathbf{dB} \) and \( \mathcal{VS} \)) under **full contexts**.
We now consider the inhabitation problem for CBV of basis, thus allowing to restrict the search of the Inh

6.1 Call-by-Name Inhabitation

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 \( \Pi \) in system \( \mathcal{N} \) with conclusion \( \Gamma \vdash t : \sigma \) is noted \( \Pi \vdash_{\mathcal{N}} \Gamma \vdash t : \sigma \), and similarly with \( \vdash_{\mathcal{V}} \) 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 \( \Lambda_\lambda \).

1. CBN normalization: \( t \) is \( \mathcal{N} \)-typable if and only if it is \( \xrightarrow{\mathcal{N}} \)-normalizing.
2. CBV normalization: \( t \) is \( \mathcal{V} \)-typable if and only if it is \( \xrightarrow{\mathcal{V}} \)-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 \( \text{Inh}_\mathcal{U} \) 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 \( \text{Inh}_\mathcal{U} \) 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 \( \bot \) is added to the set \( \Lambda_\lambda \) to canonically represent untyped subterms of typed terms. The
resulting set $\Lambda^\mathbb{N}_\perp$ of $\perp$-terms of CBN is given by the inductive definition below:
$$a, b \coloneqq \perp | x | \lambda x. a | ab | a[x/\cdot b]$$

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

$$\begin{align*}
(N) & \quad a \mapsto \text{Var} | \text{App}(a, b) \quad b \mapsto c \mid \perp \quad c \mapsto \text{Lam}(c) \mid a.
\end{align*}$$

Grammar $N$ will play a major role to develop our CBN inhabitation algorithm. Indeed, in a new grammar, so that the same $\text{Inh}_\mathcal{U}$ 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:

$$\begin{align*}
\text{Basis}_\mathbb{N}(\Gamma; \sigma) & \coloneqq \{ b \in \Lambda^\mathbb{N}_\perp \mid b \text{ canonical } \mathbb{N}\text{-solution for } (\Gamma; \sigma) \} \\
\text{Span}_\mathbb{N}(S) & \coloneqq \{ t \in \Lambda_\perp \mid \exists a \in S, \exists u \in \Lambda_\perp, \ a \preceq_\mathbb{N} u \text{ and } t \overset{\mathbb{N}}{\rightarrow} u \}
\end{align*}$$

The solution set of the CBN IP for the typing $(\Gamma; \sigma)$ is $\text{Sol}_\mathbb{N}(\Gamma; \sigma) \coloneqq \{ t \in \Lambda_\perp \mid \exists \Pi \overset{\mathbb{N}}{\rightarrow} \Gamma \vdash t : \sigma \}$. It is generated by the basis as expected:

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

**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 $(_\text{cbn})$; $\Lambda_\perp \rightarrow \Lambda$ is defined as follows:

$$\begin{align*}
x^{\text{cbn}} & \coloneqq x \\
(\lambda x.t)^{\text{cbn}} & \coloneqq \lambda x.t^{\text{cbn}} \\
(tu)^{\text{cbn}} & \coloneqq t^{\text{cbn}}u^{\text{cbn}} \\
(t[x/\cdot u])^{\text{cbn}} & \coloneqq t^{\text{cbn}}[x/\cdot u^{\text{cbn}}].
\end{align*}$$

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

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 $-_\text{cbn}$ translates $\overset{\mathbb{N}}{\rightarrow}$-reduction steps (in CBN) into $\overset{\mathbb{F}}{\rightarrow}$-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 $\perp$-terms by setting in particular $\perp^{\text{cbn}} := \perp$. Note that the embedding is injective, and that its preimage, denoted $(_\text{cbn})^{-1}$, 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 $\text{Basis}_{\mathcal{U}}(\Gamma; \sigma)$ the set $\text{Basis}(\Gamma; \sigma)$ (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 $\rightarrow \lambda!$) Let $\Pi \overset{\mathbb{N}}{\rightarrow} \Gamma \vdash a : \sigma$. Then there exists $\Pi' \overset{\mathcal{U}}{\rightarrow} \Gamma \vdash a^{\text{cbn}} : \sigma$. Moreover, if $\Pi$ is normal then $\Pi'$ is also normal and $\mathcal{A}_{\mathcal{U}}(\Pi') = \mathcal{A}_{\mathbb{N}}(\Pi)^{\text{cbn}}$ when defined.

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

Thus, the translation of an element of $\text{Basis}_{\mathbb{N}}(\Gamma; \sigma)$ is an element of $\text{Basis}_{\mathcal{U}}(\Gamma; \sigma)$. Conversely, any $\perp$-term that translates to an element of $\text{Basis}_{\mathcal{U}}(\Gamma; \sigma)$ is an element of $\text{Basis}_{\mathbb{N}}(\Gamma; \sigma)$.
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 $\text{Inh}_{{\mathcal{U}}}^G$ algorithm to also decide the IP for CBN, where the driving grammar $G$ is no longer $B$, but another $\text{NH}$-grammar. For that, let us focus on the image of the CBN basis by considering the C-grammar $\text{BN}$ below, where $C = \{\perp\}$ and $\text{nno}$ is the start symbol:

$\text{(BN)} \quad \text{nne} \mapsto \text{Var} \mid \text{App}(\text{nne}, \text{nna}) \quad \text{nna} \mapsto \text{Bng}(\text{nno}) \mid \text{Bng}(\perp) \quad \text{nno} \mapsto \text{Lam}(\text{nno}) \mid \text{nne}$

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

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

\[
a \in \text{BN} \iff \exists b \in N, \ a = b^{\text{cbn}} \iff \exists b \in \Lambda_\perp^N, \ a = b^{\text{cbn}} \text{ and } a \in B.
\]

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

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

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

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

<table>
<thead>
<tr>
<th>--- Example 1 ---</th>
<th>--- Example 4 ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Inh}\ N(x:[[[a]]]; \ a)$</td>
<td>$\text{Inh}\ N(\emptyset; [[[\rightarrow]]] \rightarrow [[]])$</td>
</tr>
<tr>
<td>Sol $&gt; \emptyset$</td>
<td>Sol $&gt; \emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>--- Example 2 ---</th>
<th>--- Example 5 ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Inh}\ N(\emptyset; [[[\rightarrow]a] \rightarrow [a] \rightarrow a])$</td>
<td>$\text{Inh}\ N(x:[[\rightarrow]a]; \ a)$</td>
</tr>
<tr>
<td>Sol $&gt; \lambda x.x, \ \lambda x.\lambda y.x y$</td>
<td>Sol $&gt; x \perp$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>--- Example 3 ---</th>
<th>--- Example 6 ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Inh}\ N(\emptyset; [[[\rightarrow]a] \rightarrow [[a] \rightarrow [a]])$</td>
<td>$\text{Inh}\ N(\emptyset; [[[\rightarrow]a] \rightarrow [a])$</td>
</tr>
<tr>
<td>Sol $&gt; \emptyset$</td>
<td>Sol $&gt; \emptyset$</td>
</tr>
</tbody>
</table>

### 6.2 CBN Inhabitation Direct Method

By instantiating the $\text{Inh}_{{\mathcal{U}}}^G$ algorithm with the grammar $\text{BN}$, one can see that our (indirect) algorithm $\text{Inh}_{{\mathcal{U}}}^N$, working on the type system $\mathcal{U}$ and being driven by $\text{BN}$, encodes a (direct) algorithm solving the IP for CBN, working on system $\mathcal{N}$ and grammar $N$. Indeed, grammar $\text{BN}$ is the image of grammar $N$ by the CBN encoding. The direct algorithm for CBN, called $\text{Inh}_N$, 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 $\text{sz}(\_)$ 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 $\lambda!$. 

As expected, the solution set is generated by the basis, as stated below.

\[
\frac{\Gamma = \Gamma_1 + \Gamma_2}{[M \Rightarrow \sigma] \triangleright S(\tau, [\emptyset \Rightarrow \sigma])} \quad \frac{a_1 \triangleright H^x[\tau](\Gamma_1; [M \Rightarrow \sigma])}{a_1a_2 \triangleright H^x[\tau](\Gamma; \sigma)}
\]

\[
\frac{\Gamma \neq \emptyset \quad i \in I}{\Gamma = \iota_{el} \Gamma_i \quad \bigwedge_{i \in I} a_i \triangleright N_i(\Gamma; [\sigma_i])} \quad \frac{\bigwedge_{i \in I} a_i \triangleright N(\Gamma; \sigma)}{[\emptyset \Rightarrow \sigma]} \quad \frac{\bigwedge_{i \in I} a_i \triangleright N_i(\emptyset; [\emptyset])}{\triangleright}
\]

\[
\frac{\Gamma = \Gamma' + x : [\tau]}{a \triangleright H^x[\tau](\Gamma'; \sigma)} \quad \frac{\mathrm{fix} \; x \in \mathrm{dom}(\Gamma) \text{ and } a' \triangleright N(\Gamma, x : M; \sigma)}{\lambda x.a' \triangleright N(\Gamma, M \Rightarrow \sigma)}
\]

Fig. 8. Rules of the Direct Algorithm \(\text{Inh}_N\) for System \(N\)

### 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 \(\text{Inh}^G\) 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 \(V\)-typed locations, normal \(V\)-derivations, and \(V\)-typed reductions are introduced as expected for the type system \(V\) 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 \(V\)-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 \(\perp_V\) and the latter by the constant \(\perp\). The resulting set \(\Lambda^V_{\perp, \perp}\) of \((\perp, \perp)_V\)-terms of CBV is given by the inductive definition below:

\[
a, b := \perp \mid \perp_V \mid x \mid \lambda x.a \mid ab \mid a[x/b]
\]

A preorder \(\preceq_V\) on \(\Lambda^V_{\perp, \perp}\) is given by the \(V\)-contextual closure of \(\perp_V \preceq_V x\) for any variable \(x\) and \(\perp \preceq_V a\) for any \(\perp_V\)-term \(a\). Thus e.g. \(\perp_V (y, \perp) \preceq_V x(yzw)\). Then \(V\)-approximants of normal \(V\)-derivations II, noted \(\mathcal{A}_V(\Pi)\), as well as \textbf{canonical} \(V\)-derivations, and \textbf{canonical} \(V\)-solutions, are defined following the same concepts as in Sect. 3. Canonical \(V\)-solutions can be generated by the following \textbf{C-grammar} \(V\), where \(C = \{\perp, \perp_V\}\) and \(c\) is the start nonterminal symbol:

\[
(V) \quad a \mapsto \text{Var} \mid \text{Sub}(a, b) \mid c \mapsto \text{Lam}(\perp) \mid \text{Lam}(c) \mid \perp_V \mid \text{Var} \mid b \mid \text{Sub}(c, b)
\]

\[
b \mapsto \text{App}(a, c) \mid \text{App}(b, c) \mid \text{Sub}(b, b)
\]

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

The key notions of \textbf{basis} and \textbf{span} for the CBN case are also defined as expected:

\[
\text{Basis}_V(\Gamma; \sigma) := \{b \in \Lambda^V_{\perp, \perp} \mid b \text{ canonical } V\text{-solution for } (\Gamma; \sigma)\}
\]

\[
\text{Span}_V(S) := \{t \in \Lambda^V_{\perp} \mid \exists a \in S, \exists u \in \Lambda^V_{\perp}, a \preceq_V u \text{ and } t \mapsto_{FV} u\}
\]

The solution set of the CBV IP for the typing \((\Gamma; \sigma)\) is \(\text{Sol}_V(\Gamma; \sigma) := \{t \in \Lambda^V_{\perp} \mid \exists \Pi \vdash_{V} \Gamma \vdash t : \sigma\}\). As expected, the solution set is generated by the basis, as stated below.
LEMMA 6.9 (Sound & Complete Basis). For any typing \((\Gamma; \sigma), \text{Span}_{\nu}(\text{Basis}_{\nu}(\Gamma; \sigma)) = \text{Sol}_{\nu}(\Gamma; \sigma)\).

Embedded 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 \((\_)^{\text{cbv}}: \Lambda_{\lambda} \to \Lambda\) is defined as follows:

\[
\begin{align*}
x^{\text{cbv}} & := !x \\
(tu)^{\text{cbv}} & := \begin{cases} 
    L(\langle s \rangle \ u^{\text{cbv}}) & \text{if } t^{\text{cbv}} = L(\langle !s \rangle) \\
    \text{der}(t^{\text{cbv}})u^{\text{cbv}} & \text{otherwise}
\end{cases} \\
(\lambda x.t)^{\text{cbv}} & := !\lambda x.t^{\text{cbv}} \\
(t[x\!\downarrow])^{\text{cbv}} & := t^{\text{cbv}}[x\!\downarrow u^{\text{cbv}}]
\end{align*}
\]

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 \(t^{\text{cbv}} = !\lambda x.x!x \) and \(t^{\text{cbv}} = !\Delta = !((\delta^{\text{cbv}}) \) and \((\delta \delta)^{\text{cbv}} = \Delta !\Delta = (\delta \delta)^{\text{cbv}} \). Moreover, \((xy)z)^{\text{cbv}} = \text{der}(x!y)!z \).

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 \((\_)^{\text{cbv}}\) translates \(\to_{\text{fn}}\)-reduction steps (in CBV) into \(\to_{\!\downarrow}\)-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 (\(\bot, \bot_{\nu}\))-terms, by setting in particular:

\[
\begin{align*}
\bot^{\text{cbv}} & := \bot \\
\bot_{\nu}^{\text{cbv}} & := !\bot \\
(\lambda x.\bot)^{\text{cbv}} & := !\bot
\end{align*}
\]

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 \((\_)^{\text{cbv}}\) 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 \(\bot_{\nu}\) in the CBV side. Next lemma states that typing, normal derivations and approximants are related by the CBV embedding.

LEMMA 6.12 (Bridge).

(1) \((\text{CBV} \to \lambda!)\). Let \(\Pi \not\vdash_{\nu} \Gamma \vdash a : \sigma\). Then there exists \(\Pi' \not\vdash_{\!\downarrow} \Gamma \vdash a^{\text{cbv}} : \sigma\). Moreover, if \(\Pi\) is normal then \(\Pi'\) is also normal and \(\text{A}_{\!\downarrow}(\Pi') = \text{A}_{\nu}(\Pi)^{\text{cbv}}\) when defined;

(2) \((\lambda! \to \text{CBV})\). Let \(\Pi \not\vdash_{\nu} \Gamma \vdash a : \sigma\). Then for any \(b \in a^{\text{cbv}}\), there exists \(\Pi' \not\vdash_{\nu} \Gamma \vdash b : \sigma\). Moreover, if \(\Pi\) is normal then \(\Pi'\) is also normal and \(\text{A}_{\!\downarrow}(\Pi') = \text{A}_{\nu}(\Pi)^{\text{cbv}}\) with \(\text{A}_{\nu}(\Pi)^{\text{cbv}} = b\) if \(a = \text{A}_{\!\downarrow}(\Pi)\) when defined.

As with CBN, the translation of an element of \(\text{Basis}_{\nu}(\Gamma; \sigma)\) is also an element of \(\text{Basis}_{\!\downarrow}(\Gamma; \sigma)\). Conversely, any \(\bot\)-term in the preimage of an element of \(\text{Basis}_{\!\downarrow}(\Gamma; \sigma)\) is an element of \(\text{Basis}_{\nu}(\Gamma; \sigma)\).

CBV inhabitation. Again, we use the restriction of the \(\text{Inh}_{\nu}^{C}\) algorithm on a new \(\text{NH}\)-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 \(\text{vno}\) is the start symbol:

\[
\begin{align*}
\text{vne}_{e} & \mapsto \text{Var} \\
\text{vne}_{f} & \mapsto \text{Var} \mid \text{Sub}(\text{vne}_{e}, \text{vne}_{a}) \\
\text{vne}_{d} & \mapsto \text{Der}(\text{vne}_{a}) \\
\text{vne}_{a} & \mapsto \text{App}(\text{vne}_{e}, \text{vno}) \mid \text{App}(\text{vne}_{d}, \text{vno}) \mid \text{Sub}(\text{vne}_{a}, \text{vne}_{a}) \\
\text{vnb} & \mapsto \text{Lam}(\text{vno}) \mid \text{vne}_{v} \\
vno & \mapsto \text{Bng}(\text{vnb}) \mid \text{Bng}(\bot) \mid \text{vne}_{a} \mid \text{Sub}(\text{vno}, \text{vne}_{a})
\end{align*}
\]

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 \( a \in \Lambda \)
\[
   a \in \text{BV} \iff \exists b \in \mathcal{V}, \ a = b^{cbv} \iff \exists b \in \Lambda^{W}, \ a = b^{cbv} \text{ and } a \in B.
\]

Using Cor. 5.14 and Thm. 5.15, we deduce that \( \text{Inh}^{BV}_U \) 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 \((\Gamma; \sigma)\), \(\text{Span}_\mathcal{V}(\text{Inh}^{BV}_U(\Gamma; \sigma)^{cbv}) = \text{Sol}_\mathcal{V}(\Gamma; \sigma)\).

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

Example 6.15. Let us see some examples of the IP in CBV, i.e., some runs of \( \text{Inh}^G_U \) using the grammar \( G = \text{BV} \), 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:([[\alpha]])); \( a \)  
Sol> \( \emptyset \)

--- Example 2 ---
Inh N(\( \emptyset \); \([[\alpha]\rightarrow [\alpha]] \rightarrow [\alpha]\rightarrow [\alpha]\))  
Sol> \( \emptyset \)

--- Example 3 ---
Inh N(\( \emptyset \); \([[\alpha]\rightarrow [\alpha]] \rightarrow [\alpha]\rightarrow [\alpha]])  
Sol> \( \lambda x . x , \lambda y . \lambda y . \lambda y . x (x := y z) \)

--- Example 4 ---
Inh N(\( \emptyset \); \( [\emptyset] \rightarrow [\emptyset] \rightarrow [\emptyset] \))  
Sol> \( \emptyset \)

--- Example 5 ---
Inh N(\( x : [\emptyset] \rightarrow [\alpha]; \ a \))  
Sol> \( x \rightarrow \lambda \ldots \)

--- Example 6 ---
Inh N(\( \emptyset \); \( [\alpha] \rightarrow [\alpha] \))  
Sol> \( \lambda x . x \)

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 \( \text{Inh}_V \) for CBN in Sect. 6.2. Indeed, the (indirect) algorithm \( \text{Inh}^{BV}_U \), working on system \( U \) and grammar BV, was obtained by instantiating the general \( \text{Inh}^G_U \) algorithm with grammar BV. Grammar BV is the image of grammar V by the CBV encoding so that \( \text{Inh}^{BV}_U \) encodes some (direct) algorithm solving the IP for CBV and working on system \( V \) and grammar V. This direct algorithm, called \( \text{Inh}_V \), can also be obtained by the "preimage extraction", and does not pass through \( \lambda ! \). The rules of \( \text{Inh}_V \) (Fig. 9) appear quite similar to the ones of \( \text{Inh}^{BV}_U \) (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 \( U \). 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...
Therefore, soundness and completeness of their inhabitation algorithm are not guaranteed.

\[ \begin{align*}
\Gamma = \Gamma' + x : [\tau] & \quad \frac{a \vdash H^x_\varnothing(\Gamma'; \sigma)}{a_1 a_2 \vdash H^x_\varnothing(\Gamma'; \sigma)} \\
\Gamma = \Gamma' + x : [\tau] & \quad \frac{x \vdash N(\sigma; \Gamma')}{x \vdash N(\Gamma'; \sigma)} \\
\Gamma = \Gamma' + x : [\tau] & \quad \frac{a \vdash H^x_\varnothing(\Gamma'; \sigma)}{a \vdash H^x_\varnothing(\Gamma'; \sigma)} \\
\Gamma = \Gamma' + x : [\tau] & \quad \frac{\Gamma = \Gamma + y : [\rho], \text{fix } y \notin \text{dom}(\Gamma) \cup \{x\} \quad n \in [0, sz(\rho)], \quad M \vdash S(\rho_1, \varnothing_1, \ldots, \varnothing_n)}{a \vdash H^N_\varnothing(\Gamma_0, \sigma; \rho_1) \quad b \vdash H^N_\varnothing(\Gamma_0; \sigma) \\
\Gamma = \Gamma + y : [\rho_1] & \quad \frac{a \vdash H^N_\varnothing(\Gamma_0, y : [\rho_1]; \sigma)}{a \vdash H^N_\varnothing(\Gamma_0; \sigma) \\
\Gamma = \Gamma + y : [\rho_1] & \quad \frac{\Gamma = \Gamma + y : [\rho_1] \quad n \in [1, sz(\rho_1)], \quad \rho_1 \in [1,n]}{a \vdash H^N_\varnothing(\Gamma_0, y : [\rho_1]; \sigma)} \quad b \vdash H^N_\varnothing(\Gamma_0; \sigma) \\
\Gamma = \Gamma + y : [\rho_1] & \quad \frac{\Gamma = \Gamma + y : [\rho_1] \quad n \in [1, sz(\rho_1)], \quad \rho_1 \in [1,n]}{a \vdash H^N_\varnothing(\Gamma_0, y : [\rho_1]; \sigma)} \quad b \vdash H^N_\varnothing(\Gamma_0; \sigma)
\end{align*} \]

Fig. 9. Rules of the Direct Algorithm \( \text{Inh}_V \) for System \( \forall \)

\((Q \in \{F, A\} \text{ in rules (APP-Q), (ES-H), (ES-CH)})\)

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 CBV and CBV \( \lambda! \)-calculi in a quantitative framework. This is done by using (untyped and typed) appropriate embeddings of CBV/CBV into the \( \lambda! \)-calculus. In the first case, we use the CBV embedding by Girard, and the resulting CBV 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, 2022,a,b], and their calculus without permutations contains normal forms that are untappable. Therefore, soundness and completeness of their inhabitation algorithm are not guaranteed.

It is worth noticing that, although our method encodes the inhabitation problems of CBV/CBV into the one for the \( \lambda! \)-calculus, direct algorithms (Sect. 6.2 for CBV, 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 CBV 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 $t[x\downarrow s]$ can be seen as syntactic sugar for a $\beta$-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 $\beta$-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 $\beta$-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 $\lambda$-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 $\lambda$-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


References


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-00367725


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