An Implementation of the Quantitative Inhabitation Algorithm for Different Lambda Calculi in a Unifying Framework

Victor Arrial*

Contents

1 Introduction

This in an Ocaml[1] implementation of the inhabitation algorithm presented in the article "Quantitative Inhabitation for Different Lambda Calculi in a Unifying Framework".

*IRIF - Université Paris Cité

[1]This project has been developed with `Dune 3.0.3` and run on `Ocaml 4.14.0`.

Goals. This project is not and was never envisonned as a highly optimized implementation of the algorithm but rather to solve the following goals: (1) *Compute actual inhabitation solutions*: The algorithm is fairly complex and hand-crancking simple examples is in fact a very challenging exercice. Having an implementation allows us to compute inhabitation solutions for all Call-by-Push-Value, Call-by-Name and Call-by-Value λ-calculus. This also allows us to check the examples presented in the article. Moreover, the general approach is preserved in this implementation which allows us to conveniently picture the addition of other models of computation. (2) *Prototyping futur potential applications*: The algorithm can be seen as a particular tool for type-based program synthesis and we could envision its use in software testing frameworks as QuickChick. This implementation provides first elements to tinker with and have better understanding of our work. (3) *Research tool*: Multiple techniques have been developped to reduce the overall complexity of proof search in logic programming and more generally of program synthesis. This implementation allows us to concretely visualize the strong and weak points of our approach and being able to strengthen it in the futur.

Road-Map. Section 1 is dedicated to a quick presentation of the inhabitation problem. We briefly recall the definitions of the $\lambda!$-calculus and its associated typing system $\mathcal{U}$. Section 2 is a general presentation of the inhabitation algorithm. We shortly discuss the algorithm search, introduce the subtype search algorithm and present the general form of algorithm. Section 3 explains how to use our implementation and recreate the paper examples. Finally, section 4 is a (partial) technical documentation.

1.1 The $\lambda!$-Calculus

Let us briefly introduce the term syntax of the $\lambda!$-calculus. Given a countably infinite set $\mathcal{X}$ of variables $x, y, z, \ldots$, the set of terms $\mathcal{T}$ is given by the following inductive definition:

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

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

1.2 The Multityping System $\mathcal{U}$

We now quickly introduce the quantitative typing system. It contains functional and intersection types. Intersection is considered to be associative, commutative but not idempotent, thus an intersection type is represented by a (possibly empty) finite multiset $[\![\sigma_i]\!]_{i\in I}$. Formally, given a countably infinite set $\mathcal{TV}$ of type variables $\alpha, \beta, \gamma, \ldots$, we inductively define:

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

Type environments, noted Γ, Δ, are functions from variables to multitypes, assigning the empty multitype $[\![\,]\!]$ to all the variables except a finite number (possibly zero). The empty

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

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

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

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

A typing judgment is a triple of the form $\Gamma \vdash t : \sigma$, where Γ is a typing environment, t is a term (called the subject of the typing judgment), and σ is a type. The typing system $\mathcal{U}$ for the $\lambda!$-calculus is defined by the rules in Fig. 1.

1.3 The Inhabitation Problem

Given a calculus and a type system, three problems naturally arises:

- *type checking*: given an environment Γ, a term t and a type σ, decide whether $\Pi \triangleright \Gamma \vdash t : \sigma$;
- *typability*: given a term t, decide whether there exists an environment Γ and a type σ such that $\Pi \triangleright \Gamma \vdash t : \sigma$;
- *inhabitation*: given an environment Γ, and a type σ, decide whether there is a term such that $\triangleright\Gamma \vdash t : \sigma$.

Inhabitation corresponds to decide the existence of a program (term t) that satisfies the given specification (type σ) under some assumptions (environment Γ). Decidability of the inhabitation problem can be seen as a particular tool for type-based *program synthesis*, whose task is to construct —from scratch—a program that satisfies some high-level formal specification.

In our work, a generalization of the inhabitation problem is considered and defined as follows:

Definition. *The generalized inhabitation problem is defined as follows:*
Input: *A type context* Γ *and a type* σ*.*
Output: *The set of all solutions to the IP problem* $(\Gamma; \sigma)$*.*

The paper studies the generalized inhabitation problem for the $\Lambda!$-calculus equipped with the typing system $\mathcal{U}$. In particular, we denote by $\mathtt{Sol}(\Gamma; \sigma) := \{t \in \mathcal{T} \mid \exists\, \Pi \triangleright \Gamma \vdash t : \sigma\}$ the set of solutions for a given instance $(\Gamma; \sigma)$.

2 The Inhabitation Algorithm

The paper presents an algorithm solving the inhabitation problem for the $\lambda!$-calculus (Sect. 1.1) equipped with the multityping system $\mathcal{U}$ (Sect. 1.2). This is achieved through a typing derivation search. In this section, we explain the different mechanisms of the algorithm. We first focus on defining the space to explore (Sect. 2.1) and secondly detail a subtyping algorithm (**??**). We finally present the main algorithm (Sect. 2.2) and discuss the overall shape of the implementation (Sect. 2.3).

2.1 Canonical Type Derivations

Solving the generalized inhabitation problem requires being able to compute all solutions of any given instance of the inhabitation problem. Exhaustively exploring all type derivations $\Pi \triangleright \Gamma \vdash t : \sigma$ in order to produce all solutions is not possible since there might exists infinitely many of both. Instead, the algorithm limits its search to a specific kind of type derivations: canonical derivations.

Canonicity is developped in an extended calculus denoted $\Lambda_\perp$ obtained from the $\Lambda!$-calculus by addition of a constant $\perp$. This constant can be introduced anywhere in terms. In typed terms, the bang rule (`bag`) allows to introduce them in untyped subterm. Canonical derivation are then defined as follows:

Definition. *A type derivation is said canonical if it is normal and types its approximant.*

A derivation is normal if every typed subterm is in normal form. A normal derivation types its approximant if every untyped subterm is $\perp$. More details and example can be found in the paper. We focus on solutions obtained by canonical derivations.

Definition. *A basis for the typing* $(\Gamma; \sigma)$ *is the set:*

$$\texttt{Basis}(\Gamma; \sigma) := \{b \in \Lambda_\perp \mid b \textit{ has a canonical derivation for } (\Gamma; \sigma)\}$$

Using a very simple notion of span (redex expansions and constant replacements) we show that limiting the search to canonical derivations and therefore using the basis as finite descriptions of the set of solution is sound and complete.

Theorem. *For every typing* $(\Gamma; \sigma)$, $\texttt{Span}(\texttt{Basis}(\Gamma; \sigma)) = \texttt{Sol}(\Gamma; \sigma)$

The goal of the algorithm is to build such basis. Note that one can easily characterize the union of all basis through the following grammar:

$$\begin{array}{lcl} \texttt{cne} & := & \texttt{Var} \mid \texttt{App}(\texttt{cne}, \texttt{cna}) \mid \texttt{Der}(\texttt{cne}) \mid \texttt{Sub}(\texttt{cne}, \texttt{cne}) \\ \texttt{cnb} & := & \texttt{cne} \mid \texttt{Lam}(\texttt{cno}) \mid \texttt{Sub}(\texttt{cnb}, \texttt{cne}) \\ \texttt{cna} & := & \texttt{cno} \mid \texttt{Bng}(\texttt{cno}) \mid \texttt{Bng}(\perp) \mid \texttt{Sub}(\texttt{cna}, \texttt{cne}) \\ \texttt{cno} & := & \texttt{cna} \mid \texttt{cnb} \end{array}$$

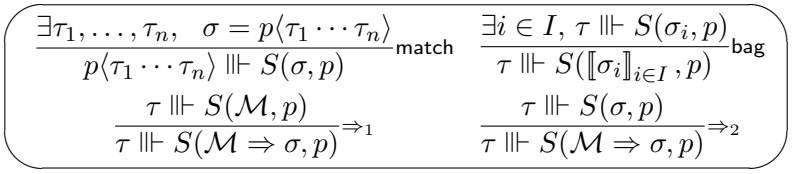

Figure 2: Rules of the subtype search Algorithm.

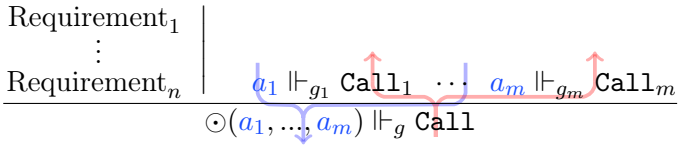

Figure 3: Pattern of the Inhabitation Algorithm Rules

2.2 Main Algorithm

Tree Structure. Type derivations are made of typing rules assembled into a tree structure, so the inhabitation algorithm is written in a similar spirit: each rule of the typing system $\mathcal{U}$ has at least a corresponding rule in **Inh**, declined in variants as we will explain. The algorithm builds a tree having the rules of **Inh** for (hyper) edges and sequents of the form $a \Vdash_g \texttt{Call}$ for nodes, where `Call` is any kind of **call** (different kinds of calls are detailed later), a is an *answer* (a canonical term) to the call, and g is a nonterminal symbol of a grammar, considered to be the **parameter** of `Call`. Such a tree is built by the algorithm search in two subsequent stages:

- *bottom-up* on the right-hand sides (↑ in Fig. 3): from a node of the form $\_ \Vdash_g \texttt{Call}$, corresponding to the conclusion of some rule of the algorithm (meaning that **Inh** is searching for an answer for the call `Call`), the algorithm performs *recursive calls* $\_ \Vdash_{g_1} \texttt{Call}_1, \ldots, \_ \Vdash_{g_n} \texttt{Call}_n$ —the premises of the corresponding algorithm rule;
- *top-down* on the left-hand sides (↓ 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 `Call`, written $\odot(a_1, \ldots, a_m) \Vdash_g \texttt{Call}$, is built from the premises answers $a_1 \Vdash_{g_1} \texttt{Call}_1, \ldots, a_m \Vdash_{g_m} \texttt{Call}_n$.

Moreover, at any node, the set of applicable rules is regulated by some preliminary requirements. So, the general form of the rules of the algorithm **Inh** 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 **Inh** 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 (↑ in Fig. 3). For example, the rule

(N-H) in Fig. 4 has three requirements, the first one demands the existence of *some* rule in the grammar of the form $g \rightsquigarrow g'$, the second demands the existence of *some* splitting of the environment $\Gamma = \Gamma' + x : [\![\tau]\!]$, and the last one requires a subtype verification $\sigma \Vdash S(\tau, \diamond)$ w.r.t. the type τ found in the second 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 algorithm (Fig. 2) used by **Inh** 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 **Inh** has two different kinds of **calls**, generically denoted `Call`: they are either N*-calls* of the form $N(\Gamma; \sigma)$, or H*-calls* of the form $H^{x:[\![\tau]\!]}(\Gamma; \sigma)$. The former (resp. latter) correspond to the search for a solution to the IP with input (Γ, σ) (resp. $(\Gamma + x : [\tau], \sigma)$). In H-calls, a specific variable x and type $[\tau]$ is picked out of the type context to act as a search hint.

Run. Given an input typing $(\Gamma; \sigma)$, a **run** of **Inh** is a tree of recursive calls starting from the root $N(\Gamma; \sigma)$ with the start nonterminal symbol `cno` of the associated grammar `B`; 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 ($\mathsf{BG}_\perp$), the only ones with no premises,
- in all the instances of the rule (BG), the least upper bound of all recursive answers ($\uparrow_{i \in I} a_i$) exists (this is necessarily checked at the top-down stage, *after* all recursive calls).

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

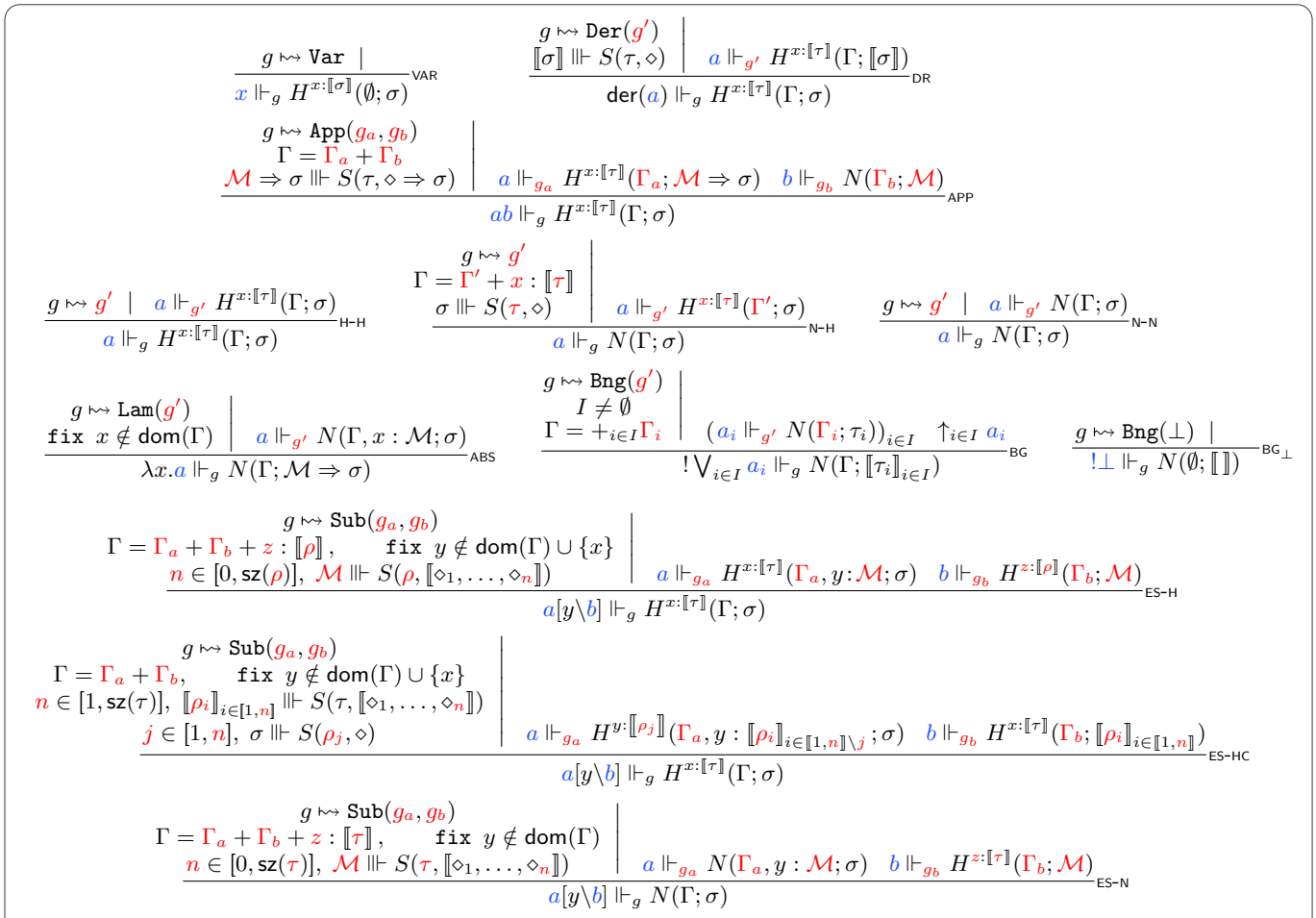

Figure 4: Rules of the $\mathbf{Inh}_{\mathcal{U}}$ Algorithm for System $\mathcal{U}$

2.3 Implementation

The file `CBPVBotTerms.ml` handles the terms of the calculus. The file `Grammar.ml` handles grammars. The file `Multitype.ml` handles the intersection types. The file `SubtypeSearch.ml` handles the subtype search. The file `TypeContext.ml` handles the type contexts. The file `Inhabitation.ml` contains the search algorithm solving the inhabitation problem.

Each rule presented in Fig. 4 of the algorithm has a corresponding function in our implementation. For exeample, the rule (`app`) is implemented by the function `rule_app`.

The search is guided by the grammar. In the implementation, the first step consists in computing the set of productions accessible from the current non-terminal. Then all corresponding rules are applied. Themselves are able to recursively call the inhabitation algorithm. Dependencies are depicted in Fig. 5

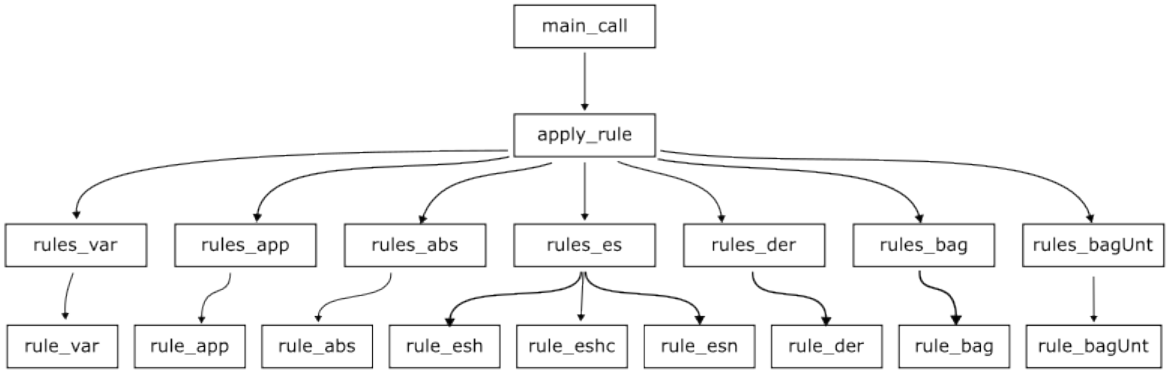

Figure 5: Call Dependencies in the Search Function

3 Examples

This project has been developed with `Dune 3.0.3` and run on `Ocaml 4.14.0`. Ocaml can be installed through Opam which can be found at the following link :

`https://opam.ocaml.org/doc/Install.html`

Dune can then be installed using Opam.

Once both Ocaml and Dune have been setup, a simple `make` in the directory allows to reproduce the examples of the article.

3.1 Article Examples

Three different grammars are defined. A first grammar stored in the variable `canCBPV` is used to guide the search in order to solve the inhabitation problem for the $\lambda!$-calculus.

Two more restrictive grammars (stored in `canCBN` and `canCBV`) are used to solve the inhabitation problem in Call-by-Name and Call-by-Value respectively.

The six examples are the following one:

$$\begin{array}{lcc}
(& x : [\![[\![[\alpha]\!]]\!] & ; & \alpha &), \\
(& \emptyset & ; & [\![\alpha \Rightarrow \alpha]\!] \Rightarrow [\![\alpha]\!] \Rightarrow \alpha &), \\
(& \emptyset & ; & [\![[\![[\alpha]\!] \Rightarrow [\![\alpha]\!]]\!] \Rightarrow [\![[\![\alpha]\!] \Rightarrow [\![\alpha]\!]]\!]]\!] &), \\
(& \emptyset & ; & ([\![]\!] \Rightarrow [\![]\!]) \Rightarrow [\![]\!] &), \\
(& x : [\![[\![]\!] \Rightarrow \alpha]\!] & ; & \alpha &), \\
(& \emptyset & ; & \alpha &)
\end{array}$$

All these examples are solved for the $\lambda!$-calculus, Call-by-Name and Call-by-Value.

3.2 Run Tree (Verbose)

The inhabitation function possesses a verbose argument. It has three possibles values: 0: Non-verbose mode; 1: All and only successful calls are printed; 2: All calls are printed.

Non-deterministic choices are stacked on top of each over with an additional horizontal separator. Multiple premisses are indicated using alternating white and gray squares. Successful and unsuccessful calls are indicated using checkmarks.

4 Technical Documentation

4.1 Grammars (`Grammar.ml`)

The OCaml file `Grammar.ml` contains an implementation of the notion of C-grammar defined in Section 3 (Definition 3.6) of our paper. Let us recall the definition here:

Definition. *A C-grammar $G = (\Sigma_C, N, R, s)$ is a first-order tree grammar whose set of ranked alphabet Σ_C is given by the zero-ary $C \cup \{\texttt{Var}\}$, the unary symbols $\texttt{Der}(\cdot)$, $\texttt{Bng}(\cdot)$ and $\texttt{Lam}(\cdot)$, and the binary symbols $\texttt{App}(\cdot,\cdot)$ and $\texttt{Sub}(\cdot,\cdot)$.*

Constants (C) and non-terminal symbols (N) are implemented using strings.

```
module GrammarCst       = String
module GrammarCstSet    = Set.Make(GrammarCst)
```

```
module GrammarNTerm     = String
module GrammarNTermSet  = Set.Make(GrammarNTerm)
```

The notion of production rule using a map from non-terminal symbols to lists of (later defined) algebraic terms $T_{\Sigma_C}(N)$.

```
module GrammarProdRules = Map.Make(GrammarNTerm)
```

Algebraic terms are terms formed from the alphabet of the C-grammar. In our implementation, each symbol has a corresponding constructor with the expected arity. In particular, the constructor `NTERM` of arity 1 introduces non-terminal symbols, and similarly with the constructor `CST` for constant symbols.

```
type grammarTerm =
    | VAR
    | NTERM of GrammarNTerm.t
    | CST   of GrammarCst.t
    | APP   of grammarTerm * grammarTerm
    | ABS   of grammarTerm
    | ES    of grammarTerm * grammarTerm
    | DER   of grammarTerm
    | BAG   of grammarTerm
```

Two functions defined on `grammarTerm` are introduced to compute the set of non-terminal symbols and constants appearing in them.

```
let rec computeNonTerm = ...
    : grammarTerm -> GrammarNTermSet.t
```

```
let rec computeCst = ...
    : grammarTerm -> GrammarCstSet.t
```

These functions are used internally to ensure that C-grammars are well-formed.

The set of production rules is implemented using a map from non-terminal symbols to a list of algebraic terms. For example, the following set of production rules:

$$n_1 \rightsquigarrow \texttt{Der}(n_2) \qquad n_1 \rightsquigarrow \texttt{App}(n_1, n_2) \qquad n_2 \rightsquigarrow \bot$$

is represented by a map with the following (key, value) couples:

```
        (NTERM "n1",  [DER(NTERM "n2"); APP(NTERM "n1", NTERM "n2")])
    and (NTERM "n2",  [CST "⊥"])
```

The notion of C-grammar is implemented as a 4-uplet composed of a set of constant symbols, a set of non-terminal symbols, a map from non-terminals to algebraic terms and a non-terminal start symbol.

```
type grammar = Grammar of
    GrammarCstSet.t
    * GrammarNTermSet.t
    * grammarTerm list GrammarProdRules.t
    * GrammarNTerm.t
```

All grammar projections are defined as expected.

```
let getGramCst = ...      : grammar -> GrammarCstSet.t
let getGramNonTerm = ... : grammar -> GrammarNTermSet.t
let getGramProd = ...     : grammar -> grammarTerm list GrammarProdRules.t
let getGramStart = ...    : grammar -> GrammarNTerm.t
```

Another function allows to seek for all production rules with a given non-terminal symbol at their left.

```
let getProdTerms gram nTerm = ...
    : grammar -> GrammarNTerm.t -> grammarTerm list
```

Given a grammar `gram` and a non-terminal symbol `nTerm`, the function returns a list $[\mathtt{t_1}; ...; \mathtt{t_n}]$ of algebraic terms such that for each t_i there exists a production of the form $\mathtt{nTerm} \rightsquigarrow \mathtt{t_i}$ in the given grammar `gram`.

Example. *Consider* $G = (C, \{a, b\}, P, s)$ *where* $P = \{a \rightsquigarrow \mathtt{Der}(b), b \rightsquigarrow \mathtt{App}(a, b), b \rightsquigarrow \mathtt{Sub}(a, a)\}$.

1. *Then* `getProdTerms G "a"` *returns* `[DER "b"]`.
2. *Then* `getProdTerms G "b"` *returns* `[APP("a", "b"); ES("a", "a")]`.

In a C-grammar $G = (C, N, P, s)$, every symbol occuring in a production rule of P is expected to either be a constant from C, a non-terminal symbol from N or one of the symbols `Var`, `App`, `Lam`, `Sub`, `Der` and `Bng`. To ensure such property, C-grammar are manipulated through the following functions.

```
let emptyGram start = ...
    : GrammarNTerm.t -> grammar
```

Given a non-terminal symbol `start`, this function returns a C-grammar $G = (\emptyset, \{\mathtt{start}\}, \emptyset, \mathtt{start})$ which has no constant symbol, no production rule and `start` as unique non-terminal symbol and start symbol.

```
let addCst gram cst = ...
    : grammar -> GrammarCst.t -> grammar
```

Given a grammar `gram` and a constant symbol `cst`, this function returns the given grammar whose constant symbol set has been extended to include `cst`. If `cst` was already a constant of `gram` then the grammar is left untouched.

```
let addNonTerm gram nTerm = ...
    : grammar -> GrammarNTerm.t -> grammar
```

Given a grammar `gram` and a non-terminal symbol `nTerm`, this function returns the given grammar whose non-terminal symbol set have been extended to include the non-terminal symbol `nTerm`. If `nTerm` was already a non-terminal symbol of `gram`, then all productions rules attached to it are removed.

Example. *Consider* $G = (C, \{a, b\}, P, s)$ *where* $P = \{a \rightsquigarrow \mathtt{Der}(b), b \rightsquigarrow \mathtt{App}(a, b), b \rightsquigarrow \mathtt{Sub}(a, a)\}$.

1. *Then* `addNonTerm G "c"` *returns* $H = (C, \{a, b, c\}, P, s)$.
2. *Then* `addNonTerm G "b"` *returns* $H = (C, \{a, b\}, \{a \rightsquigarrow \mathtt{Der}(a)\}, s)$.

```
let appendGramProd gram nTerm term = ...
    : grammar -> GrammarNTerm.t -> grammarTerm -> grammar
```

Given a grammar `gram`, a non-terminal symbol `nTerm` and a grammar term `term`, the function returns the given grammar whose production rule set have been extended to include the production rule $\mathtt{nTerm} \rightsquigarrow \mathtt{term}$. If `nTerm` or if any non-terminal symbol of `term` is not a non-terminal symbol of `gram` then the grammar is left untouched. And similarly if any constant symbol of `term` is not a constant symbol of `gram`.

Example.

1. *Consider* $G = (C, \{a, b\}, P, s)$ *where* $P = \{a \mapsto \texttt{Der}(a)\}$. *Then* `appendGramProd G "a" (APP (NTERM "b") (NTERM "a"))` *returns* $H = (C, \{a, b\}, P', s)$ *where* $P' = \{a \mapsto \texttt{Der}(a), a \mapsto \texttt{App}(b, a)\}$
2. *Consider* $G = (C, \{a\}, \{a \mapsto \texttt{Der}(b)\}, s)$. *Then* `appendGramProd G "a" (APP (NTERM "a") (NTERM "b")` *returns* G.

```
let removeGram gram nTerm = ...
    : grammar -> GrammarNTerm.t -> grammar
```

Given a grammar `gram` and a non-terminal symbol `nTerm`, the function returns the given grammar whose non-terminal `nTerm` has been removed. If `nTerm` was not a non-terminal symbol of `gram` then the grammar is left untouched. Moreover, if `nTerm` is the start symbol of `gram` or if it is contained in any production rule associated to another non-terminal symbol, then the removal is considered unsafe and the grammar is left untouched.

Example. *Consider* $G = (C, \{s, a, b\}, \{b \mapsto \texttt{Der}(a)\}, s)$.

1. *Then* `removeGram G "b"` *returns* $H = (C, \{s, a\}, \emptyset, s)$.
2. *Then* `removeGram G "s"` *and* `removeGram G "a"` *returns* G.

Finally, we consider two functions translating respectively `grammarTerm` and `grammar` into `Strings` in order to be printed.

```
let rec grammarTerm_to_str = ...
    : grammarTerm -> String
```

```
let grammar_to_str gram = ...
    : grammar -> String
```

4.2 Type System (`Multitype.ml` and `TypeContext.ml`)

This subsection is dedicated to the implementation of intersection types. Everything type related is contained in the two files `Multitype.ml` and `TypeContext.ml`.

4.2.1 Multitypes

The OCaml file `Multitype.ml` contains an implementation of the notion of multitypes. Types variables are represented using integers and intersection types as lists of types. Recall that intersections are considered modulo associativity and commutativity. We therefore define an order on their implementation so that the lists used intersections are always sorted in order to manipulate canonical representants.

The subtype search algorithm uses partial knowledge which are types with holes. For pratical reasons, this implementation does not make a distinction between both. Hole as simply carefully placed when needed in the subtype search. However, the two notions could in reality be easily separated.

We end up with the following definition:

```
type typ =
        | TVar of int
        | TArr of typ * typ
        | TMlt of typ list
        | THole
```

```
let typComp t1 t2 = ...
    : typ -> typ -> int
```

Given two types `t1` and `t2`, this function returns -1 if `t1` is smaller than `t2`, 0 in case of equality and 1 if `t1` is greater than `t2`.

```
let size t = ...
    : typ -> int
```

Given a type `t`, this function return the size of the type.

Example. *Consider* $t = [\![\tau_1, \tau_2]\!] \Rightarrow \sigma$ *(encoded as* `TArr (TMult [(TVar 1); (TVar 2)]) (TVar 3)`*). Then* `size t` *returns* `6`*.*

In order to always manipulate canonical representant, we require all type list appearing in the constructor `TMult` to be sorted. The following fonctions are introduce to introduce types in a safe way. All constructors have a corresponding function.

```
let tHole = ...
    : typ
```

This function returns a type hole.

```
let tVar i = ...
    : int -> typ
```

Given an integer `i`, this function returns a type variable with identifier `i`.

```
let tEmpty = ...
    : typ
```

This function returns an empty intersection type.

```
let tSingleton t = ...
    : typ -> typ
```

Given a type `t`, this function returns and singleton of the given type `t`.

```
let tMultiset ls = ...
    : typ list -> typ
```

Given a list of types `ls`, this function returns the intersection type whose types are the elements of the given list `ls`.

Example. *Consider the types* $\mathtt{t}_1 = \alpha \Rightarrow \beta$, $\mathtt{t}_2 = \square$ *and* $\mathtt{t}_3 = \beta$*, each represented by:*

$\mathtt{t}_1$ = `TArr(TVar 1, TVar 2)`
$\mathtt{t}_2$ = `THole`
$\mathtt{t}_3$ = `TVar 2`

Then `tMultiset` $[\mathtt{t}_1;\mathtt{t}_2;\mathtt{t}_3]$ *returns the canonical representant of* $[\![\alpha \Rightarrow \beta, \square, \beta]\!]$ *being:*
`TMult ([THole; TVar 2; TArr(TVar 1, TVar 2)])`

```
let add_opt mTyp t = ...
    : typ -> typ -> typ option
```

Given a multitype `mTyp` and a type `t`, this function returns an option value of an intersection type containing the same types as 'M' and the additional type `t`. If the type `mTyp` is not an intersection, then the function returns None.

Example. *Consider the types* $\mathtt{t}_1 = [\alpha \Rightarrow \beta, \square]$ *and* $\mathtt{t}_2 = \beta$*, each represented by:*
$\mathtt{t}_1$ = `TMult ([THole; TArr(TVar 1, TVar 2)])`
$\mathtt{t}_2$ = `TVar 2`

1. *Then* `add_opt` $\mathtt{t}_1$ $\mathtt{t}_2$ *returns the optional value* `Some` *containing the canonical representant of* $[\![\alpha \Rightarrow \beta, \square, \beta]\!]$ *being:*
   `TMult ([THole; TVar 2; TArr(TVar 1, TVar 2)])`
2. *Then* `add_opt` $\mathtt{t}_2$ $\mathtt{t}_2$ *returns* `None` *since one cannot add a type to something that is not a intersection.*

```
let ( ++ ) mTyp t = ...
    : typ -> typ -> typ
```

This notation is the unsafe variant of `add_opt`.

```
let merge_opt mTyp1 mTyp2 = ...
    : typ -> typ -> typ option
```

Given two intersections `mTyp1` and `mTyp2`, this function returns an option value of an intersection type containing the types of `mTyp1` and `mTyp2`. If either `mTyp1` of `mTyp2` are not intersections then the function returns None.

Example. *Consider the types* $\mathtt{t}_1 = [\![\beta, \square]\!]$, $\mathtt{t}_2 = [\![\alpha \Rightarrow \beta]\!]$ *and* $\mathtt{t}_3 = \gamma$*, each represented by:*
$\mathtt{t}_1$ = `TMult ([THole; TVar 2])`
$\mathtt{t}_2$ = `TMult ([TArr(TVar 1, TVar 2)])`
$\mathtt{t}_3$ = `TVar 3`

1. *Then* `merge_opt` $\mathtt{t}_1$ $\mathtt{t}_2$ *returns the option value* `Some` *containing the union* $\mathtt{t}_1 \uplus \mathtt{t}_2$ *canonically represented by:*
   `TMult ([THole; TVar 2; TArr(TVar 1, TVar 2)])`
2. *Then* `merge_opt` $\mathtt{t}_1$ $\mathtt{t}_3$ *and* `merge_opt` $\mathtt{t}_3$ $\mathtt{t}_1$ *returns* `None` *since we can only merge two intersections.*

```
let ( |+| ) mTyp1 mTyp2 = ...
    : typ -> typ -> typ
```

This notation is the unsafe variant of `merge_opt`.

```
let pickDirectSubtype t = ...
    : typ -> (typ * typ) list
```

Given a type `t`, this function returns a list of all pairs of a direct subtypes of `t` and a multitype of the remaining others.

Example. *Consider the types* $\texttt{t}_1 = [\![\beta, [\![\alpha \Rightarrow \beta]\!], \gamma]\!]$, $\texttt{t}_2 = \gamma$ *and* $\texttt{t}_3 = [\![\alpha, \alpha]\!] \Rightarrow \gamma$.

1. *Then* `pickDirectSubtype` $\texttt{t}_1$ *returns* $[(\beta, [\![[\![\alpha \Rightarrow \beta]\!], \gamma]\!]);\ ([\![\alpha \Rightarrow \beta]\!], [\![\beta, \gamma]\!]);\ (\gamma, [\![\beta, [\![\alpha \Rightarrow \beta]\!]]\!])]$.
2. *Then* `pickDirectSubtype` $\texttt{t}_2$ *returns* $[\,]$.
3. *Then* `pickDirectSubtype` $\texttt{t}_3$ *returns* $[([\![\alpha, \alpha]\!], [\![\gamma]\!]);\ (\gamma, [\![[\![\alpha, \alpha]\!]]\!])]$.

```
let pickHeadType_opt t = ...
    : typ -> (typ * typ) list option
```

Given an intersection `t`, this function returns an option value with the content of a call to pickDirectSubtype when `t` (i.e. a list of all pairs of a typed picked out of `t` and a multitype of the the remaining ones). It `t` is not an intersection, the function return None.

Example. *Consider the types* $\texttt{t}_1 = [\![\beta, [\![\alpha \Rightarrow \beta]\!], \gamma]\!]$, $\texttt{t}_2 = \gamma$ *and* $\texttt{t}_3 = [\![\alpha, \alpha]\!] \Rightarrow \gamma$.

1. *Then* `pickHeadType_opt` $\texttt{t}_1$ *returns the option value* `Some` *containing the following list:* $[(\beta, [\![[\![\alpha \Rightarrow \beta]\!], \gamma]\!]);\ ([\![\alpha \Rightarrow \beta]\!], [\![\beta, \gamma]\!]);\ (\gamma, [\![\beta, [\![\alpha \Rightarrow \beta]\!]]\!])]$.
2. *Then* `pickHeadType_opt` $\texttt{t}_2$ *(resp.* `pickHeadType_opt` $\texttt{t}_3$*) returns* `None` *since one cannot pick a head type (intersection) from an arrow type (resp. base type).*

```
let pick2SplitType_opt t = ...
    : typ -> typ list list option
```

Given a an intersection `t`, this function return option value of a list of all possible splits of `t` as two intersections. If `t` is not an intersection, the function returns None.

Example. *Consider the types* $\texttt{t}_1 = [\![\beta, [\![\alpha \Rightarrow \beta]\!], \gamma]\!]$, $\texttt{t}_2 = \gamma$ *and* $\texttt{t}_3 = [\![\alpha, \alpha]\!] \Rightarrow \gamma$.

1. *Then* `pick2SplitType_opt` $\texttt{t}_1$ *returns the option value* `Some` *containing the following list:*

$$\begin{array}{l}[\quad [\ [\![\beta, [\![\alpha \Rightarrow \beta]\!], \gamma]\!]\ ;\ [\![]\!]\ ];\quad [\ [\![[\![\alpha \Rightarrow \beta]\!], \gamma]\!]\ ;\ [\![\beta]\!]\ ];\quad [\ [\![\beta, \gamma]\!]\ ;\ [\![[\![\alpha \Rightarrow \beta]\!]]\!]\ ];\\ \quad\ [\ [\![\gamma]\!]\ ;\ [\![\beta, [\![\alpha \Rightarrow \beta]\!]]\!]\ ];\quad [\ [\![\beta, [\![\alpha \Rightarrow \beta]\!]]\!]\ ;\ [\![\gamma]\!]\ ];\quad [\ [\![[\![\alpha \Rightarrow \beta]\!]]\!]\ ;\ [\![\beta, \gamma]\!]\ ];\\ \quad\ [\ [\![\beta]\!]\ ;\ [\![[\![\alpha \Rightarrow \beta]\!], \gamma]\!]\ ];\quad [\ [\![]\!]\ ;\ [\![\beta, [\![\alpha \Rightarrow \beta]\!], \gamma]\!]\ ] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad]\end{array}$$

2. *Then* `pick2SplitType_opt` $\texttt{t}_2$ *(resp.* `pickHeadType_opt` $\texttt{t}_3$*) returns* `None` *since one cannot split an arrow type (resp. base type).*

```
let pickNSplitType_opt i t = ...
    : int -> typ -> typ list list option
```

Given an integer `i` and an intersection `t`, this function returns an option value of a list of all possible splits of `t` as `i` intersections. If `t` is not an intersection, the function returns None.

Example. *See* `pick2SplitType_opt` *but with i splits.*

Finally, we consider the following function translating `typ` to `string` in order to define type printings.

```
typ_to_str = ...
    : typ -> string
```

4.2.2 Semantical Heads

There are two types of calls in the inhabitation algorithm: N-calls $N(\Gamma; \sigma)$ and H-calls $H^{x:[\tau]}(\Gamma; \sigma)$. H-calls have an additional variable (x) and singleton type ($[\tau]$) that have been marked out of the type context. This variable/singleton couple called head is used to guide the search of the algorithm.

The Ocaml file `TypeContext.ml` contains the implementation of the semantical heads.

As previously mentionned, heads are defined as couples of a variable and a type.

```
type head = Head of Variable.t * typ
```

In order to ensure that the type head is always a singleton, we introduce the following handling functions.

```
let semHead_opt var mTyp :
    : Variable.t -> typ -> head option
```

Given a variable identifier `var` and a singleton 'mTyp', this function returns an option value containing an head with such variable and type.

Example.

1. *The call* `semHead_opt 1 (TMult [TVar 1])` *returns the option value* `Some` *containing the head* `Head(1, TMult([TVar 1]))`.

2. *The calls* `semHead_opt 1 (TMult [])` *and* `semHead_opt 1 (TMult [TVar 1; TVar 2])` *both return* `None`.

```
let semHead var mTyp = get (semHead_opt var mTyp)
```

This function is the unsafe version of `semHead_opt`.

All head projections are defined as expected.

```
let getTyp head = ...           : head -> typ
let getVar = ...                : head -> Variable.t
```

Further projection can be defined.

```
let getStripTyp_opt head = ...  : head -> typ option
let getStripTyp head = ...      : head -> typ
```

Given a head `head`, the function `getStrypTyp_opt` returns an optional value containing the type of the semantical head singleton. If the type of the head is not a singleton, then the function returns `None`.

Example.

1. *The call* `getStripTyp_opt (Head(1, [TVar 1]))` *returns the option value* `Some` *containing the type* `TVar 1`.

2. *The calls* `getStripTyp_opt (Head(1, TMult []))`, `getStripTyp_opt (Head(1, TMult [TVar 1; TVar 2]))` *and* `getStripTyp_opt (Head(1, TArr(TVar 1, TVar 2)))` *all return* `None`.

The function `getStripTyp` is the unsafe version of the function `getStripTyp_opt`.

We finally consider two functions translating a head to a string in order to defube head printings.

```
let to_str head = ...            : head -> string
let to_str_pretty head = ...     : head -> string
```

4.2.3 Type Contexts

The Ocaml file `TypeContext.ml` contains the implementation of the notion of type contexts. Type contexts are build as a mapping from variables to intersections. We implement them using a map from variables to types.

```
type typCtxt = typ MultitypeMap.t
```

To ensure that variables are mapped only to intersections in type contexts, we introduce the following handeling functions.

```
let emptyCtxt = ...      : typCtxt
```

This function returns the empty type context.

```
let ctxtAdd_opt ctxt var mTyp = ...
    : typCtxt -> Variable.t -> typ -> typCtxt option
```

Given a type context `ctxt`, a variable `var` and a type `mTyp`, this function returns an optional value containing the given type context `ctxt` where the mapping of the variable `var` has been merged with the intersection `mTyp`. If the type `mTyp` is not an intersection, the function returns `None`.

Example. *Consider* $\Gamma = \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}$ *represented by a map with the following (variable, type) couples:*
`(1, TMult [TVar 1, TArr (TVar 2, TVar 1)]), (3, TMult [TVar 3])`.

1. *Then* `ctxtAdd_opt` Γ `3 (TMult [`α`])` *returns the value option* `Some` *containing a type context with the following variables and types couples:*
`(1, TMult [TVar 1, TArr (TVar 2, TVar 1)]), (3, TMult [TVar 1, TVar 3])` *which represents the type context:* $\Gamma + \{z : [\![\alpha]\!]\} = \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma, \alpha]\!]\}$.

2. *Then* `ctxtAdd_opt Γ 3 (TVar 1)` *and* `ctxtAdd_opt Γ 3 (TArr (TVar 2) (TVar 2))` *both return* `None`.

```
let ctxtAdd ctxt var mTyp = ...
    : typCtxt -> Variable.t -> typ -> typCtxt
```

This function is the unsafe version of `ctxtAdd_opt`.

```
let ctxtRemove ctxt var = ...
    : typCtxt -> Variable.t -> typCtxt
```

Given a type context `ctxt` and a variable `var`, this function returns the given type context `ctxt` where the value associated to the variable `var` have been removed.

Example. *Consider* $\Gamma = \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}$ *represented by a map with the following (variables, types) couples* `(1, TMult [TVar 1, TArr (TVar 2, TVar 1)]), (3, TMult [TVar 3])`. *Then* `ctxtAdd_opt Γ 1` *returns the type context with the following (variable, type) couples:* `(3, TMult [TVar 1, TVar 3])`.

```
let ctxtMerge ctxt1 ctxt2 = ...
    : typCtxt -> typCtxt -> typCtxt
```

Given two type contexts `ctxt1` and `ctxt2`, this function returns a type context whose map represents the union of the two type contexts `ctxt1` and `ctxt2`. For a given variable `var`, the binding in the resulting type context is given by merging the bindings of the two type context for that variable. If the variable is only bound in the first type context (resp. the second), then its binding in the resulting type context is the binding in the first (resp. second) type context. If the variable is unbound in both, then it is unbound in the resulting type context.

Example. *Consider the type contexts* $\Gamma_1 = \{x : [\![\alpha]\!], z : [\![\gamma]\!]\}$ *and* $\Gamma_2 = \{x : [\![\beta \Rightarrow \alpha]\!]\}$ *represented by the maps with following (variable, type) couples:*

`(1, TMult [TVar 1]), (3, TMult [TVar 3])`
and `(1, TMult [TArr (TVar 2, TVar 1)])`

Then `ctxtMerge` Γ_1 Γ_2 *returns a type context with the following (variable, type) couples:*

`(1, TMult [TVar 1, TArr (TVar 2, TVar 1)]), (3, TMult [TVar 3])`

which represents the type context $\Gamma_1 + \Gamma_2 = \{x : [\![alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}$.

```
let isEmpty ctxt = ...
    : typCtxt -> bool
```

Given a type context `ctxt`, this function returns a boolean with value `true` is the empty context and `false` otherwise.

```
let domain ctxt = ...
    : typCtxt -> VariableSet.t
```

Given a type context `ctxt`, this function returns a variable set containing every variable with a binding in the corresponding map.

```
let newVar set = ...
    : VariableSet.t -> Variable.t
```

Given a variable set `set`, this function returns a variable with a fresh name with respect to the given set `set`.

```
let pickHead ctxt = ...
    : typCtxt -> (Head.head * typCtxt) list
```

Given a type context `ctxt`, this function returns a list of all (head, type context) couples where the head is extracted from `ctxt` and the type context is obtained from `ctxt` by removing the head.

Example. *Consider* $\Gamma = \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}$. *Then* `pickHead` Γ *returns the list*

$$\begin{array}{lll} [& (x : [\![\alpha]\!], & \{x : [\![\beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}); \\ & (x : [\![\beta \Rightarrow \alpha]\!], & \{x : [\![\alpha]\!], z : [\![\gamma]\!]\}); \\ & (z : [\![\gamma]\!], & \{x : [\![\alpha, \beta \Rightarrow \alpha]\!]\}) \quad] \end{array}$$

```
let pickTwoSplitCtxt ctxt = ...
    : typCtxt -> typCtxt list list
```

Given a type context `ctxt`, this function returns a list of list of type contexts corresponding to all possible two splits of the given type context `ctxt`.

Example. *Consider the type context* $\Gamma = \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}$. *Then* `pickTwoSplitCtxt` Γ *returns the list:*

$$\begin{array}{llll} [& [& \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}; \quad \emptyset &]; \\ & [& \{x : [\![\alpha]\!], z : [\![\gamma]\!]\}; \quad \{x : [\![\beta \Rightarrow \alpha]\!]\} &]; \\ & [& \{x : [\![\alpha, \beta \Rightarrow \alpha]\!]\}; \quad \{z : [\![\gamma]\!]\} &]; \\ & [& \{x : [\![\alpha]\!]\}; \quad \{z : [\![\beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\} &]; \end{array}$$

$$\begin{array}{llll} [& \{x : [\![\beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\}; \quad \{x : [\![\alpha]\!]\} &]; \\ & [& \{z : [\![\gamma]\!]\}; \quad \{x : [\![\alpha, \beta \Rightarrow \alpha]\!]\} &]; \\ & [& \{x : [\![\beta \Rightarrow \alpha]\!]\}; \quad \{x : [\![\alpha]\!], z : [\![\gamma]\!]\} &]; \\ & [& \emptyset; \quad \{x : [\![\alpha, \beta \Rightarrow \alpha]\!], z : [\![\gamma]\!]\} &] \quad] \end{array}$$

```
let pickNSplitCtxt n ctxt = ...
    : int -> typCtxt -> typCtxt list list
```

Given an integer `n` and a type context `ctxt`, this function returns a list of list of type contexts corresponding to all possible `n` splits of the given type context `ctxt`.

Example. *See* `pickTwoSplitCtxt` *but with* `n` *splits.*

We finally consider two functions translating a type context into a string in order to define type context printings.

```
let ctxt_to_str ctxt = ...          : typCtxt -> string
let ctxt_to_str_pretty ctxt = ...   : typCtxt -> string
```

References