Quantitative Inhabitation in Call-by-Value

Victor Arrial
Université Paris Cité - IRIF - CNRS

Abstract

We show the decidability of the inhabitation problem in a quantitative Call-by-Value setting.

Inhabitation. Type systems are formalisms made of rules 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, in the sense that ”well-typed programs cannot go wrong” [15]. Typing in a given type system $\mathcal{X}$ is written $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$, where $t$ is a term, $\sigma$ is the type assigned to $t$, and $\Gamma$ is an environment assigning types to the (free) variables of $t$. The inhabitation problem naturally arises for any given type assignment system: given an environment $\Gamma$, and a type $\sigma$, decide whether there exists a term such that $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$. Inhabitation corresponds to decide the existence of a program (term $t$) that satisfies the given specification (type $\sigma$) under additional assumptions (environment $\Gamma$). Decidability of the inhabitation problem naturally provides tools for type-based program synthesis [14, 3], whose task is to construct—from scratch—a program that satisfies some high-level formal specification (the one guaranteed by the type assignment).

Quantitative Typing Systems. Intersection type assignment systems [8, 9] 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 ($\sigma \land \sigma = \sigma$). Intersection types allow terms having 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 [17]. However, intersection types constitute a powerful tool to reason about qualitative properties of programs, for example, there are intersection type systems characterizing different notions of normalization [16, 10], in the sense that a term $t$ is typable in a given system if and only if $t$ is normalizing for some particular notion. By removing idempotence [13, 11], 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 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) [12, 1]. In such a setting, typability is still undecidable, nevertheless inhabitation has proven to be decidable in the Call-by-Name case [6, 7]. So, an algorithm solving the inhabitation problem for a quantitative type system provides a decidable tool for type-based quantitative program synthesis, which aims to construct—from scratch—a program that satisfies some quantitative specification.

Call-by-Value. Call-by-Value evaluation is the most common parameter passing mechanism for programming languages: parameters are evaluated before being passed. We use the $\lambda_{\text{CBV}}$ calculus introduced by Accattoli and Paolini [2] which exploits explicit substitutions for both delaying CBV redex restrictions as well as acting at a distance, and which thus presents good
operational properties. This calculus can be equipped with a non-idempotent intersection type system [4] (presented in Fig. 1) which characterizes head normalization. We use multisets to denote intersections.

\[
\begin{align*}
\frac{x : M \vdash x : M}{\text{ax}} & \quad \frac{(\Gamma; x : M_i \vdash t : \sigma_i)_{i \in I}}{\text{abs}} \\
\frac{\Gamma_1 \vdash t : [M \Rightarrow \sigma] \quad \Gamma_2 \vdash u : M}{\Gamma_1 + \Gamma_2 \vdash tu : \sigma} & \quad \frac{\Gamma_1 \vdash x : M \vdash t : \sigma \quad \Gamma_2 \vdash u : M}{\Gamma_1 + \Gamma_2 \vdash t[x \backslash u] : \sigma}
\end{align*}
\]

Figure 1: Call-by-Value Type System \( \mathcal{V} \)

Contributions. We show that the inhabitation problem for the Call-by-Value \( \lambda \)-calculus with respect to the quantitative type system \( \mathcal{V} \) [4] is decidable. 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: we look for all and only such typable terms. This provides either a strong and powerful tool for (quantitative) program synthesis.

Canonical Derivations as Finite Basis: The set of all solutions of any instance of the inhabitation problem is either infinite or empty. Building an algorithm providing all solutions for a given instance is therefore worthless. However, following the technique used in the Call-by-Name \( \lambda \)-calculus case [6, 7] and generalizing it to explicit substitutions [5] allows us to introduce the notion of canonical derivation which later forms a finite basis for the solution set of each instance. It is a basis as it exactly generates each solution set through two simple operations: redex expansion and term plugging. We show how to compute the canonical representative of any solution as well as how it is recovered from its canonical representative, thus providing proofs of correction and completeness for each basis.

Basis Search Algorithm: Equipped with such tools, we provide an algorithm computing the basis for any given instance. It highly exploits a central property of any basis of terms called head subtype property which indicates that some of its types are contained in the context and thus helps guiding the search. This (non-deterministic) algorithm is shown to be correct and complete as it finds all and only basis terms. We show that it can in fact be deterministically simulated in a finite time, which provides for free a proof of the finiteness of each basis. It therefore constitutes a program synthesis mechanism for an expressive programming language.

References


