Which Simple Types Have a Unique Inhabitant?

Gabriel Scherer Didier Rémy

Gallium, INRIA Paris-Rocquencourt, France

{gabriel.scherer,didier.remy}@inria.fr

Abstract

We study the question of whether a given type has a unique inhabitant modulo program equivalence. In the setting of simply-typed lambda-calculus with sums, equipped with the strong $\beta\eta$-equivalence, we show that uniqueness is decidable. We present a *saturating* focused logic that introduces irreducible cuts on positive types "as soon as possible". Backward search in this logic gives an effective algorithm that returns either zero, one or two distinct inhabitants for any given type. Preliminary application studies show that such a feature can be useful in strongly-typed programs, inferring the code of highly-polymorphic library functions, or "glue code" inside more complex terms.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords Unique inhabitants, proof search, simply-typed lambda-calculus, focusing, canonicity, sums, saturation, code inference

1. Introduction

In this article, we answer an instance of the following question: "Which types have a unique inhabitant"? In other words, for which type is there exactly one program of this type? Which logical statements have exactly one proof term?

To formally consider this question, we need to choose one specific type system, and one specific notion of equality of programs – which determines uniqueness. In this article, we work with the simply-typed λ-calculus with atoms, functions, products and sums as our type system, and we consider programs modulo $\beta\eta$-equivalence. We show that unique inhabitation is decidable in this setting; we provide and prove correct an algorithm to answer it, and suggest several applications for it. This is only a first step: simply-typed calculus *with sums* is, in some sense, the simplest system in which the question is delicate enough to be interesting. We hope that our approach can be extended to richer type systems – with polymorphism, dependent types, and substructural logics.

ICFP'15, August 31 – September 2, 2015, Vancouver, British Columbia, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/2784731.2784757

For reasons of space, the proofs of the formal results are only present in the long version of this article (Scherer and Rémy 2015).

1.1 Why Unique?

We see three different sources of justification for studying uniqueness of inhabitation: practical use of code inference, programming language design, and understanding of type theory.

In practice, if the context of a not-yet-written code fragment determines a type that is uniquely inhabited, then the programming system can automatically fill the code. This is a strongly principal form of code inference: it cannot guess wrong. Some forms of code completion and synthesis have been proposed (Perelman, Gulwani, Ball, and Grossman 2012; Gvero, Kuncak, Kuraj, and Piskac 2013), to be suggested interactively and approved by the programmer. Here, the strong restriction of uniqueness would make it suitable for a code elaboration pass at compile-time: it is of different nature. Of course, a strong restriction also means that it will be applicable less often. Yet we think it becomes a useful tool when combined with strongly typed, strongly specified programming disciplines and language designs – we have found in preliminary work (Scherer 2013) potential use cases in dependently typed programming. The simply-typed lambda-calculus is very restricted compared to dependent types, or even the type systems of ML, System F, *etc.* used in practice in functional programming languages; but we have already found a few examples of applications (Section 6). This shows promises for future work on more expressive type systems.

For programming language design, we hope that a better understanding of the question of unicity will let us better understand, compare and extend other code inference mechanisms, keeping the question of coherence, or non-ambiguity, central to the system. Type classes or implicits have traditionally been presented (Wadler and Blott 1989; Stuckey and Sulzmann 2002; Oliveira, Schrijvers, Choi, Lee, Yi, and Wadler 2014) as a mechanism for elaboration, solving a constraint or proof search problem, with coherence or non-ambiguity results proved as a second step as a property of the proposed elaboration procedure. Reformulating coherence as a unique inhabitation property, it is not anymore an operational property of the specific search/elaboration procedure used, but a semantic property of the typing environment and instance type in which search is performed. Non-ambiguity is achieved not by fixing the search strategy, but by building the right typing environment from declared instances and potential conflict resolution policies, with a general, mechanism-agnostic procedure validating that the resulting type judgments are uniquely inhabited.

In terms of type theory, unique inhabitation is an occasion to take inspiration from the vast literature on proof inhabitation and proof search, keeping relevance in mind: all proofs of the same statement may be equally valid, but programs at a given type are distinct in important and interesting ways. We use *focusing* (Andreoli 1992), a proof search discipline that is more canonical (enumerates less duplicates of each proof term) than simply goal-directed proof search, and its recent extension into (maximal) *multi-focusing* (Chaudhuri, Miller, and Saurin 2008).

1.2 Example Use Cases

Most types that occur in a program are, of course, not uniquely inhabited. Writing a term at a type that happens to be uniquely inhabited is a rather dull part of the programming activity, as they are no meaningful choices. While we do not hope unique inhabitants would cure all instances of boring programming assignment, we have identified two areas where they may be of practical use:

- inferring the code of highly parametric (strongly specified) auxiliary functions
- inferring fragments of glue code in the middle of a more complex (and not uniquely determined) term

For example, if you write down the signature of `flip` $\forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$ to document your standard library, you should not have to write the code itself. The types involved can be presented equivalently as simple types, replacing prenex polymorphic variables by uninterpreted atomic types (X, Y, Z...). Our algorithm confirms that $(X \to Y \to Z) \to (Y \to X \to Z)$ is uniquely inhabited and returns the expected program – same for `curry` and `uncurry`, `const`, etc.

In the middle of a term, you may have forgotten whether the function `proceedings` excepts a `conf` as first argument and a `year` as second argument, or the other way around. Suppose a language construct `?!` that infers a unique inhabitant at its expected type (and fails if there are several choices), understanding abstract types (such as `year`) as uninterpreted atoms. You can then write `(?! proceedings icfp this_year)`, and let the programming system infer the unique inhabitant of either (`conf` $\to$ `year` $\to$ `proceedings`) $\to$ (`conf` $\to$ `year` $\to$ `proceedings`) or (`conf` $\to$ `year` $\to$ `proceedings`) $\to$ (`year` $\to$ `conf` $\to$ `proceedings`) depending on the actual argument order – it would also work for `conf` $*$ `year` $\to$ `proceedings`, etc.

1.3 Aside: Parametricity?

Can we deduce unique inhabitation from the free theorem of a sufficiently parametric type? We worked out some typical examples, and our conclusion is that this is not the right approach. Although it was possible to derive uniqueness from a type's parametric interpretation, proving this implication (from the free theorem to uniqueness) requires arbitrary reasoning steps, that is, a form of proof search. If we have to implement proof search mechanically, we may as well work with convenient syntactic objects, namely typing judgments and their derivations.

For example, the unary free theorem for the type of composition $\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$ tells us that for any sets of terms $S_\alpha, S_\beta, S_\gamma$, if f and g are such that, for any $a \in S_\alpha$ we have $f\ a \in S_\beta$, and for any $b \in S_\beta$ we have $g\ b \in S_\gamma$, and if t is of the type of composition, then for any $a \in S_\alpha$ we have $t\ f\ g\ a \in S_\gamma$. The reasoning to prove unicity is as follows. Suppose we are given functions (terms) f and g. For any term a, first define $S_\alpha \stackrel{\text{def}}{=} \{a\}$. Because we wish f to map elements of S_α to S_β, define $S_\beta \stackrel{\text{def}}{=} \{f\ a\}$. Then, because we wish g to map elements of S_β to S_γ, define $S_\gamma \stackrel{\text{def}}{=} \{g\ (f\ a)\}$. We have that $t\ f\ g\ a$ is in S_γ, thus $t\ f\ g$ is uniquely determined as $\lambda a.\, g\ (f\ a)$.

This reasoning exactly corresponds to a (forward) proof search for the type $\alpha \to \gamma$ in the environment $\alpha, \beta, \gamma, f : \alpha \to \beta, g : \beta \to \gamma$. We know that we can always start with a λ-abstraction (formally, arrow-introduction is an invertible rule), so introduce $x : \alpha$ in the context and look for a term of type γ. This type has no head constructor, so no introduction rules are available; we shall look for an elimination (function application or pair projection). The only elimination we can perform from our context is the application $f\ x$, which gives a β. From this, the only elimination we can perform is the application $g\ (f\ x)$, which gives a γ. This has the expected goal type: our full term is $\lambda x.\, g\ (f\ x)$. It is uniquely determined, as we never had a choice during term construction.

1.4 Formal Definition of Equivalence

We recall the syntax of the simply-typed lambda-calculus types (Figure 1), terms (Figure 2) and neutral terms. The standard typing judgment $\Delta \vdash t : A$ is recalled in Figure 3, where Δ is a general context mapping term variables to types. The equivalence relation we consider, namely $\beta\eta$-equivalence, is defined as the least congruence satisfying the equations of Figure 4. Writing $t : A$ in an equivalence rule means that the rule only applies when the subterm t has type A – we only accept equivalences that preserve well-typedness.

$$
\begin{array}{lll}
A, B, C, D ::= & & \text{types} \\
& \mid X, Y, Z & \text{atoms} \\
& \mid P, Q & \text{positive types} \\
& \mid N, M & \text{negative types} \\
P, Q & ::= A + B & \text{strict positive} \\
N, M & ::= A \to B \mid A * B & \text{strict negative} \\
P_{\mathsf{at}}, Q_{\mathsf{at}} & ::= P, Q \mid X, Y, Z & \text{positive or atom} \\
N_{\mathsf{at}}, M_{\mathsf{at}} & ::= N, M \mid X, Y, Z & \text{negative or atom}
\end{array}
$$

Figure 1. Types of the simply-typed calculus

$$
\begin{array}{lll}
t, u, r ::= & & \text{terms} \\
& \mid x, y, z & \text{variables} \\
& \mid \lambda x.\, t & \lambda\text{-abstraction} \\
& \mid t\ u & \text{application} \\
& \mid (t, u) & \text{pair} \\
& \mid \pi_i\ t & \text{projection } (i \in \{1, 2\}) \\
& \mid \sigma_i\ t & \text{sum injection } (i \in \{1, 2\}) \\
& \mid \delta(t,\ x_1.u_1,\ x_2.u_2) & \text{sum elimination (case split)} \\
\\
n, m & := x, y, z \mid \pi_i\ n \mid n\ t & \text{neutral terms}
\end{array}
$$

Figure 2. Terms of the lambda-calculus with sums

$$
\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.\, t : A \to B}
\qquad
\frac{\Delta \vdash t : A \to B \qquad \Delta \vdash u : A}{\Delta \vdash t\ u : B}
$$

$$
\frac{\Delta \vdash t : A \qquad \Delta \vdash u : B}{\Delta \vdash (t, u) : A * B}
\qquad
\frac{\Delta \vdash t : A_1 * A_2}{\Delta \vdash \pi_i\ t : A_i}
$$

$$
\Delta, x : A \vdash x : A
\qquad
\frac{\Delta \vdash t : A_i}{\Delta \vdash \sigma_i\ t : A_1 + A_2}
$$

$$
\frac{\Delta \vdash t : A_1 + A_2 \qquad \Delta, x_1 : A_1 \vdash u_1 : C \qquad \Delta, x_2 : A_2 \vdash u_2 : C}{\Delta \vdash \delta(t,\ x_1.u_1,\ x_2.u_2) : C}
$$

Figure 3. Typing rules for the simply-typed lambda-calculus

We distinguish *positive* types, *negative* types, and atomic types. The presentation of focusing (subsection 1.6) will justify this distinction. The equivalence rules of Figure 4 make it apparent that

$$(\lambda x.\, t)\ u \to_\beta u[t/x] \qquad (t : A \to B) =_\eta \lambda x.\, t\ x$$

$$\pi_i\ (t_1, t_2) \to_\beta t_i \qquad (t : A * B) =_\eta (\pi_1\ t, \pi_2\ t)$$

$$\delta(\sigma_i\ t,\ x_1.u_1,\ x_2.u_2) \to_\beta u_i[t/x_i]$$

$$\forall C[\square], \quad C[t : A + B] =_\eta \delta(t,\ x.C[\sigma_1\ x],\ x.C[\sigma_2\ x])$$

Figure 4. $\beta\eta$-equivalence for the simply-typed lambda-calculus

the η-equivalence rule for sums is more difficult to handle than the other η-rule, as it quantifies on any term context $C[\square]$. More generally, systems with only negative, or only positive types have an easier equational theory than those with mixed polarities. In fact, it is only at the end of the 20th century (Ghani 1995; Altenkirch, Dybjer, Hofmann, and Scott 2001; Balat, Di Cosmo, and Fiore 2004; Lindley 2007) that decision procedures for equivalence in the lambda-calculus with sums were first proposed.

Can we reduce the question of unicity to deciding equivalence? One would think of enumerating terms at the given type, and using an equivalence test as a post-processing filter to remove duplicates: as soon as one has found two distinct terms, the type can be declared non-uniquely inhabited. Unfortunately, this method does not give a terminating decision procedure, as naive proof search may enumerate infinitely many equivalent proofs, taking infinite time to post-process. We need to integrate canonicity in the structure of proof search itself.

1.5 Terminology

We distinguish and discuss the following properties:

- *provability completeness*: A search procedure is complete for provability if, for any type that is inhabited in the unrestricted type system, it finds at least one proof term.
- *unicity completeness*: A search procedure is complete for unicity if it is complete for provability and, if there exists two proofs distinct as programs in the unrestricted calculus, then the search finds at least two proofs distinct as programs.
- *computational completeness*: A search procedure is computationally complete if, for any proof term t in the unrestricted calculus, there exists a proof in the restricted search space that is equivalent to t as a program. This implies both previous notions of completeness.
- *canonicity*: A search procedure is canonical if it has no duplicates: any two enumerated proofs are distinct as programs. Such procedures require no filtering of results after the fact. We will say that a system is *more canonical* than another if it enumerates less redundant terms, but this does not imply canonicity.

There is a tension between computational completeness and termination of the corresponding search algorithm: when termination is obtained by cutting the search space, it may remove some computational behaviors. Canonicity is not a strong requirement: we could have a terminating, unicity-complete procedure and filter duplicates after the fact, but have found no such middle-ground. This article presents a logic that is both *computationally complete* and *canonical* (Section 3), and can be restricted (Section 4) to obtain a *terminating* yet *unicity-complete* algorithm (Section 5).

1.6 Focusing for a Less Redundant Proof Search

Focusing (Andreoli 1992) is a generic search discipline that can be used to restrict redundancy among searched proofs; it relies on the general idea that some proof steps are *invertible* (the premises are provable exactly when the conclusion is, hence performing this step during proof search can never lead you to a dead-end) while others are not. By imposing an order on the application of invertible and non-invertible proof steps, focusing restricts the number of valid proofs, but it remains complete for provability and, in fact, computationally complete (§1.5).

More precisely, a focused proof system alternates between two phases of proof search. During the *invertible phase*, rules recognized as invertible are applied as long as possible – this stops when no invertible rule can be applied anymore. During the *non-invertible phase*, non-invertible rules are applied in the following way: a formula (in the context or the goal) is chosen as the *focus*, and non-invertible rules are applied as long as possible.

For example, consider the judgment $x : X + Y \vdash X + Y$. Introducing the sum on the right by starting with a σ_1 ? or σ_2 ? would be a non-invertible proof step: we are permanently committing to a choice – which would here lead to a dead-end. On the contrary, doing a case-split on the variable x is an invertible step: it leaves all our options open. For non-focused proof search, simply using the variable $x : X + Y$ as an axiom would be a valid proof term. It is not a valid focused proof, however, as the case-split on x is a possible invertible step, and invertible rules must be performed as long as they are possible. This gives a partial proof term $\delta(x,\ y.?,\ z.?)$, with two subgoals $y : X \vdash X + Y$ and $z : X \vdash X + Y$; for each of them, no invertible rule can be applied anymore, so one can only *focus* on the goal and do an injection. While the non-focused calculus had two syntactically distinct but equivalent proofs, x and $\delta(x,\ y.\sigma_1\ y,\ z.\sigma_2\ z)$, only the latter is a valid focused proof: redundancy of proof search is reduced.

The interesting steps of a proof are the non-invertible ones. We call *positive* the type constructors that are "interesting to introduce". Conversely, their *elimination* rule is invertible (sums). We call *negative* the type constructors that are "interesting to eliminate", that is, whose *introduction* rule is invertible (arrow and product). While the mechanics of focusing are logic-agnostic, the polarity of constructors depends on the specific inference rules; linear logic needs to distinguish positive and negative products. Some focused systems also assign a polarity to atomic types, which allows to express interesting aspects of the dynamics of proof search (positive atoms correspond to *forward search*, and negative atoms to *backward search*). In Section 2 we present a simple focused variant of natural deduction for intuitionistic logic.

1.7 Limitations of Focusing

In absence of sums, focused proof terms correspond exactly to β-short η-long normal forms. In particular, focused search is *canonical* (§1.5). However, in presence of both polarities, focused proofs are not canonical anymore. They correspond to η-long form for the strictly weaker eta-rule defined without context quantification $x : A + B =_{\text{weak-}\eta} \delta(t,\ x.\sigma_1\ x,\ y.\sigma_2\ y)$.

This can be seen for example on the judgment $z : Z, x : Z \to X + Y \vdash X + Y$, a variant on the previous example where the sum in the context is "thunked" under a negative datatype. The expected proof is $\delta(x\ \ z,\ y_1.\sigma_1\ \ y_1,\ y_2.\sigma_2\ \ y_2)$, but the focused discipline will accept infinitely many equivalent proof terms, such as $\delta(x\ z,\ y_1.\sigma_1\ y_1,\ y_2.\delta(x\ z,\ y_1'.\sigma_1\ y_1',\ \_.\sigma_2\ y_2))$. The result of the application $x\ z$ can be matched upon again and again without breaking the focusing discipline.

This limitation can also be understood as a strength of focusing: despite equalizing more terms, the focusing discipline can still be used to reason about impure calculi where the eliminations corresponding to non-invertible proof terms may perform side-effects, and thus cannot be reordered, duplicated or dropped. As we work on pure, terminating calculi – indeed, even adding non-termination as an uncontrolled effect ruins unicity – we need a stronger equational theory than suggested by focusing alone.

1.8 Our Idea: Saturating Proof Search

Our idea is that instead of only deconstructing the sums that appear immediately as the top type constructor of a type in context, we shall deconstruct all the sums that can be reached from the context by applying eliminations (function application and pair projection). Each time we introduce a new hypothesis in the context, we *saturate* it by computing all neutrals of sum type that can be built using this new hypothesis. At the end of each saturation phase, all the positives that could be deduced from the context have been deconstructed, and we can move forward applying non-invertible rules on the goal. Eliminating negatives until we get a positive and matching in the result corresponds to a cut (which is not reducible, as the scrutinee is a neutral term), hence our technique can be summarized as "*Cut the positives as soon as you can*".

The idea was inspired by Sam Lindley's equivalence procedure for the lambda-calculus with sums, whose rewriting relation can be understood as moving case-splits *down* in the derivation tree, until they get blocked by the introduction of one of the variable appearing in their scrutinee (so moving down again would break scoping) – this also corresponds to "restriction (A)" in Balat, Di Cosmo, and Fiore (2004). In our saturating proof search, after introducing a new formal parameter in the context, we look for all possible new scrutinees using this parameter, and case-split on them. Of course, this is rather inefficient as most proofs will in fact not make use of the result of those case-splits, but this allows to give a common structure to all possible proofs of this judgment.

In our example $z : Z, x : Z \to X + Y \vdash X + Y$, the saturation discipline requires to cut on $x\ z$. But after this sum has been eliminated, the newly introduced variables $y_1 : X$ or $y_2 : Y$ do not allow to deduce new positives – we would need a new Z for this. Thus, saturation stops and focused search restarts, to find a unique normal form $\delta(x\ z,\ y_1.\sigma_1\ y_1,\ y_2.\sigma_2\ y_2)$. In Section 3 we show that saturating proof search is *computationally complete* and *canonical* (§1.5).

1.9 Termination

The saturation process described above does not necessarily terminate. For example, consider the type of Church numerals specialized to a positive $X+Y$, that is, $X+Y \to (X+Y \to X+Y) \to X + Y$. Each time we cut on a new sum $X + Y$, we get new arguments to apply to the function $(X + Y \to X + Y)$, giving yet another sum to cut on.

In the literature on proof search for propositional logic, the usual termination argument is based on the subformula property: in a closed, fully cut-eliminated proof, the formulas that appear in subderivations of subderivations are always subformulas of the formulas of the main judgment. In particular, in a logic where judgments are of the form $S \vdash A$ where S is a finite *set* of formulas, the number of distinct judgments appearing in subderivations is finite (there is a finite number of subformulas of the main judgment, and thus finitely many possible finite sets as contexts). Finally, in a goal-directed proof search process, we can kill any recursive subgoals whose judgment already appears in the path from the root of the proof to the subgoal. There is no point trying to complete a partial proof P_{above} of $S \vdash A$ as a strict subproof of a partial proof P_{below} of the same $S \vdash A$ (itself a subproof of the main judgment): if there is a closed subproof for P_{above}, we can use that subproof directly for P_{below}, obviating the need for proving P_{above} in the first place. Because the space of judgments is finite, a search process forbidding such recurring judgments always terminates.

We cannot directly apply this reasoning, for two reasons.

- Our contexts are mapping from term variables to formulas or, seen abstractly, *multisets* of formulas; even if the space of possible formulas is finite for the same reason as above, the space of multisets over them is still infinite.
- Erasing such multiset to sets, and cutting according to the non-recurrence criteria above, breaks *unicity completeness* (§1.5). Consider the construction of Church numerals by a judgment of the form $x : X, y : X \to X \vdash X$. One proof is just x, and all other proofs require providing an argument of type X to the function y, which corresponds to a subgoal that is equal to our goal; they would be forbidden by the no-recurrence discipline.

We must adapt these techniques to preserve not only *provability completeness*, but also *unicity completeness* (§1.5). Our solution is to use *bounded multisets* to represent contexts and collect recursive subgoals. We store at most M variables for each given formula, for a suitably chosen M such that if there are two different programs for a given judgment $\Delta \vdash A$, then there are also two different programs for $\lfloor\Delta\rfloor_M \vdash A$, where $\lfloor\Delta\rfloor_M$ is the bounded erasure keeping at most M variables at each formula.

While it seems reasonable that such a M exists, it is not intuitively clear what its value is, or whether it is a constant or depends on the judgment to prove. Could it be that a given goal A is provable in two different ways with four copies of X in the context, but uniquely inhabited if we only have three X?

In Section 4 we prove that $M \overset{\mathsf{def}}{=} 2$ suffices. In fact, we prove a stronger result: for any $n \in \mathbb{N}$, keeping at most n copies of each formula in context suffices to find at least n distinct proofs of any goal, if they exist.

For recursive subgoals as well, we only need to remember at most 2 copies of each subgoal: if some P_{above} appears as the subgoal of P_{below} and has the same judgment, we look for a closed proof of P_{above}. Because it would also have been a valid proof for P_{below}, we have found two proofs for P_{below}: the one using P_{above} and its closed proof, and the closed proof directly. P_{above} itself needs not allow new recursive subgoal at the same judgment, so we can kill any subgoal that has at least two ancestors with the same judgment while preserving completeness for unicity (§1.5).

1.10 Contributions

We show that the unique inhabitation problem for simply-typed lambda-calculus for sums is decidable, and propose an effective algorithm for it. Given a context and a type, it answers that there are zero, one, or "at least two" inhabitants, and correspondingly provides zero, one, or two distinct terms at this typing. Our algorithm relies on a novel *saturating* focused logic for intuitionistic natural deduction, with strong relations to the idea of *maximal multi-focusing* in the proof search literature (Chaudhuri, Miller, and Saurin 2008), that is both *computationally complete* (§1.5) and *canonical* with respect to $\beta\eta$-equivalence.

We provide an approximation result for program multiplicity of simply-typed derivations with bounded contexts. We use it to show that our *terminating* algorithm is *complete for unicity* (§1.5), but it is a general result (on the common, non-focused intuitionistic logic) that is of independent interest.

Finally, we present preliminary studies of applications for code inference. While extension to more realistic type systems is left for future work, simply-typed lambda-calculus with atomic types already allow to encode some prenex-polymorphic types typically found in libraries of strongly-typed functional programs.

2. Intuitionistic Focused Natural Deduction

$$\begin{array}{lll} \Gamma & ::= \texttt{varmap}(N_{\mathsf{at}}) & \text{negative or atomic context} \\ \Delta & ::= \texttt{varmap}(A) & \text{general context} \end{array}$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{inv}} t : A \qquad \Gamma; \Delta \vdash_{\mathsf{inv}} u : B}{\Gamma; \Delta \vdash_{\mathsf{inv}} (t, u) : A * B}\ \textsc{inv-pair}$$

$$\frac{\Gamma; \Delta, x : A \vdash_{\mathsf{inv}} t : C \qquad \Gamma; \Delta, x : B \vdash_{\mathsf{inv}} u : C}{\Gamma; \Delta, x : A + B \vdash_{\mathsf{inv}} \delta(x,\ x.t,\ x.u) : C}\ \textsc{inv-sum}$$

$$\frac{\Gamma; \Delta, x : A \vdash_{\mathsf{inv}} t : B}{\Gamma; \Delta \vdash_{\mathsf{inv}} \lambda x.\, t : A \to B}\ \textsc{inv-arr} \qquad \frac{\Gamma, \Gamma' \vdash_{\mathsf{foc}} t : P_{\mathsf{at}}}{\Gamma; \Gamma' \vdash_{\mathsf{inv}} t : P_{\mathsf{at}}}\ \textsc{inv-end}$$

$$\frac{\Gamma \vdash t \Uparrow P}{\Gamma \vdash_{\mathsf{foc}} t : P}\ \textsc{foc-intro} \qquad \frac{\Gamma \vdash n \Downarrow X}{\Gamma \vdash_{\mathsf{foc}} n : X}\ \textsc{foc-atom}$$

$$\frac{\Gamma \vdash n \Downarrow P \qquad \Gamma; x : P \vdash_{\mathsf{inv}} t : Q_{\mathsf{at}}}{\Gamma \vdash_{\mathsf{foc}} \texttt{let}\, x = n\, \texttt{in}\, t : Q_{\mathsf{at}}}\ \textsc{foc-elim} \qquad \frac{\Gamma \vdash t \Uparrow A_i}{\Gamma \vdash \sigma_i\ t \Uparrow A_1 + A_2}\ \textsc{intro-sum}$$

$$\frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} t : N_{\mathsf{at}}}{\Gamma \vdash t \Uparrow N_{\mathsf{at}}}\ \textsc{intro-end} \qquad \frac{\Gamma \vdash n \Downarrow A_1 * A_2}{\Gamma \vdash \pi_i\ n \Downarrow A_i}\ \textsc{elim-pair} \qquad \frac{(x : N_{\mathsf{at}}) \in \Gamma}{\Gamma \vdash x \Downarrow N_{\mathsf{at}}}\ \textsc{elim-start}$$

$$\frac{\Gamma \vdash n \Downarrow A \to B \qquad \Gamma \vdash u \Uparrow A}{\Gamma \vdash n\ u \Downarrow B}\ \textsc{elim-arr}$$

Figure 5. Cut-free focused natural deduction for intuitionistic logic

In Figure 5 we introduce a focused natural deduction for intuitionistic logic, as a typing system for the simply-typed lambda-calculus – with an explicit `let` construct. It is relatively standard, strongly related to the linear intuitionistic calculus of Brock-Nannestad and Schürmann (2010), or the intuitionistic calculus of Krishnaswami (2009). We distinguish four judgments: $\Gamma; \Delta \vdash_{\mathsf{inv}} t : A$ is the *invertible* judgment, $\Gamma \vdash_{\mathsf{foc}} t : P_{\mathsf{at}}$ the *focusing* judgment, $\Gamma \vdash t \Uparrow A$ the *non-invertible introduction* judgment and $\Gamma \vdash n \Downarrow A$ the *non-invertible elimination* judgment. The system is best understood by following the "life cycle" of the proof search process (forgetting about proof terms for now), which initially starts with a sequent to prove of the form $\emptyset; \Delta \vdash_{\mathsf{inv}} ? : A$.

During the invertible phase $\Gamma; \Delta \vdash_{\mathsf{inv}} ? : A$, invertible rules are applied as long as possible. We defined *negative* types as those whose introduction in the goal is invertible, and *positives* as those whose elimination in the context is invertible. Thus, the invertible phase stops only when all types in the context are negative, and the goal is positive or atomic: this is enforced by the rule INV-END. The two contexts correspond to an "old" context Γ, which is negative or atomic (all positives have been eliminated in a previous invertible phase), and a "new" context Δ of any polarity, which is the one being processed by invertible rule. INV-END only applies when the new context Γ' is negative or atomic, and the goal P_{at} positive or atomic.

The focusing phase $\Gamma \vdash_{\mathsf{foc}} ? : P_{\mathsf{at}}$ is where choices are made: a sequence of non-invertible steps will be started, and continue as long as possible. Those non-invertible steps may be eliminations in the context (FOC-ELIM), introductions of a strict positive in the goal (FOC-INTRO), or conclusion of the proof when the goal is atomic (FOC-ATOM).

In terms of search process, the introduction judgment $\Gamma \vdash ? \Uparrow A$ should be read from the bottom to the top, and the elimination judgment $\Gamma \vdash ? \Downarrow A$ from the top to the bottom. Introductions correspond to backward reasoning (to prove $A_1 + A_2$ it *suffices* to prove A_i); they must be applied as long as the goal is positive, to end on negatives or atoms (INTRO-END) where invertible search takes over. Eliminations correspond to forward reasoning (from the hypothesis $A_1 * A_2$ we can *deduce* A_i) started from the context (ELIM-START); they must also be applied as long as possible, as they can only end in the rule FOC-ELIM on a strict positive, or in the rule FOC-ATOM on an atom.

Sequent-Style Left Invertible Rules The left-introduction rule for sums INV-SUM is sequent-style rather than in the expected natural deduction style: we only destruct variables found in the context, instead of allowing to destruct arbitrary expressions. We also shadow the matched variable, as we know we will never need the sum again.

Let-Binding The proof-term $\texttt{let}\ x = n\ \texttt{in}\ t$ used in the FOC-ELIM rule is not part of the syntax we gave for the simply-typed lambda-calculus in Section 1.4. Indeed, focusing re-introduces a restricted cut rule which does not exist in standard natural deduction. We could write $t[n/x]$ instead, to get a proper λ-term – and indeed when we speak of focused proof term as λ-term this substitution is to be understood as implicit. We prefer the `let` syntax which better reflects the dynamics of the search it witnesses.

We call $\texttt{letexp}(t)$ the λ-term obtained by performing let-expansion (in depth) on t, defined by the only non-trivial case:

$$\texttt{letexp}(\texttt{let}\, x = n\, \texttt{in}\, t) \quad \stackrel{\mathsf{def}}{=} \quad \texttt{letexp}(t)[\texttt{letexp}(n)/x]$$

Normality If we explained $\texttt{let}\ x = n\ \texttt{in}\ t$ as syntactic sugar for $(\lambda x.\, t)\ n$, our proofs term would contain β-redexes. We prefer to explain them as a notation for the substitution $t[n/x]$, as it is then apparent that proof term for the focused logic are in β-normal form. Indeed, x being of strictly positive type, it is necessarily a sum and is destructed in the immediately following invertible phase by a rule INV-SUM (which shadows the variable, never to be used again). As the terms corresponding to non-invertible introductions $\Gamma \vdash n \Downarrow P$ are all neutrals, the substitution creates a subterm of the form $\delta(n,\ x.t,\ x.u)$ with no new redex.

One can also check that proof terms for judgments that do not contain sums are in η-long normal form. For example, a subterm of type $A \to B$ is either type-checked by an invertible judgment $\Gamma; \Delta \vdash_{\mathsf{inv}} t : A \to B$ or an elimination judgment $\Gamma \vdash n \Downarrow A \to B$. In the first case, the invertible judgment is either a sum elimination (excluded by hypothesis) or a function introduction $\lambda x.\, u$. In the second case, because an elimination phase can only end on a positive or atomic type, we know that immediately below is the elimination rule for arrows: it is applied to some argument, and η-expanding it would create a β-redex.

Fact 1. *The focused intuitionistic logic is complete for provability. It is also computationally complete (§1.5).*

2.1 Invertible Commuting Conversions

The *invertible commuting conversion* (or *invertible commutative cuts*) relation ($=_{\mathsf{icc}}$) expresses that, inside a given invertible phase, the ordering of invertible step does not matter.

$$\delta(t,\, x.\lambda y_1.\, u_1,\, x.\lambda y_2.\, u_2) =_{\mathsf{icc}} \lambda y.\, \delta(t,\, x.u_1[y/y_1],\, x.u_2[y/y_2])$$

$$\delta(t,\, x.(u_1,u_2),\, x.(r_1,r_2)) =_{\mathsf{icc}}$$
$$(\delta(t,\, x.u_1,\, x.r_1), \delta(t,\, x.u_2,\, x.r_2))$$

$$\delta(t,\, x.\delta(u,\, y.r_1,\, y.r_1'),\, x.\delta(u,\, y.r_2,\, y.r_2')) =_{\mathsf{icc}}$$
$$\delta(u,\, y.\delta(t,\, x.r_1,\, x.r_2),\, x.\delta(t,\, x.r_1',\, x.r_2'))$$

This equivalence relation is easily decidable. We could do without it. We could force a specific operation order by restricting typing rules, typically by making Δ a list to enforce sum-elimination order, and requiring the goal C of sum-eliminations to be positive or atomic to enforce an order between sum-eliminations and invertibles utintroductions. We could also provide more expressive syntactic forms (parallel multi-sums elimination (Altenkirch, Dybjer, Hofmann, and Scott 2001)) and normalize to this more canonical syntax. We prefer to make the non-determinism explicit in the specification. Our algorithm uses some implementation-defined order for proof search, it never has to compute $(=_{\mathsf{icc}})$-convertibility.

Note that there are term calculi (Curien and Munch-Maccagnoni 2010) inspired from sequent-calculus, where commuting conversions naturally correspond to computational reductions, which would form better basis for studying normal forms than λ-terms. In the present work we wished to keep a term language resembling functional programs.

3. A Saturating Focused System

In this section, we introduce the novel *saturating* focused proof search, again as a term typing system that is both *computationally complete* (§1.5) and *canonical*. It serves as a specification of our normal forms; our algorithm shall only search for a finite subspace of saturated proofs, while remaining *unicity complete*.

Saturated focusing logic is a variant of the previous focused natural deduction, where the *focusing* judgment $\Gamma \vdash_{\mathsf{foc}} t : P_{\mathsf{at}}$ is replaced by a *saturating* judgment $\Gamma; \Gamma' \vdash_{\mathsf{sat}} t : P_{\mathsf{at}}$. The system is presented in Figure 6; the rules for non-invertible elimination and introductions, and the invertible rules, are identical to the previous ones and have not been repeated.

(rules for $\Gamma \vdash t \Uparrow A$ and $\Gamma \vdash n \Downarrow A$ as in Figure 5)
(invertible rules, except INV-END, as in Figure 5)

$$\frac{\Gamma; \Gamma' \vdash_{\mathsf{sat}} t : P_{\mathsf{at}}}{\Gamma; \Gamma' \vdash_{\mathsf{sinv}} t : P_{\mathsf{at}}}\,\textsc{sinv-end} \qquad \frac{\Gamma \vdash t \Uparrow P}{\Gamma; \emptyset \vdash_{\mathsf{sat}} t : P}\,\textsc{sat-intro} \qquad \frac{\Gamma \vdash n \Downarrow X}{\Gamma; \emptyset \vdash_{\mathsf{sat}} n : X}\,\textsc{sat-atom}$$

$$\frac{\begin{array}{c}(\bar{n}, \bar{P}) \subseteq \{(n, P) \mid (\Gamma, \Gamma' \vdash n \Downarrow P) \wedge n \;\texttt{uses}\; \Gamma'\} \\ \Gamma, \Gamma'; \bar{x} : \bar{P} \vdash_{\mathsf{sinv}} t : Q_{\mathsf{at}} \qquad \forall x \in \bar{x},\, t \;\texttt{uses}\; x\end{array}}{\Gamma; \Gamma' \vdash_{\mathsf{sat}} \texttt{let}\; \bar{x} \,=\, \bar{n} \;\texttt{in}\; t : Q_{\mathsf{at}}}\,\textsc{sat}$$

$$\frac{x \in \Delta}{x \;\texttt{uses}\; \Delta} \qquad \frac{(\exists n \in \bar{n},\, n \;\texttt{uses}\; \Delta) \vee t \;\texttt{uses}\; \Delta}{\texttt{let}\; \bar{x} \,=\, \bar{n} \;\texttt{in}\; t \;\texttt{uses}\; \Delta}$$

$$\frac{(t_1 \;\texttt{uses}\; \Delta) \vee (t_2 \;\texttt{uses}\; \Delta)}{\delta(x,\, x.t_1,\, x.t_2) \;\texttt{uses}\; \Delta} \qquad \frac{(t \;\texttt{uses}\; \Delta) \vee (u \;\texttt{uses}\; \Delta)}{t\; u \;\texttt{uses}\; \Delta}$$

(other (t $\texttt{uses}$ Δ): simple or-mapping like for $t\ u$)

Figure 6. Cut-free saturating focused intuitionistic logic

In this new judgment, the information that a part of the context is "new", which is available from the invertible judgment $\Gamma; \Gamma' \vdash_{\mathsf{sinv}} t : A$, is retained. The "old" context Γ has already been saturated, and all the positives deducible from it have already been cut – the result of their destruction is somewhere in the context. In the new saturation phase, we must cut all new sums, that were not available before, that is, those that use Γ' in some way. It would not only be inefficient to cut old sums again, it would break *canonicity* (§1.5): with redundant formal variables in the context our algorithm could wrongly believe to have found several distinct proofs.

The right-focusing rules SAT-INTRO and SAT-ATOM behave exactly as FOC-INTRO and FOC-ATOM in the previous focused system. But they can only be used when there is no new context.

When there is a new context to saturate, the judgment must go through the SAT rule – there is no other way to end the proof. The left premise of the rule, corresponding to the definition in SAT, quantifies over all strictly positive neutrals that can be deduced from the old and new contexts combined (Γ, Γ'), but selects those that are "new", in the sense that they use at least one variable coming from the new context fragment Γ'. Then, we simultaneously cut on all those new neutrals, by adding a fresh variable for each of them in the general context, and continuing with an invertible phase: those positives need to be deconstructed for saturation to start again.

The n $\texttt{uses}$ Γ' restriction imposes a unique place at which each cut, each binder may be introduced in the proof term: exactly as soon as it becomes defineable. This enforces canonicity by eliminating redudant proofs that just differ in the place of introduction of a binder, or bind the same value twice. For example, consider the context $\Gamma \overset{\mathsf{def}}{=} (x : X, y : X \to (Y + Y))$, and suppose we are tryind to find all distinct terms of type Y. During the first saturation phase $(\emptyset; \Gamma \vdash_{\mathsf{sat}} ? : Y)$, we would build the neutral term $y\ x$ of type $Y + Y$; it passes the test $y\ x$ $\texttt{uses}$ Γ as it uses both variables of Γ. Then, the invertible phase $\Gamma; z : Y + Y \vdash_{\mathsf{sinv}} ? : Y$ decomposes the goal in two subgoals $\Gamma; z : Y \vdash_{\mathsf{sat}} ? : Y$. Without the n $\texttt{uses}$ Γ' restriction, the SAT rule could cut again on $y\ x$, with would lead, after the next invertible phase, to contexts of the form $\Gamma, z : Y; z' : Y$. But it is wrong to have two distinct variables of type Y here, as there should be only one way to build a Y.

The relation n $\texttt{uses}$ Γ' is defined structurally on proof terms (or, equivalently, their typing derivations). Basically, a term "uses" a context if it uses at least one of its variables; for most terms, it is defined as a big "or" on its subterms. The only subtlety is that the case-split $\delta(x,\, x.t_1,\, x.t_2)$ does not by itself count as a use of the split variable: to be counted as "used", either t_1 or t_2 must use the shadowing variable x.

Finally, the last condition of the SAT rule ($\forall x \in \bar{x}$, t $\texttt{uses}$ x) restricts the saturated variables listed in the $\texttt{let}$-binding to be only those actually used by the term. In terms of proof search, this restriction is applied after the fact: first, cut all positives, then search for all possible subproofs, and finally trim each of them, so that it binds only the positives it uses. This restriction thus does not influence proof search, but it ensures that there always exist finite saturating proofs for inhabited types, by allowing proof search to drop unnecessary bindings instead of saturating them forever. Consider Church numerals on a sum type, $X + Y \to (X + Y \to X + Y) \to X + Y$, there would be no finite saturating proof without this restriction, which would break provability completeness.

Theorem 1 (Canonicity of saturating focused logic)**.** *If we have* $\Gamma; \Delta \vdash_{\mathsf{sinv}} t : A$ *and* $\Gamma; \Delta \vdash_{\mathsf{sinv}} u : A$ *in saturating focused logic with* $t \neq_{\mathsf{icc}} u$*, then* $t \neq_{\beta\eta} u$*.*

Theorem 2 (Computational completeness of saturating focused logic)**.** *If we have* $\emptyset; \Delta \vdash_{\mathsf{inv}} t : A$ *in the non-saturating focused logic, then for some* $u =_{\beta\eta} t$ *we have* $\emptyset; \Delta \vdash_{\mathsf{sinv}} u : A$ *in the saturating focused logic.*

4. Two-Or-More Approximation

A complete presentation of the content of this section, along with complete proofs, is available as a research report (Scherer 2014).

Our algorithm bounds contexts to at most two formal variables at each type. To ensure it correctly predicts unicity (it never claims that there are zero or one programs when two distinct programs exist), we need to prove that if there exists two distinct saturated proofs of a goal A in a given context Γ, then there already exist two distinct proofs of A in the context $\lfloor\Gamma\rfloor_2$, which drops variables from Γ so that no formula occurs more than twice.

We formulate this property in a more general way: instead of talking about the cut-free proofs of the saturating focused logics, we prove a general result about the set of derivations of a typing judgment $\Delta \vdash ? : A$ that have "the same shape", that is, that erase to the same derivation of intuitionistic logic $\lfloor\Delta\rfloor_1 \vdash A$, where $\lfloor\Delta\rfloor_1$ is the set of formulas present in Δ, forgetting multiplicity. This result applies in particular to saturating focused proof terms, (their let-expansion) seen as programs in the unfocused λ-calculus.

We define an explicit syntax for "shapes" S in Figure 7, which are in one-to-one correspondence with (variable-less) natural deduction proofs. It also define the erasure function $\lfloor t\rfloor_1$ from typed λ-terms to typed shapes.

$$
\begin{array}{rlll}
S, T & := & & \text{typed shapes} \\
 & \mid & A, B, C, D & \text{axioms} \\
 & \mid & \lambda A.\, S & \lambda\text{-abstraction} \\
 & \mid & S\; T & \text{application} \\
 & \mid & (S, T) & \text{pair} \\
 & \mid & \pi_i\; S & \text{projection} \\
 & \mid & \sigma_i\; S & \text{sum injection} \\
 & \mid & \delta(S,\, A.T_1,\, B.T_2) & \text{sum destruction}
\end{array}
$$

$$
\lfloor x : A\rfloor_1 \overset{\text{def}}{=} A \qquad \lfloor \lambda x : A.\, t\rfloor_1 \overset{\text{def}}{=} \lambda A.\, \lfloor t\rfloor_1
$$

$$
\lfloor t\; u\rfloor_1 \overset{\text{def}}{=} \lfloor t\rfloor_1\; \lfloor u\rfloor_1 \qquad \lfloor (t, u)\rfloor_1 \overset{\text{def}}{=} (\lfloor t\rfloor_1, \lfloor u\rfloor_1)
$$

$$
\lfloor \pi_i\; t\rfloor_1 \overset{\text{def}}{=} \pi_i\; \lfloor t\rfloor_1 \qquad \lfloor \sigma_i\; t\rfloor_1 \overset{\text{def}}{=} \sigma_i\; \lfloor t\rfloor_1
$$

$$
\lfloor \delta((t : A + B),\, y.u,\, z.r)\rfloor_1 \overset{\text{def}}{=} \delta(\lfloor t\rfloor_1,\, A.\lfloor u\rfloor_1,\, B.\lfloor r\rfloor_1)
$$

Figure 7. Shapes of variable-less natural deduction proofs

The central idea of our approximation result is the use of *counting logics*, that counts the number of λ-terms of different shapes. A counting logic is parametrized over a semiring[1] K; picking the semiring of natural numbers precisely corresponds to counting the number of terms of a given shape, counting in the semiring $\{(0, 1)\}$ corresponds to the variable-less logic (which only expresses inhabitation), and counting in finite semirings of support $\{0, 1, \ldots, M\}$ corresponds to counting proofs with approximative bounded contexts of size at most M.

The counting logic, defined in Figure 8, is parametrized over a semiring $(K, 0_K, 1_K, +_K, \times_K)$. The judgment is of the form $S :: \Phi \vdash_K A : a$, where S is the shape of corresponding logic derivation, Φ is a context mapping formulas to a multiplicity in K, A is the type of the goal being proven, and a is the "output count", a scalar of K.

Let us write $\#S$ the cardinal of a set S and $\lfloor\Delta\rfloor_\#$ for the "cardinal erasure" of the typing context Δ, defined as $\#\{x \mid (x : A) \in \Delta\}$. We can express the relation between counts in the semiring $\mathbb{N}$ and cardinality of typed λ-terms of a given shape:

[1] A semiring $(K, 0_K, 1_K, +_K, \times_K)$ is defined as a two-operation algebraic structure where $(0_K, +_K)$ and $(1_K, \times_K)$ are monoids, $(+_K)$ commutes and distributes over $(\times_K)$ (which may or may not commute), 0_K is a zero/absorbing element for $(\times_K)$, but $(+_K)$ and $(\times_K)$ need not have inverses ($\mathbb{Z}$'s addition is invertible so it is a ring, $\mathbb{N}$ is only a semiring).

$$
(\Phi, \Psi) \overset{\text{def}}{=} A \mapsto (\Phi(A) +_K \Psi(A))
$$

$$
(A : 1) \overset{\text{def}}{=} \begin{cases} A & \mapsto 1_K \\ B \neq A & \mapsto 0_K \end{cases}
$$

$$
\text{COUNT-AXIOM}\quad A :: \Phi \vdash_K A : \Phi(A)
\qquad
\text{COUNT-INTRO-ARR}\quad \frac{S :: \Phi, A : 1 \vdash_K B : a}{\lambda A.\, S :: \Phi \vdash_K A \to B : a}
$$

$$
\text{COUNT-ELIM-ARR}\quad \frac{S_1 :: \Phi \vdash_K A \to B : a_1 \qquad S_2 :: \Phi \vdash_K A : a_2}{S_1\; S_2 :: \Phi \vdash_K B : a_1 \times a_2}
$$

$$
\text{COUNT-INTRO-PAIR}\quad \frac{S_1 :: \Phi \vdash_K A : a_1 \qquad S_2 :: \Phi \vdash_K B : a_2}{(S_1, S_2) :: \Phi \vdash_K A * B : a_1 \times a_2}
$$

$$
\text{COUNT-ELIM-PAIR}\quad \frac{S :: \Phi \vdash_K A_1 * A_2 : a}{\pi_i\; S :: \Phi \vdash_K A_i : a}
\qquad
\text{COUNT-INTRO-SUM}\quad \frac{S :: \Phi \vdash_K A_i : a}{\sigma_i\; S :: \Phi \vdash_K A_1 + A_2 : a}
$$

$$
\text{COUNT-ELIM-SUM}\quad \frac{\begin{array}{c} S :: \Phi \vdash_K A + B : a_1 \\ T_1 :: \Phi, A : 1 \vdash_K C : a_2 \qquad T_2 :: \Phi, B : 1 \vdash_K C : a_3 \end{array}}{\delta(S,\, A.T_1,\, B.T_2) :: \Phi \vdash_K C : a_1 \times a_2 \times a_3}
$$

Figure 8. Counting logic over $(K, 0_K, 1_K, +_K, \times_K)$

Lemma 1. *For any environment Δ, shape S and type A, the following counting judgment is derivable:*

$$
S :: \lfloor\Delta\rfloor_\# \vdash_{\mathbb{N}} A : \#\{t \mid \Delta \vdash t : A \wedge \lfloor t\rfloor_1 = S\}
$$

Note that the counting logic does not have a convincing dynamic semantics – the dynamic semantics of variable-less shapes themselves have been studied in Dowek and Jiang (2011). We only use it as a reasoning tool to count programs.

If $\phi : K \to K'$ map the scalars of one semiring to another, and Φ is a counting context in K, we write $\lfloor\Phi\rfloor_\phi$ its erasure in K' defined by $\lfloor\Phi\rfloor_\phi(A) \overset{\text{def}}{=} \phi(\Phi(A))$. We can then formulate the main result on counting logics:

Theorem 3 (Morphism of derivations)**.** *If $\phi : K \to K'$ is a semiring* morphism *and $S :: \Phi \vdash_K A : a$ is derivable, then $S :: \lfloor\Phi\rfloor_\phi \vdash_{K'} A : \phi(a)$ is also derivable.*

To conclude, we only need to remark that the derivation count is uniquely determined by the multiplicity context.

Lemma 2 (Determinism)**.** *If we have both $S :: \Phi \vdash_K A : a$ and $S :: \Phi \vdash_K A : b$ then $a =_K b$.*

Corollary 1 (Counting approximation)**.** *If ϕ is a semiring* morphism *and $\lfloor\Phi\rfloor_\phi = \lfloor\Psi\rfloor_\phi$ then $S :: \Phi \vdash_K A : a$ and $S :: \Psi \vdash_K A : b$ imply $\phi(a) = \phi(b)$.*

Approximating arbitrary contexts into zero, one or "two-or-more" variables corresponds to the semiring $\bar{2}$ of support $\{0, 1, 2\}$, with commutative semiring operations fully determined by $1 + 1 = 2$, $2 + a = 2$, and $2 \times 2 = 2$. Then, the function $n \mapsto \mathsf{min}(2, n)$ is a semiring morphism from $\mathbb{N}$ to $\bar{2}$, and the corollary above tells us that number of derivations of the judgments $\Delta \vdash A$ and $\lfloor\Delta\rfloor_2 \vdash A$ project to the same value in $\{0, 1, 2\}$. This results extend to any n, as $\{0, 1, \ldots, n\}$ can be similarly given a semiring structure.

5. Search Algorithm

The saturating focused logic corresponds to a computationally complete presentation of the structure of canonical proofs we are interested in. From this presentation it is extremely easy to derive a terminating search algorithm complete for unicity – we moved from a whiteboard description of the saturating rules to a working implementation of the algorithm usable on actual examples in exactly one day of work. The implementation (Scherer and Rémy 2015) is around 700 lines of readable OCaml code.

The central idea to cut the search space while remaining complete for unicity is the *two-or-more* approximation: there is no need to store more than two formal variables of each type, as it suffices to find at least two distinct proofs if they exist – this was proved in the Section 4. We use a *plurality* monad `Plur`, defined in set-theoretic terms as $\mathtt{Plur}(S) \stackrel{\mathsf{def}}{=} 1 + S + S \times S$, representing zero, one or "at least two" distinct elements of the set S. Each typing judgment is reformulated into a search function which takes as input the context(s) of the judgment and its goal, and returns a plurality of proof terms – we search not for *one* proof term, but for (a bounded set of) *all* proof terms. Reversing the usual mapping from variables to types, the contexts map types to pluralities of formal variables.

In the search algorithm, the SINV-END rule does merely pass its new context Γ' to the saturation rules, but it also *trims* it by applying the two-or-more rule: if the old context Γ already has two variables of a given formula N_{at}, drop all variables for N_{at} from Γ'; if it already has one variable, retain at most one variable in Γ'. This corresponds to an eager application of the variable-use restriction of the SAT rule: we have decided to search only for terms that will not use those extraneous variables, hence they are never useful during saturation and we may as well drop them now. This trimming is sound, because it corresponds to an application of the SAT rule that would bind the empty set. Proving that it is complete for unicity is the topic of Section 4.

To effectively implement the saturation rules, a useful tool is a *selection* function (called `select_oblis` in our prototype) which takes a selection predicate on positive or atomic formulas P_{at}, and selects (a plurality of) each negative formula N_{at} from the context that might be the starting point of an elimination judgment of the form $\Gamma \vdash n \Downarrow P_{\mathsf{at}}$, for a P_{at} accepted by the selection predicate. For example, if we want to prove X and there is a formula $Y \to Z * X$, this formula will be selected – although we don't know yet if we will be able to prove Y. For each such P_{at}, it returns a *proof obligation*, that is either a valid derivation of $\Gamma \vdash n \Downarrow P_{\mathsf{at}}$, or a *request*, giving some formula A and expecting a derivation of $\Gamma \vdash ? \Uparrow A$ before returning another proof obligation.

The rule SAT-ATOM ($\Gamma; \emptyset \vdash_{\mathsf{sat}} ? : X$) uses this selection function to select all negatives that could potentially be eliminated into a X, and feeding (pluralities of) answers to the returned proof obligations (by recursively searching for introduction judgments) to obtain (pluralities of) elimination proofs of X.

The rule SAT uses the selection function to find the negatives that could be eliminated in any strictly positive formula and tries to fullfill (pluralities of) proof obligations. This returns a binding context (with a plurality of neutrals for each positive formula), which is filtered a posteriori to keep only the "new" bindings – that use the new context. The new binding are all added to the search environment, and saturating search is called recursively. It returns a plurality of proof terms; each of them results in a proof derivation (where the saturating set is trimmed to retain only the bindings useful to that particular proof term).

Finally, to ensure termination while remaining complete for unicity, we do not search for proofs where a given subgoal occurs strictly more than twice along a given search path. This is easily implemented by threading an extra "memory" argument through each recursive call, which counts the number of identical subgoals below a recursive call and kills the search (by returning the "zero" element of the plurality monad) at two. Note that this does not correspond to memoization in the usual sense, as information is only propagated along a recursive search branch, and never shared between several branches.

This fully describes the algorithm, which is easily derived from the logic. It is effective, and our implementation answers instantly on all the (small) types of polymorphic functions we tried. But it is not designed for efficiency, and in particular saturation duplicates a lot of work (re-computing old values before throwing them away).

In the long version of this article (Scherer and Rémy 2015), we give a presentation of the algorithm as a system of inference rules that is terminating and deterministic. Using the two-or-more counting approximation result (Corollary 1) of the next section, we can prove the correctness of this presentation.

Theorem 4. *Our unicity-deciding algorithm is terminating and complete for unicity.*

The search space restrictions described above are those necessary for *termination*. Many extra optimizations are possible, that can be adapted from the proof search literature – with some care to avoid losing completness for unicity. For example, there is no need to cut on a positive if its atoms do not appear in negative positions (nested to the left of an odd number of times) in the rest of the goal. We did not develop such optimizations, except for two low-hanging fruits we describe below.

Eager Redundancy Elimination Whenever we consider selecting a proof obligation to prove a strict positive during the saturation phase, we can look at the negatives that will be obtained by cutting it. If all those atoms are already present at least twice in the context, this positive is *redundant* and there is no need to cut on it. Dually, before starting a saturation phase, we can look at whether it is already possible to get two distinct neutral proofs of the goal from the current context. In this case it is not necessary to saturate at all.

This optimization is interesting because it significantly reduces the redundancy implied by only filtering of old terms after computing all of them. Indeed, we intuitively expect that most types present in the context are in fact present twice (being unique tends to be the exception rather than the rule in programming situations), and thus would not need to be saturated again. Redundancy of saturation still happens, but only on the "frontier formulas" that are present exactly once.

Subsumption by Memoization One of the techniques necessary to make the inverse method (McLaughlin and Pfenning 2008) competitive is *subsumption*: when a new judgment is derived by forward search, it is added to the set of known results if it is not subsumed by a more general judgment (same goal, smaller context) already known.

In our setting, being careful not to break computational completeness, this rule becomes the following. We use (monotonic) mutable state to grow a memoization table of each proved subgoal, indexed by the right-hand-side formula. Before proving a new subgoal, we look for all already-computed subgoals of the same right-hand-side formula. If one exists with exactly the same context, we return its result. But we also return eagerly if there exists a *larger* context (for inclusion) that returned zero result, or a *smaller* context that returned two-or-more results.

Interestingly, we found out that this optimization becomes unsound in presence of the empty type 0 (which are not yet part of the theory, but are present as an experiment in our implementation). Its equational theory tells us that in an inconsistent cotnext (0 is provable), all proofs are equal. Thus a type may have two inhabitants in a given context, but a larger context that is inconsistent (allows to prove 0) will have a unique inhabitant, breaking monotonicity.

6. Evaluation

In this section, we give some practical examples of code inference scenarios that our current algorithm can solve, and some that it cannot – because the simply-typed theory is too restrictive.

The key to our application is to translate a type using prenex-polymorphism into a simple type using atoms in stead of type variables – this is semantically correct given that bound type variables in System F are handled exactly as simply-typed atoms. The approach, of course, is only a very first step and quickly shows it limits. For example, we cannot work with polymorphic types in the environment (ML programs typically do this, for example when typing a parametrized module, or type-checking under a type-class constraint with polymorphic methods), or first-class polymorphism in function arguments. We also do not handle higher-kinded types – even pure constructors.

6.1 Inferring Polymorphic Library Functions

The Haskell standard library contains a fair number of polymorphic functions with unique types. The following examples have been checked to be uniquely defined by their types:

$$\texttt{fst} : \forall\alpha\beta.\ \alpha * \beta \to \alpha$$
$$\texttt{curry} : \forall\alpha\beta\gamma.\ (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$
$$\texttt{uncurry} : \forall\alpha\beta\gamma.\ (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma$$
$$\texttt{either} : \forall\alpha\beta\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \alpha + \beta \to \gamma$$

When the API gets more complicated, both types and terms become harder to read and uniqueness of inhabitation gets much less obvious. Consider the following operators chosen arbitrarily in the lens (Kmett 2012) library.

```
(<.) :: Indexable i p => (Indexed i s t -> r)
       -> ((a -> b) -> s -> t) -> p a b -> r
(<.>) :: Indexable (i, j) p => (Indexed i s t -> r)
        -> (Indexed j a b -> s -> t) -> p a b -> r
(%@~) :: AnIndexedSetter i s t a b
        -> (i -> a -> b) -> s -> t
non :: Eq a => a -> Iso' (Maybe a) a
```

The type and type-class definitions involved in this library usually contain first-class polymorphism, but the documentation (Kmett 2013) provides equivalent "simple types" to help user understanding. We translated the definitions of `Indexed`, `Indexable` and `Iso` using those simple types. We can then check that the first three operators are unique inhabitants; `non` is not.

6.2 Inferring Module Implementations or Type-Class Instances

The `Arrow` type-class is defined as follows:

```
class Arrow (a : * -> * -> * ) where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

It is self-evident that the arrow type $(\to)$ is an instance of this class, and *no code should have to be written* to justify this: our prototype is able to infer that all those required methods are uniquely determined when the type constructor `a` is instantiated with an arrow type. This also extends to subsequent type-classes, such as `ArrowChoice`.

As most of the difficulty in inferring unique inhabitants lies in sums, we study the "exception monad", that is, for a fixed type X, the functor $\alpha \mapsto X + \alpha$. Our implementation determines that its `Functor` and `Monad` instances are uniquely determined, but that its `Applicative` instance is not.

This is in fact a general result on applicative functors for types that are also monads: there are two distinct ways to prove that a monad is also an applicative functor.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do            ap mf ma = do
  f <- mf                    a <- ma
  a <- ma                    f <- mf
  return (f a)               return (f a)
```

Note that the type of `bind` for the exception monad, namely $\forall\alpha\beta.\ X + \alpha \to (\alpha \to X + \beta) \to X + \beta$, has a sum type thunked under a negative type. It is one typical example of type which cannot be proved unique by the focusing discipline alone, which is correctly recognized unique by our algorithm.

6.3 Non-Applications

Here are two related ideas we wanted to try, but that do not fit in the simply-typed lambda-calculus; the uniqueness algorithm must be extended to richer type systems to handle such applications.

We can check that specific instances of a given type-class are canonically defined, but it would be nice to show as well that some of the operators defined on *any* instance are uniquely defined from the type-class methods – although one would expect this to often fail in practice if the uniqueness checker doesn't understand the equational laws required of valid instances. Unfortunately, this would require uniqueness check with polymorphic types in context (for the polymorphic methods).

Another idea is to verify the coherence property of a set of declared instances by translating instance declarations into terms, and checking uniqueness of the required instance types. In particular, one can model the inheritance of one class upon another using a pair type (`Comp` α as a pair of a value of type `Eq` α and `Comp`-specific methods); and the system can then check that when an instance of `Eq` X and `Comp` X are declared, building `Eq` X directly or projecting it from `Comp` X correspond to $\beta\eta$-equivalent elaboration witnesses. Unfortunately, all but the most simplistic examples require parametrized types and polymorphic values in the environment to be faithfully modelled.

6.4 On Impure Host Programs

The type system in which program search is performed does not need to exactly coincide with the ambiant type system of the host programming language, for which the code-inference feature is proposed – forcing the same type-system would kill any use from a language with non-termination as an effect. Besides doing term search in a pure, terminating fragment of the host language, one could also refine search with type annotations in a richer type system, eg. using dependent types or substructural logic – as long as the found inhabitants can be erased back to host types.

However, this raises the delicate question of, among the unique $\beta\eta$-equivalence class of programs, which candidate to select to be actually injected into the host language. For example, the ordering or repetition of function calls can be observed in a host language passing impure function as arguments, and η-expansion of functions can delay effects. Even in a pure language, η-expanding sums and products may make the code less efficient by re-allocating data. There is a design space here that we have not explored.

7. Related and Future Work

7.1 Related Work

Previous Work on Unique Inhabitation The problem of unique inhabitation for the simply-typed lambda-calculus (without sums) has been formulated by Mints (1981), with early results by Babaev and Soloviev (1982), and later results by Aoto and Ono (1994); Aoto (1999) and Broda and Damas (2005).

These works have obtained several different *sufficient conditions* for a given type to be uniquely inhabited. While these cannot be used as an algorithm to decide unique inhabitation for any type, it reveals fascinating connections between unique inhabitation and proof or term structures. Some sufficient criterions are formulated on the types/formulas themselves, other on terms (a type is uniquely inhabited if it is inhabited by a term of a given structure).

A simple criterion on types given in Aoto and Ono (1994) is that "negatively non-duplicated formulas", that is formulas where each atom occurs at most once in negative position (nested to the left of an odd number of arrows), have at most one inhabitant. This was extended by Broda and Damas (2005) to a notion of "deterministic" formulas, defined using a specialized representation for simply-typed proofs named "proof trees".

Aoto (1999) proposed a criterion based on terms: a type is uniquely inhabited if it "provable without non-prime contraction", that is if it has at least *one* inhabitant (not necessarily cut-free) whose only variables with multiple uses are of atomic type. Recently, Bourreau and Salvati (2011) used game semantics to give an alternative presentation of Aoto's results, and a syntactic characterization of *all* inhabitants of negatively non-duplicated formulas.

Those sufficient conditions suggest deep relations between the static and dynamics semantics of restricted fragments of the lambda-calculus – it is not a coincidence that contraction at non-atomic type is also problematic in definitions of proof equivalence coming from categorial logic (Dosen 2003). However, they give little in the way of a decision procedure for all types – conversely, our decision procedure does not by itself reveal the structure of the types for which it finds unicity.

An indirectly related work is the work on retractions in simple types (A is a retract of B if B can be surjectively mapped into A by a λ-term). Indeed, in a type system with a unit type 1, a given type A is uniquely inhabited if and only if it is a retract of 1. Stirling (2013) proposes an algorithm, inspired by dialogue games, for deciding retraction in the lambda-calculus with arrows and products; but we do not know if this algorithm could be generalized to handle sums. If we remove sums, focusing already provides an algorithm for unique inhabitation.

Counting Inhabitants Broda and Damas (2005) remark that normal inhabitants of simple types can be described by a context-free structure. This suggests, as done in Zaoinc (1995), counting terms by solving a set of polynomial equations. Further references to such "grammatical" approaches to lambda-term enumeration and counting can be found in Dowek and Jiang (2011).

Of particular interest to us was the recent work of Wells and Yakobowski (2004). It is similar to our work both in terms of expected application (program fragment synthesis) and methods, as it uses (a variant of) the focused calculus LJT (Herbelin 1993) to perform proof search. It has sums (disjunctions), but because it only relies on focusing for canonicity it only implements the *weak* notion of η-equivalence for sums: as explained in Section 1.7, it counts an infinite number of inhabitants in presence of a sum thunked under a negative. Their technique to ensure termination of enumeration is very elegant. Over the graph of all possible proof steps in the type system (using multisets as contexts: an infinite search space), they superimpose the graph of all possible non-cyclic proof steps in the logic (using sets as contexts: a finite search space). Termination is obtained, in some sense, by traversing the two in lockstep. We took inspiration from this idea to obtain our termination technique: our bounded multisets can be seen as a generalization of their use of set-contexts.

Non-Classical Theorem Proving and More Canonical Systems Automated theorem proving has motivated fundamental research on more canonical representations of proofs: by reducing the number of redundant representations that are equivalent as programs, one can reduce the search space – although that does not necessarily improve speed, if the finer representation requires more bookkeeping. Most of this work was done first for (first-order) classical logic; efforts porting them to other logics (linear, intuitionistic, modal) were of particular interest, as it often reveals the general idea behind particular techniques, and is sometimes an occasion to reformulate them in terms closer to type theory.

An important brand of work studies connection-based, or matrix-based, proof methods. They have been adapted to non-classical logic as soon as Wallen (1987). It is possible to present connection-based search "uniformly" for many distinct logics (Otten and Kreitz 1996), changing only one logic-specific check to be performed a posteriori on connections (axiom rules) of proof candidates. In intuitionistic setting, that would be a comparison on indices of Kripke Worlds; it is strongly related to *labeled logics* (Galmiche and Méry 2013). On the other hand, matrix-based methods rely on guessing the number of duplications of a formula (contractions) that will be used in a particular proof, and we do not know whether that can be eventually extended to second-order polymorphism – by picking a presentation closer to the original logic, namely focused proofs, we hope for an easier extension.

Some contraction-free calculi have been developed with automated theorem proving for intuitionistic logic in mind. A presentation is given in Dyckhoff (1992) – the idea itself appeared as early as Vorob'ev (1958). The idea is that sums and (positive) products do not need to be deconstructed twice, and thus need not be contracted on the left. For functions, it is actually sufficient for provability to implicitly duplicate the arrow in the argument case of its elimination form ($A \to B$ may have to be used again to build the argument A), and to forget it after the result of application (B) is obtained. More advanced systems typically do case-distinctions on the argument type A to refine this idea, see Dyckhoff (2013) for a recent survey. Unfortunately, such techniques to reduce the search space break computational completeness: they completely remove some programmatic behaviors. Consider the type $\mathtt{Stream}(A, B) \stackrel{\text{def}}{=} A * (A \to A * B)$ of infinite streams of state A and elements B: with this restriction, the next-element function can be applied at most once, hence $\mathtt{Stream}(X, Y) \to Y$ is uniquely inhabited in those contraction-free calculi. (With focusing, only negatives are contracted, and only when picking a focus.)

Focusing was introduced for linear logic (Andreoli 1992), but is adaptable to many other logics. For a reference on focusing for intuitionistic logic, see Liang and Miller (2007). To easily elaborate programs as lambda-terms, we use a natural deduction presentation (instead of the more common sequent-calculus presentation) of focused logic, closely inspired by the work of Brock-Nannestad and Schürmann (2010) on intuitionistic linear logic.

Some of the most promising work on automated theorem proving for intuitionistic logic comes from applying the so-called "Inverse Method" (see Degtyarev and Voronkov (2001) for a classical presentation) to focused logics. The inverse method was ported to linear logic in Chaudhuri and Pfenning (2005), and turned into an efficient implementation of proof search for intuitionistic logic in McLaughlin and Pfenning (2008). It is a "forward" method: to prove a given judgment, start with the instances of axiom rules for all atoms in the judgment, then build all possible valid proofs until the desired judgment is reached – the subformula property, bounding the search space, ensures completeness for propositional logic. Focusing allows important optimization of the method, notably through the idea of "synthetic connectives": invertible or non-invertible phases have to be applied all in one go, and thus form macro-steps that speed up saturation.

In comparison, our own search process alternates forward and backward-search. At a large scale we do a backward-directed proof search, but each non-invertible phase performs saturation, that is a complete forward-search for positives. Note that the search space of those saturation phases is not the subformula space of the main judgment to prove, but the (smaller) subformula space of the current subgoal's context. When saturation is complete, backward goal-directed search restarts, and the invertible phase may grow the context, incrementally widening the search space. (The forward-directed aspects of our system could be made richer by adding positive products and positively-biased atoms; this is not our main point of interest here. Our coarse choice has the good property that, in absence of sum types in the main judgment, our algorithm immediately degrades to simple, standard focused backward search.)

Lollimon (López, Pfenning, Polakow, and Watkins 2005) mixes backward search for negatives and forward search for positives. The logic allows but does not enforce saturation; it is only in the implementation that (provability) saturation is used, and they found it useful for their applications – modelling concurrent systems.

Finally, an important result for canonical proof structures is *maximal multi-focusing* (Miller and Saurin 2007; Chaudhuri, Miller, and Saurin 2008). Multi-focusing refines focusing by introducing the ability to focus on several formulas at once, in parallel, and suggests that, among formulas equivalent modulo valid permutations of inference rules, the "more parallel" one are more canonical. Indeed, *maximal* multi-focused proofs turn out to be equivalent to existing more-canonical proof structures such as linear proof nets (Chaudhuri, Miller, and Saurin 2008) and classical expansion proofs (Chaudhuri, Hetzl, and Miller 2012).

Saturating focused proofs are almost maximal muli-focused proofs according to the definition of Chaudhuri, Miller, and Saurin (2008). The difference is that multi-focusing allow to focus on both variables in the context and the goal in the same time, while our right-focusing rule SAT-INTRO can only be applied sequentially after SAT (which does multi-left-focusing). To recover the exact structure of maximal multi-focusing, one would need to allow SAT to also focus on the right, and use it only when the right choices do not depend on the outcome on saturation of the left (the foci of the same set must be independent), that is when none of the bound variables are used (typically to saturate further) before the start of the next invertible phase. This is a rather artificial restriction from a backward-search perspective. Maximal multi-focusing is more elegant, declarative in this respect, but is less suited to proof search.

Equivalence of Terms in Presence of Sums Ghani (1995) first proved the decidability of equivalence of lambda-terms with sums, using sophisticated rewriting techniques. The two works that followed (Altenkirch, Dybjer, Hofmann, and Scott 2001; Balat, Di Cosmo, and Fiore 2004) used normalization-by-evaluation instead. Finally, Lindley (2007) was inspired by Balat, Di Cosmo, and Fiore (2004) to re-explain equivalence through rewriting. Our idea of "cutting sums as early as possible" was inspired from Lindley (2007), but in retrospect it could be seen in the "restriction (A)" in the normal forms of Balat, Di Cosmo, and Fiore (2004), or directly in the "maximal conversions" of Ghani (1995).

Note that the existence of unknown atoms is an important aspect of our calculus. Without them (starting only from base types 0 and 1), all types would be finitely inhabited. This observation is the basis of the promising unpublished work of Ahmad, Licata, and Harper (2010), also strongly relying on (higher-order) focusing. Finiteness hypotheses also play an important role in Ilik (2014), where they are used to reason on type *isomorphisms* in presence of sums. Our own work does not handle 1 or 0; the latter at least is a notorious source of difficulties for equivalence, but is also seldom necessary in practical programming applications.

Elaboration of Implicits Probably the most visible and the most elegant uses of typed-directed code inference for functional languages are *type-classes* (Wadler and Blott 1989) and *implicits* (Oliveira, Moors, and Odersky 2010). Type classes elaboration is traditionally presented as a satisfiability problem (or constraint solving problem (Stuckey and Sulzmann 2002)) that happens to have operational consequences. Implicits recast the feature as elaboration of a programming *term*, which is closer to our methodology. Type-classes traditionally try (to various degrees of success) to ensure *coherence*, namely that a given elaboration goal always give the same dynamic semantics wherever it happens in the program – often by making instance declarations a toplevel-only construct. Implicits allow a more modular construction of the elaboration environment, but have to resort to priorities to preserve determinism (Oliveira, Schrijvers, Choi, Lee, Yi, and Wadler 2014).

We propose to reformulate the question of determinism or ambiguity by presenting elaboration as a *typing* problem, and proving that the elaborated problems intrinsically have unique inhabitants. This point of view does not by itself solve the difficult questions of which are the good policies to avoid ambiguity, but it provides a more declarative setting to expose a given strategy; for example, priority to the more recently introduced implicit would translate to an explicit weakening construct, removing older candidates at introduction time, or a restricted variable lookup semantics.

(The global coherence issue is elegantly solved, independently of our work, by using a dependent type system where the values that semantically depend on specific elaboration choices (eg., a balanced tree ordered with respect to some specific order) have a type that syntactically depends on the elaboration witness. This approach meshes very well with our view, especially in systems with explicit equality proofs between terms, where features that grow the implicit environment could require proofs from the user that unicity is preserved.)

Smart Completion and Program Synthesis Type-directed program synthesis has seen sophisticated work in the recent years, notably Perelman, Gulwani, Ball, and Grossman (2012), Gvero, Kuncak, Kuraj, and Piskac (2013). Type information is used to fill missing holes in partial expressions given by the users, typically among the many choices proposed by a large software library. Many potential completions are proposed interactively to the user and ordered by various ranking heuristics.

Our uniqueness criterion is much more rigid: restrictive (it has far less potential applications) and principled (there are no heuristics or subjective preferences at play). Complementary, it aims for application in richer type systems, and in *programming constructs* (implicits, etc.) rather than *tooling* with interactive feedback.

Synthesis of glue code interfacing whole modules has been presented as a type-directed search, using type isomorphisms (Aponte and Cosmo 1996) or inhabitation search in combinatory logics with intersection types (Düdder et al. 2014).

We were very interested in the recent Osera and Zdancewic (2015), which generates code from both expected type and input/output examples. The works are complementary: they have interesting proposals for data-structures and algorithm to make term search efficient, while we bring a deeper connection to prooftheoretic methods. They independently discovered the idea that saturation must use the "new" context, in their work it plays the role of an algorithmic improvement they call "relevant term generation".

7.2 Future Work

We hope to be able to extend the uniqueness algorithm to more powerful type systems, such as System F polymorphism or dependent types. Decidability, of course, is not to be expected: deciding uniqueness is at least as hard as deciding inhabitation, and this quickly becomes undecidable for more powerful systems. Yet, we hope that the current saturation approach can be extended to give an effective semi-decision procedures. We will detail below two extensions that we have started looking at, unit and empty types, and parametric polymorphism; and two extensions we have not considered yet, substructural logics and equational reasoning.

Unit and Empty Types As an experiment, we have added a non-formalized support for the unit type 1 and the empty type 0 to our implementation. The unit types poses no difficulties, but we were more surprised to notice that they empty type seems also simple to handle – although we have not proved anything about it for now. We add it as a positive, with the following left-introduction rule (and no right-introduction rule):

$$\frac{\text{SINV-EMPTY}}{\Gamma; \Delta, x : 0 \vdash_{\mathsf{sinv}} \mathtt{absurd}(x) : A}$$

Our saturation algorithm then naturally gives the expected equivalence rule in presence of 0, which is that all programs in a inconsistent context (0 is provable) are equal ($A^0 = 1$): saturation will try to "cut all 0", and thus detect any inconsistency; if one or several proofs of 0 are found, the following invertible phase will always use the SINV-EMPTY rule, and find $\mathtt{absurd}(\_)$ as the unique derivation. For example, while the bind function for the A-translation monad $B \mapsto (B \to A) \to A$ is not unique for arbitrary formulas A, our extended prototype finds a unique bind for the non-delimited continuation monad $B \mapsto B \to 0 \to 0$.

Polymorphism Naively adding parametric polymorphism to the system would suggest the following rules:

$$\text{SINV-POLY}\ \frac{\Gamma; \Delta, \alpha \vdash_{\mathsf{sinv}} t : A}{\Gamma; \Delta \vdash_{\mathsf{sinv}} t : \forall \alpha.\, A} \qquad \text{SELIM-POLY}\ \frac{\Gamma \vdash n \Downarrow \forall \alpha.\, A \qquad \Gamma \vdash B}{\Gamma \vdash n \Downarrow A[B/\alpha]}$$

The invertible introduction rule is trivially added to our algorithm. It generalizes our treatment of atomic types by supporting a bit more than purely prenex polymorphism, as it supports all quantifiers in so-called "positive positions" (to the left of an even number of arrows), such as $1 \to (\forall \alpha.\, \alpha \to \alpha)$ or $((\forall \beta.\, \beta \to \beta) \to X) \to X$. However, saturating the elimination rule SELIM-POLY would a priori require instantiating the polymorphic type with infinitely many instances (there is no clear subformula property anymore). Even naive (and probably incomplete) strategies such as instantiating with all closed formulas of the context lead to non-termination, as for example instantiating the variable α of closed type $1 \to \forall \alpha.\, \alpha$ with the closed type itself leads to an infinite regress of deduced types of the form $1 \to 1 \to 1 \to \dots$.

Another approach would be to provide a left-introduction rule for polymorphism, based on the idea, loosely inspired by higher-order focusing (Zeilberger 2008), that destructing a value is inspecting all possible ways to construct it. For example, performing proof search determines that any possible closed proof of the term $\forall \alpha.\, (X \to Y \to \alpha)$ must have two subgoals, one of type X and another of type Y; and that there are two ways to build a closed proof of $\forall \alpha.\, (X \to \alpha) \to (Y \to \alpha)$, using either a subgoal of type X or of type Y. How far into the traditional territory of parametricity can we go using canonical syntactic proof search only?

Substructural Logics Instead of moving to more polymorphic type systems, one could move to substructural logics. We could expect to refine a type annotation using, for example, linear arrows, to get a unique inhabitant. We observed, however, that linearity is often disappointing in getting "unique enough" types. Take the polymorphic type of mapping on lists, for example: $\forall \alpha \beta.\, (\alpha \to \beta) \to (\mathtt{List}\ \alpha \to \mathtt{List}\ \beta)$. Its inhabitants are the expected map composed with any function that can reorder, duplicate or drop elements from a list. Changing the two inner arrows to be linear gives us the set of functions that may only reorder the mapped elements: still not unique. An idea to get a unique type is to request a mapping from $(\alpha \le \beta)$ to $(\mathtt{List}\ \alpha \le \mathtt{List}\ \beta)$, where the subtyping relation $(\le)$ is seen as a substructural arrow type.

(Dependent types also allow to capture `List.map`, as the unique inhabitant of the dependent induction principle on lists is unique.)

Equational Reasoning We have only considered pure, strongly terminating programs so far. One could hope to find monadic types that uniquely defined transformations of impure programs (e.g. $(\alpha \to \beta) \to \mathtt{M}\ \alpha \to \mathtt{M}\ \beta$). Unfortunately, this approach would not work by simply adding the unit and bind of the monad as formal parameters to the context, because many programs that are only equal up to the monadic laws would be returned by the system. It could be interesting to enrich the search process to also normalize by the monadic laws. In the more general case, can the search process be extended to additional rewrite systems?

7.3 Conclusion

We have presented an algorithm that decides whether a given type of the simply-typed lambda-calculus with sums has a unique inhabitant modulo $\beta\eta$-equivalence; starting from standard focused proof search, the new ingredient is *saturation* which egarly cuts any positive that can be derived from the current context by a focused elimination. Termination is obtained through a context approximation result, remembering one or "two-or-more" variables of each type.

This is a foundational approach to questions of code inference, yet preliminary studies suggest that there are already a few potential applications, to be improved with future support for richer systems.

Of course, guessing a program from its type is not necessarily beneficial if the type is as long to write (or harder to read) than the program itself. We see code and type inference as mutually-beneficial features, allowing the programmer to express intent in part through the term language, in part through the type language, playing on which has developped the more expressive definitions or abstractions for the task at hand.

Acknowledgments

We are grateful to Adrien Guatto and anonymous reviewers for their helpful feedback.

References

Arbob Ahmad, Daniel R. Licata, and Robert Harper. Deciding coproduct equality with focusing. Online draft, 2010.

Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, 2001.

Jean-Marc Andreoli. Logic Programming with Focusing Proof in Linear Logic. *Journal of Logic and Computation*, 1992.

Takahito Aoto. Uniqueness of normal proofs in implicational intuitionistic logic. *Journal of Logic, Language and Information*, 1999.

Takahito Aoto and Hiroakira Ono. Non-Uniqueness of Normal Proofs for Minimal Formulas in Implication-Conjunction Fragment of BCK. *Bulletin of the Section of Logic*, 1994.

Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *PLILP*, 1996.

Ali Babaev and Sergei Soloviev. A coherence theorem for canonical morphisms in cartesian closed categories. *Journal of Soviet Mathematics*, 1982.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 2004.

Pierre Bourreau and Sylvain Salvati. Game semantics and uniqueness of type inhabitance in the simply-typed λ-calculus. In *TLCA*, 2011.

Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In *LPAR-17*, 2010.

Sabine Broda and Luís Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 2005.

Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In *CSL*, 2005.

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *IFIP TCS*, 2008.

Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A Systematic Approach to Canonicity in the Classical Sequent Calculus. In *CSL*, 2012.

Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In *IFIP TCS*, 2010.

Anatoli Degtyarev and Andrei Voronkov. Introduction to the inverse method. In *Handbook of Automated Reasoning*. 2001.

Kosta Dosen. Identity of proofs based on normalization and generality. *Bulletin of Symbolic Logic*, 2003.

Gilles Dowek and Ying Jiang. On the expressive power of schemes. *Inf. Comput.*, 2011.

Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, 2014.

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 1992.

Roy Dyckhoff. Intuitionistic decision procedures since gentzen, 2013. Talk notes.

Didier Galmiche and Daniel Méry. A connection-based characterization of bi-intuitionistic validity. *J. Autom. Reasoning*, 2013.

Neil Ghani. Beta-eta equality for coproducts. In *TLCA*, 1995.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.

Hugo Herbelin. A lambda-calculus structure isomorphic to gentzen-style sequent calculus structure. In *CSL*, 1993.

Danko Ilik. Axioms and decidability for type isomorphism in the presence of sums. *CoRR*, 2014. URL `http://arxiv.org/abs/1401.2567`.

Edward Kmett. Lens, 2012. URL `https://github.com/ekmett/lens`.

Edward Kmett. Lens wiki – types, 2013. URL `https://github.com/ekmett/lens/wiki/Types`.

Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL*, 2009.

Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. *CoRR*, 2007. URL `http://arxiv.org/abs/0708.2252`.

Sam Lindley. Extensional rewriting with sums. In *TLCA*, 2007.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP*, 2005.

Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In *LPAR*, 2008.

Dale Miller and Alexis Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In *CSL*, 2007.

Grigori Mints. Closed categories and the theory of proofs. *Journal of Soviet Mathematics*, 1981.

Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi, and Philip Wadler. The implicit calculus: A new foundation for generic programming. 2014.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

Jens Otten and Christoph Kreitz. A uniform proof procedure for classical and non-classical logics. In *KI Advances in Artificial Intelligence*, 1996.

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.

Gabriel Scherer. Mining opportunities for unique inhabitants in dependent programs, 2013.

Gabriel Scherer. 2-or-more approximation for intuitionistic logic. 2014.

Gabriel Scherer and Didier Rémy, 2015. URL `http://gallium.inria.fr/~scherer/research/unique_inhabitants/`.

Colin Stirling. Proof systems for retracts in simply typed lambda calculus. In *Automata, Languages, and Programming - ICALP*, 2013.

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP*, 2002.

Nikolay Vorob'ev. A new algorithm of derivability in a constructive calculus of statements. In *Problems of the constructive direction in mathematics*, 1958.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.

Lincoln A. Wallen. Automated proof search in non-classical logics: Efficient matrix proof methods for modal and intuitionistic logic, 1987.

Joe B. Wells and Boris Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, 2004.

Marek Zaoinc. Fixpoint technique for counting terms in typed lambda-calculus. Technical report, State University of New York, 1995.

Noam Zeilberger. Focusing and higher-order abstract syntax. In *POPL*, 2008.