THE SAFE LAMBDA CALCULUS

VERSION OF April 22, 2008

WILLIAM BLUM AND C.-H. LUKE ONG

Oxford University Computing Laboratory – School of Informatics, University of Edinburgh, UK
e-mail address: william.blum@comlab.ox.ac.uk

Oxford University Computing Laboratory, Oxford, UK
e-mail address: luke.ong@comlab.ox.ac.uk

Abstract. Safety is a syntactic condition of higher-order grammars that constrains occurrences of variables in the production rules according to their type-theoretic order. In this paper, we introduce the *safe lambda calculus*, which is obtained by transposing (and generalizing) the safety condition to the setting of the simply-typed lambda calculus. In contrast to the original definition of safety, our calculus does not constrain types (to be homogeneous). We show that in the safe lambda calculus, there is no need to rename bound variables when performing substitution, as variable capture is guaranteed not to happen. We also propose an adequate notion of β-reduction that preserves safety. In the same vein as Schwichtenberg's 1976 characterization of the simply-typed lambda calculus, we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials; thus conditional is not definable. We also give a characterization of representable word functions. We then study the complexity of deciding beta-eta equality of two safe simply-typed terms and show that this problem is PSPACE-hard. Finally we give a game-semantic analysis of safety: We show that safe terms are denoted by *P-incrementally justified strategies*. Consequently pointers in the game semantics of safe λ-terms are only necessary from order 4 onwards.

Introduction

Background. The *safety condition* was introduced by Knapik, Niwiński and Urzyczyn at FoSSaCS 2002 [19] in a seminal study of the algorithmics of infinite trees generated by higher-order grammars. The idea, however, goes back some twenty years to Damm [10] who introduced an essentially equivalent[1] syntactic restriction (for generators of word languages) in the form of *derived types*. A higher-order grammar (that is assumed to be *homogeneously typed*) is said to be *safe* if it obeys certain syntactic conditions that constrain

2000 ACM Subject Classification: F.3.2, F.4.1.

Key words and phrases: lambda calculus, higher-order recursion scheme, safety restriction, game semantics.

Some of the results presented here were first published in TLCA proceedings [8].

[1]See de Miranda's thesis [12] for a proof.

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

the occurrences of variables in the production (or rewrite) rules according to their type-theoretic order. Though the formal definition of safety is somewhat intricate, the condition itself is manifestly important. As we survey in the following, higher-order *safe* grammars capture fundamental structures in computation, offer clear algorithmic advantages, and lend themselves to a number of compelling characterizations:

- *Word languages.* Damm and Goerdt [11] have shown that the word languages generated by order-n *safe* grammars form an infinite hierarchy as n varies over the natural numbers. The hierarchy gives an attractive classification of the semi-decidable languages: Levels 0, 1 and 2 of the hierarchy are respectively the regular, context-free, and indexed languages (in the sense of Aho [5]), although little is known about higher orders.

  Remarkably, for generating word languages, order-n *safe* grammars are equivalent to order-n pushdown automata [11], which are in turn equivalent to order-n indexed grammars [24, 25].
- *Trees.* Knapik *et al.* have shown that the Monadic Second Order (MSO) theories of trees generated by *safe* (deterministic) grammars of every finite order are decidable[2].

  They have also generalized the equi-expressivity result due to Damm and Goerdt [11] to an equivalence result with respect to generating trees: A ranked tree is generated by an order-n *safe* grammar if and only if it is generated by an order-n pushdown automaton.
- *Graphs.* Caucal [9] has shown that the MSO theories of graphs generated[3] by *safe* grammars of every finite order are decidable. In a recent preprint [17], however, Hague *et al.* have shown that the MSO theories of graphs generated by order-n *unsafe* grammars are undecidable, but deciding their modal mu-calculus theories is n-EXPTIME complete.

Overview. In this paper, we aim to understand the safety condition in the setting of the lambda calculus. Our first task is to transpose it to the lambda calculus and pin it down as an appropriate sub-system of the simply-typed theory. A first version of the *safe lambda calculus* has appeared in an unpublished technical report [4]. Here we propose a more general and cleaner version where terms are no longer required to be homogeneously typed (see Section 1 for a definition). The formation rules of the calculus are designed to maintain a simple invariant: Variables that occur free in a safe λ-term have orders no smaller than that of the term itself. We can now explain the sense in which the safe lambda calculus is safe by establishing its salient property: No variable capture can ever occur when substituting a safe term into another. In other words, in the safe lambda calculus, it is *safe* to use capture-*permitting* substitution when performing β-reduction.

There is no need for new names when computing β-reductions of safe λ-terms, because one can safely "reuse" variable names in the input term. Safe lambda calculus is thus cheaper to compute in this naïve sense. Intuitively one would expect the safety constraint to lower the expressivity of the simply-typed lambda calculus. Our next contribution is to give a precise measure of the expressivity deficit of the safe lambda calculus. An old result

[2]It has recently been shown [30] that trees generated by *unsafe* deterministic grammars (of every finite order) also have decidable MSO theories. More precisely, the MSO theory of trees generated by order-n recursion schemes is n-EXPTIME complete.

[3]These are precisely the configuration graphs of higher-order pushdown systems.

of Schwichtenberg [34] says that the numeric functions representable in the simply-typed lambda calculus are exactly the multivariate polynomials *extended with the conditional function*. In the same vein, we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials.

Our last contribution is to give a game-semantic account of the safe lambda calculus. Using a correspondence result relating the game semantics of a λ-term M to a set of *traversals* [30] over a certain abstract syntax tree of the η-long form of M (called *computation tree*), we show that safe terms are denoted by *P-incrementally justified strategies*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and the pointers associated to the O-moves therein: Specifically, a P-question always points to the last pending O-question (in the P-view) of a greater order. Consequently pointers in the game semantics of safe λ-terms are only necessary from order 4 onwards. Finally we prove that a β-normal λ-term is *safe* if and only if its strategy denotation is (innocent and) *P-incrementally justified*.

1. The safe lambda calculus

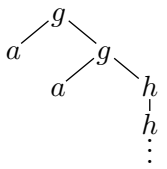
Higher-order safe grammars. We first present the safety restriction as it was originally defined [19]. We consider simple types generated by the grammar $A ::= o \mid A \to A$. By convention, $\to$ associates to the right. Thus every type can be written as $A_1 \to \cdots \to A_n \to o$, which we shall abbreviate to $(A_1, \cdots, A_n, o)$ (in case $n = 0$, we identify (o) with o). The *order* of a type is given by $\mathsf{ord}(o) = 0$ and $\mathsf{ord}(A \to B) = \max(\mathsf{ord}(A) + 1, \mathsf{ord}(B))$. We assume an infinite set of typed variables. The order of a typed term or symbol is defined to be the order of its type.

A (higher-order) ***grammar*** is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, where Σ is a ranked alphabet (in the sense that each symbol $f \in \Sigma$ has an arity $ar(f) \geq 0$) of *terminals*[4]; $\mathcal{N}$ is a finite set of typed *non-terminals*; S is a distinguished ground-type symbol of $\mathcal{N}$, called the start symbol; $\mathcal{R}$ is a finite set of production (or rewrite) rules, one for each non-terminal $F : (A_1, \ldots, A_n, o) \in \mathcal{N}$, of the form $F z_1 \ldots z_m \to e$ where each z_i (called *parameter*) is a variable of type A_i and e is an applicative term of type o generated from the typed symbols in $\Sigma \cup \mathcal{N} \cup \{z_1, \ldots, z_m\}$. We say that the grammar is *order-n* just in case the order of the highest-order non-terminal is n.

The ***tree generated by a recursion scheme*** G is a possibly infinite applicative term, but viewed as a Σ-labelled tree; it is *constructed from the terminals in* Σ, and is obtained by unfolding the rewrite rules of G *ad infinitum*, replacing formal by actual parameters each time, starting from the start symbol S. See e.g. [19] for a formal definition.

Example 1.1. Let G be the following order-2 recursion scheme:

$$\begin{array}{rcl} S & \to & H\, a \\ H\, z^o & \to & F\, (g\, z) \\ F\, \phi^{(o,o)} & \to & \phi\, (\phi\, (F\, h)) \end{array}$$



where the arities of the terminals g, h, a are $2, 1, 0$ respectively. The tree generated by G is defined by the infinite term $g\, a\, (g\, a\, (h\, (h\, (h\, \cdots))))$.

[4]Each $f \in \Sigma$ of arity $r \geq 0$ is assumed to have type $(\underbrace{o, \cdots, o}_{r}, o)$.

A type $(A_1, \cdots, A_n, o)$ is said to be ***homogeneous*** if $\mathsf{ord}(A_1) \geq \mathsf{ord}(A_2) \geq \cdots \geq \mathsf{ord}(A_n)$, and each $A_1, \ldots, A_n$ is homogeneous [19]. We reproduce the following definition from [19].

Definition 1.2 (Safe grammar)**.** (All types are assumed to be homogeneous.) A term of order $k > 0$ is *unsafe* if it contains an occurrence of a parameter of order strictly less than k, otherwise the term is *safe*. An occurrence of an unsafe term t as a subexpression of a term t' is *safe* if it is in the context $\cdots(ts)\cdots$, otherwise the occurrence is *unsafe*. A grammar is ***safe*** if no unsafe term has an unsafe occurrence at a right-hand side of any production.

Example 1.3. (i) Take $H : ((o,o),o)$ and $f : (o,o,o)$; the following rewrite rules are unsafe (in each case we underline the unsafe subterm that occurs unsafely):

$$\begin{array}{rcl} G^{(o,o)}\, x & \to & H\, \underline{(f\, x)} \\ F^{((o,o),o,o,o)}\, z\, x\, y & \to & f\, (F\, \underline{(F\, z\, y)}\, y\, (z\, x))\, x \end{array}$$

(ii) The order-2 grammar defined in Example 1.1 is unsafe.

Safety adapted to the lambda calculus. We assume a set Ξ of higher-order constants. We use sequents of the form $\Gamma \vdash^{\Xi}_{\$} M : A$ to represent term-in-context where Γ is the context and A is the type of M. For convenience, we shall omit the superscript from $\vdash^{\Xi}_{\mathsf{s}}$ whenever the set of constants Ξ is clear from the context. The subscript in $\vdash^{\Xi}_{\$}$ specifies which type system we are using to form the judgment: we use the subscript 'st' to refer to the traditional system of rules of the Church-style simply-typed lambda calculus augmented with constants from Ξ. In the following we will use new subscripts for each type system that we introduce. For simplicity we write $(A_1, \cdots, A_n, B)$ to mean $A_1 \to \cdots \to A_n \to B$, where B is not necessarily ground.

Definition 1.4. (i) The ***safe lambda calculus*** is a sub-system of the simply-typed lambda calculus. It is defined as the set of judgments of the form $\Gamma \vdash_{\mathsf{s}} M : A$ that are derivable from the following Church-style system of rules:

$$(\mathsf{var})\ \frac{}{x : A \vdash_{\mathsf{s}} x : A} \qquad (\mathsf{const})\ \frac{}{\vdash_{\mathsf{s}} f : A}\ f \in \Xi \qquad (\mathsf{wk})\ \frac{\Gamma \vdash_{\mathsf{s}} s : A}{\Delta \vdash_{\mathsf{s}} s : A}\ \Gamma \subset \Delta$$

$$(\mathsf{app}_{\mathsf{as}})\ \frac{\Gamma \vdash_{\mathsf{as}} s : A \to B \quad \Gamma \vdash_{\mathsf{s}} t : A}{\Gamma \vdash_{\mathsf{as}} s\ t : B} \qquad (\delta)\ \frac{\Gamma \vdash_{\mathsf{s}} s : A}{\Gamma \vdash_{\mathsf{as}} s : A}$$

$$(\mathsf{app})\ \frac{\Gamma \vdash_{\mathsf{as}} s : A \to B \quad \Gamma \vdash_{\mathsf{s}} t : A}{\Gamma \vdash_{\mathsf{s}} s\ t : B}\ \mathsf{ord}(B) \leq \mathsf{ord}(\Gamma)$$

$$(\mathsf{abs})\ \frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathsf{as}} s : B}{\Gamma \vdash_{\mathsf{s}} \lambda x_1 \ldots x_n.s : (A_1, \ldots, A_n, B)}\ \mathsf{ord}(A_1, \ldots, A_n, B) \leq \mathsf{ord}(\Gamma)$$

where $\mathsf{ord}(\Gamma)$ denotes the set $\{\mathsf{ord}(y) : y \in \Gamma\}$ and "$c \leq S$" means that c is a lower-bound of the set S. The subscripts in $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{as}}$ stand for "safe" and "almost safe" respectively.
(ii) The sub-system that is defined by the same rules in (i), such that all types that occur in them are homogeneous, is called the ***homogeneous safe lambda calculus***.

An equivalent notion of homogeneous safe lambda calculus is given in de Miranda's thesis [12].

The safe lambda calculus deviates from the standard definition of the simply-typed lambda calculus in a number of ways. First the rule (abs) can abstract several variables at once. (Of course this feature alone does not alter expressivity.) Crucially, the side conditions in the application rule and abstraction rule require the variables in the typing context to have orders no smaller than that of the term being formed. We do not impose any constraint on types. In particular, type-homogeneity, which was an assumption of the original definition of safe grammars [19], is not required here. Another difference is that we allow Ξ-constants to have arbitrary higher-order types.

Example 1.5 (Kierstead terms)**.** Consider the terms $M_1 = \lambda f.f(\lambda x.f(\lambda y.y))$ and $M_2 = \lambda f.f(\lambda x.f(\lambda y.x))$ where $x, y : o$ and $f : ((o, o), o)$. The term M_2 is not safe because in the subterm $f(\lambda y.x)$, the free variable x has order 0 which is smaller than $\mathsf{ord}(\lambda y.x) = 1$. On the other hand, M_1 is safe.

It is easy to see that valid typing judgements of the safe lambda calculus satisfy the following simple invariant:

Lemma 1.6. *If $\Gamma \vdash_{\mathsf{s}} M : A$ then every variable in Γ occurring free in M has order at least $ord(M)$.*

Moreover we have the following immediate result from the definition of the rules ($\mathsf{app}_{\mathsf{as}}$) and (app):

Lemma 1.7. *An almost safe term is an applicative term built out of safe terms i.e. it has the form $N_1 \dots N_m$ for some $m \geq 1$ where N_1 is not an application and for every $1 \leq i \leq m$, N_i is safe.*

When restricted to the homogeneously-typed sub-system, the safe lambda calculus captures the original notion of safety due to Knapik *et al.* in the context of higher-order grammars:

Proposition 1.8. *Let $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ be a grammar and let e be an applicative term generated from the symbols in $\mathcal{N} \cup \Sigma \cup \{ z_1^{A_1}, \cdots, z_m^{A_m} \}$. A rule $F z_1 \dots z_m \to e$ in $\mathcal{R}$ is safe (in the original sense of Knapik* et al.*) if and only if $z_1 : A_1, \cdots, z_m : A_m \vdash_{\mathsf{s}}^{\Sigma \cup \mathcal{N}} e : o$ is a valid typing judgement of the* homogeneous *safe lambda calculus.*

See [12] for a proof.

In what sense is the safe lambda calculus safe? It is an elementary fact that when performing β-reduction in the lambda calculus, one must use capture-*avoiding* substitution, which is standardly implemented by renaming bound variables afresh upon each substitution. In the safe lambda calculus, however, variable capture can never happen (as the following lemma shows). Substitution can therefore be implemented simply by capture-*permitting* replacement, without any need for variable renaming. In the following, we write $M\{N/x\}$ to denote the capture-*permitting* substitution[5] of N for x in M.

Lemma 1.9 (No variable capture)**.** *There is no variable capture when performing capture-permitting substitution of N for x in M provided that $\Gamma, x : B \vdash_{\mathsf{s}} M : A$ and $\Gamma \vdash_{\mathsf{s}} N : B$ are valid judgments of the safe lambda calculus.*

[5]This substitution is done by textually replacing all free occurrences of x in M by N without performing variable renaming. In particular for the abstraction case we have $(\lambda y_1 \dots y_n.M)\{N/x\} = \lambda y_1 \dots y_n.M\{N/x\}$ when $x \notin \{y_1 \dots y_n\}$.

Proof. We proceed by structural induction on M. The variable, constant, weakening and application cases are trivial. For the abstraction case, suppose $M = \lambda\overline{y}.R$ where $\overline{y} = y_1 \dots y_p$. If $x \in \overline{y}$ then $M\{N/x\} = M$ and there is no variable capture.

Otherwise, $x \notin \overline{y}$. By Lemma 1.7 R is of the form $M_1 \dots M_m$ for some $m \geq 1$ where M_1 is not an application and for every $1 \leq i \leq m$, M_i is safe. Thus we have $M\{N/x\} = \lambda\overline{y}.M_1\{N/x\} \dots M_m\{N/x\}$. Let $i \in \{1..m\}$. By the induction hypothesis there is no variable capture in $M_i\{N/x\}$. Thus variable capture can only happen if the following two conditions are met: x occurs freely in M_i, and some variable y_i for $1 \leq i \leq p$ occurs freely in N. By Lemma 1.6, the latter condition implies $\mathsf{ord}(y_i) \geq \mathsf{ord}(N) = \mathsf{ord}(x)$ and since $x \notin \overline{y}$, the former condition implies that x occurs freely in the safe term $\lambda\overline{y}.R$ thus by Lemma 1.6 we have $\mathsf{ord}(x) \geq \mathsf{ord}(\lambda\overline{y}.R) \geq 1 + \mathsf{ord}(y_i) > \mathsf{ord}(y_i)$ which gives a contradiction. □

Remark 1.10. A version of the No-variable-capture Lemma also holds in safe grammars, as is implicit in (for example Lemma 3.2 of) the original paper [19].

Example 1.11. In order to contract the β-redex in the term

$$f : (o,o,o), x : o \vdash_{\mathsf{st}} (\lambda\varphi^{(o,o)}x^o.\varphi\, x)(\underline{f\, x}) : (o,o)$$

one should rename the bound variable x to a fresh name to prevent the capture of the free occurrence of x in the underlined term during substitution. Consequently, by the previous lemma, the term is not safe (because $\mathsf{ord}(x) = 0 < 1 = \mathsf{ord}(fx)$).

Note that λ-terms that 'satisfy' the No-variable-capture Lemma are not necessarily safe. For instance the β-redex in $\lambda y^o z^o.(\lambda x^o.y)z$ can be contracted using capture-permitting substitution, even though the term is not safe.

Safe beta reduction. From now on we will use the standard notation $M\,[N/x]$ to denote the substitution of N for x in M. It is understood that, provided that M and N are safe, this substitution is capture-permitting.

Lemma 1.12 (Substitution preserves safety)**.** *If* $\Gamma, x : B \vdash_{\mathsf{s}} M : A$ *and* $\Gamma \vdash_{\mathsf{s}} N : B$ *then* $\Gamma \vdash_{\mathsf{s}} M[N/x] : A$.

This is proved by an easy induction on the structure of the safe term M.

It is desirable to have an appropriate notion of reduction for our calculus. However the standard β-reduction rule is not adequate. Indeed, safety is not preserved by β-reduction as the following example shows. Suppose that $w, x, y, z : o$ and $f : (o,o,o) \in \Sigma$ then the safe term $(\lambda xy.fxy)zw$ β-reduces to $(\underline{\lambda y.fzy})w$, which is unsafe since the underlined order-1 subterm contains a free occurrence of the ground-type z. However if we perform one more reduction we obtain the safe term fzw. This suggests simultaneous contraction of "consecutive" β-redexes. In order to define this notion of reduction we first introduce the corresponding notion of redex.

In the simply-typed lambda calculus a redex is a term of the form $(\lambda x.M)N$. In the safe lambda calculus, a redex is a succession of several standard redexes:

Definition 1.13. Let $l \geq 1$ and $n \geq 1$. We use the abbreviations $\overline{x}$ and $\overline{x} : \overline{A}$ for $x_1 \dots x_n$ and $x_1 : A_1, \dots, x_n : A_n$ respectively. A ***safe redex*** is a safe term of the form $(\lambda\overline{x}.M)N_1 \dots N_l$ such that the variables $\overline{x}$ are abstracted altogether by one application of the (abs) rule.

For instance, in the case $n < l$, a safe redex has a derivation tree of the following form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma', \overline{x} : \overline{A} \vdash_{\mathsf{s}} M : (A_{n+1}, \dots, A_l, B)}\,(\mathsf{abs})}{\Gamma' \vdash_{\mathsf{s}} \lambda\overline{x}.M : (A_1, \dots, A_l, B)}\,(\mathsf{wk})}{\Gamma \vdash_{\mathsf{s}} \lambda\overline{x}.M : (A_1, \dots, A_l, B)}\,(\delta)}{\Gamma \vdash_{\mathsf{as}} \lambda\overline{x}.M : (A_1, \dots, A_l, B)} \qquad \dfrac{\dots}{\Gamma \vdash_{\mathsf{s}} N_1 : A_1}}{\Gamma \vdash_{\mathsf{as}} (\lambda\overline{x}.M)N_1 : (A_2, \dots A_l, B)}\,(\mathsf{app}_{\mathsf{as}})$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash_{\mathsf{as}} (\lambda\overline{x}.M)N_1 \dots N_{l-1} : (A_l, B)}\,(\mathsf{app}_{\mathsf{as}}) \qquad \dfrac{\dots}{\Gamma \vdash_{\mathsf{s}} N_l : A_l}}{\Gamma \vdash_{\mathsf{s}} (\lambda\overline{x}.M)N_1 \dots N_l : B}\,(\mathsf{app})$$

We can now define a notion of reduction for safe terms.

Definition 1.14. We use the abbreviations $\overline{x} = x_1 \dots x_n$, $\overline{N} = N_1 \dots N_l$. The relation β_s (when viewed as a function) is defined on the set of safe redexes as follows:

$$\begin{aligned} \beta_s &= \{\ (\lambda\overline{x}.M)N_1 \dots N_l \mapsto \lambda x_{l+1} \dots x_n.M\left[\overline{N}/x_1 \dots x_l\right], \text{ for } n > l\} \\ &\cup \{\ (\lambda\overline{x}.M)N_1 \dots N_l \mapsto M\left[N_1 \dots N_n/\overline{x}\right] N_{n+1} \dots N_l, \text{ for } n \leq l\}\ . \end{aligned}$$

where $M\left[R_1 \dots R_k/z_1 \dots z_k\right]$ denotes the simultaneous substitution in M of R_1,...,R_k for $z_1, \dots, z_k$. The ***safe β-reduction***, written $\rightarrow_{\beta_s}$, is the compatible closure of the relation β_s with respect to the formation rules of the safe lambda calculus.

Remark: The safe β-reduction is a multi-step β-reduction *i.e.* it is a subset of the transitive closure of $\rightarrow_\beta$.

Lemma 1.15 (β_s-reduction preserves safety)**.** *If* $\Gamma \vdash_{\mathsf{s}} s : A$ *and* $s \rightarrow_{\beta_s} t$ *then* $\Gamma \vdash_{\mathsf{s}} t : A$.

Proof. It suffices to show that the relation β_s preserves safety. Suppose that $s\ \beta_s\ t$ where s is the safe-redex $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ with $x_1 : B_1, \dots, x_n : B_n$ and M of type C. W.l.o.g we can assume that the last rule used to form the term s is (app) (and not the weakening rule (wk), thus we have $\Gamma = fv(s)$, and since s is safe, Lemma 1.6 gives us $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$.

Suppose $n > l$ then $A = (B_{l+1}, \dots, B_n, C)$. By Lemma 1.12 we can form the safe term $\Gamma, x_{l+1} : B_{l+1}, \dots x_n : B_n \vdash_{\mathsf{s}} M\left[\overline{N}/x_1 \dots x_l\right] : C$. And since we have $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$, we can use the (abs) rule to form the safe term $\Gamma \vdash_{\mathsf{s}} \lambda x_{l+1} \dots x_n.M\left[\overline{N}/x_1 \dots x_l\right] \equiv t : A$.

Suppose $n \leq l$. The Substitution Lemma gives $\Gamma \vdash_{\mathsf{s}} M\left[N_1 \dots N_n/\overline{x}\right] : C$. If $n < l$ then we further apply the rule $(\mathsf{app}_{\mathsf{as}})$ $l - n - 1$ times followed by one application of (app). This gives us the sequent $\Gamma \vdash_{\mathsf{s}} t : A$. □

Eta-long expansion. The η-long normal form (or simply η-long form) of a term is obtained by hereditarily η-expanding the body of every lambda abstraction as well as every subterm occurring in an *operand position* (*i.e.* occurring as the second argument of some occurrence of the binary application operator). Formally the ***η-long form***, written $\lceil t \rceil$, of a term $t : (A_1, \dots, A_n, o)$ with $n \geq 0$ is defined by cases according to the syntactic shape of t:

$$\begin{aligned} \lceil \lambda x.s \rceil &= \lambda x.\lceil s \rceil \\ \lceil x s_1 \dots s_m \rceil &= \lambda\overline{\varphi}.x\lceil s_1 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil \\ \lceil (\lambda x.s) s_1 \dots s_p \rceil &= \lambda\overline{\varphi}.(\lambda x.\lceil s \rceil)\lceil s_1 \rceil \dots \lceil s_p \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil \end{aligned}$$

where $m \geq 0$, $p \geq 1$, x is either a variable or constant, $\overline{\varphi} = \varphi_1 \dots \varphi_n$ and each $\varphi_i : A_i$ is a fresh variable.

Remark 1.16. This transformation does not introduce new redexes therefore the η-long normal form of a β-normal term is also β-normal.

Let us introduce a new typing system:

Definition 1.17. We define the set of ***long-safe terms*** by induction over the following system of rules:

$$(\mathsf{var_l})\ \frac{}{x : A \vdash_{\mathsf{l}} x : A} \qquad (\mathsf{const_l})\ \frac{}{\vdash_{\mathsf{l}} f : A}\quad f \in \Xi \qquad (\mathsf{wk_l})\ \frac{\Gamma \vdash_{\mathsf{l}} s : A}{\Delta \vdash_{\mathsf{l}} s : A}\quad \Gamma \subset \Delta$$

$$(\mathsf{app_l})\ \frac{\Gamma \vdash_{\mathsf{l}} s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_{\mathsf{l}} t_1 : A_1 \quad \dots \quad \Gamma \vdash_{\mathsf{l}} t_n : A_n}{\Gamma \vdash_{\mathsf{l}} st_1 \dots t_n : B}\quad \mathsf{ord}(B) \leq \mathsf{ord}(\Gamma)$$

$$(\mathsf{abs_l})\ \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_{\mathsf{l}} s : B}{\Gamma \vdash_{\mathsf{l}} \lambda x_1 \dots x_n.s : (A_1, \dots, A_n, B)}\quad \mathsf{ord}(A_1, \dots, A_n, B) \leq \mathsf{ord}(\Gamma)$$

The subscript in $\vdash_{\mathsf{l}}$ stands for "long".

The terminology "long-safe" is deliberately suggestive of a forthcoming lemma. Note that long-safe terms are not necessarily in η-long normal form.

Observe that the system of rules from Def. 1.17 is a sub-system of the typing system of Def. 1.4 where the application rule is restricted the same way as the abstraction rule *i.e.* it can perform multiple applications at once provided that all the variables in the context of the resulting term have order greater than the order of the term itself. Thus we clearly have:

Lemma 1.18. *If a term is long-safe then it is safe.*

In general, long-safety is not preserved by η-expansion: for instance we have $\vdash_{\mathsf{l}} \lambda y^o z^o.y : (o, o, o)$ but $\not\vdash_{\mathsf{l}} \lambda x^o.(\lambda y^o z^o.y)x : (o, o, o)$. On the other hand, η-reduction preserves long-safety:

Lemma 1.19 (η-reduction of one variable preserves long-safety)**.** *$\Gamma \vdash_{\mathsf{l}} \lambda\varphi.s\varphi : A$ with φ not occurring free in s implies $\Gamma \vdash_{\mathsf{l}} s : A$.*

Proof. Suppose $\Gamma \vdash_{\mathsf{l}} \lambda\varphi.s\varphi : A$. If s is an abstraction then by construction of the safe term $\lambda\varphi.s\varphi$, s is necessarily safe. If $s = N_0 \dots N_p$ with $p \geq 1$ then again, since $\lambda\varphi.N_0 \dots N_p\varphi$ is safe, each of the N_i is safe for $0 \leq i \leq p$ and for any $z \in fv(\lambda\varphi.s\varphi)$, $\mathsf{ord}(z) \geq \mathsf{ord}(\lambda\varphi.s\varphi) = \mathsf{ord}(s)$. Since φ does not occur free in s we have $fv(s) = fv(\lambda\varphi.s\varphi)$, thus we can use the application rule to form $fv(s) \vdash_{\mathsf{l}} N_0 \dots N_p : A$. The weakening rules permits us to conclude $\Gamma \vdash_{\mathsf{l}} s : A$. □

Lemma 1.20 (Long-safety is preserved by η-long expansion)**.** *$\Gamma \vdash_{\mathsf{l}} s : A$ then $\Gamma \vdash_{\mathsf{l}} \lceil s \rceil : A$.*

Proof. First we observe that for any variable or constant $x : A$ we have $x : A \vdash_{\mathsf{l}} \lceil x \rceil : A$. We show this by induction on $\mathsf{ord}(x)$. It is verified for any ground type variable x since $x = \lceil x \rceil$. Step case: $x : A$ with $A = (A_1, \ldots, A_n, o)$ and $n > 0$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. Since $\mathsf{ord}(A_i) < \mathsf{ord}(x)$ the induction hypothesis gives $\varphi_i : A_i \vdash_{\mathsf{l}} \lceil \varphi_i \rceil : A_i$. Using $(\mathsf{wk}_{\mathsf{l}})$ we obtain $x : A, \overline{\varphi} : \overline{A} \vdash_{\mathsf{l}} \lceil \varphi_i \rceil : A_i$. The application rule gives $x : A, \overline{\varphi} : \overline{A} \vdash_{\mathsf{l}} x\lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : o$ and the abstraction rule gives $x : A \vdash_{\mathsf{l}} \lambda\overline{\varphi}.x\lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil = \lceil x \rceil : A$.

We now prove the lemma by induction on s. The base case is covered by the previous observation. *Step case:*

- $s = xs_1 \ldots s_m$ with $x : (B_1, \ldots, B_m, A)$, $A = (A_1, \ldots, A_n, o)$ for some $m \geq 0$, $n > 0$ and $s_i : B_i$ for $1 \leq i \leq m$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. By the previous observation we have $\varphi_i : A_i \vdash_{\mathsf{l}} \lceil \varphi_i \rceil : A_i$, the weakening rule then gives us $\Gamma, \overline{\varphi} : \overline{A} \vdash_{\mathsf{l}} \lceil \varphi_i \rceil : A_i$. Since the judgement $\Gamma \vdash_{\mathsf{l}} xs_1 \ldots s_m : A$ is formed using the $(\mathsf{app}_{\mathsf{l}})$ rule, each s_j must be long-safe for $1 \leq j \leq m$, thus by the induction hypothesis we have $\Gamma \vdash_{\mathsf{l}} \lceil s_j \rceil : B_j$ and by weakening we get $\Gamma, \overline{\varphi} : \overline{A} \vdash_{\mathsf{l}} \lceil s_j \rceil : B_j$. The $(\mathsf{app}_{\mathsf{l}})$ rule then gives $\Gamma, \overline{\varphi} : \overline{A} \vdash_{\mathsf{l}} x\lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : o$. Finally the $(\mathsf{abs}_{\mathsf{l}})$ rule gives $\Gamma \vdash_{\mathsf{l}} \lambda\overline{\varphi}.x\lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil = \lceil s \rceil : A$, the side-condition of $(\mathsf{abs}_{\mathsf{l}})$ being verified since $\mathsf{ord}(\lceil s \rceil) = \mathsf{ord}(s)$.
- $s = ts_0 \ldots s_m$ where t is an abstraction. For some fresh variables $\varphi_1, \ldots, \varphi_n$ we have $\lceil s \rceil = \lambda\overline{\varphi}.\lceil t \rceil \lceil s_0 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil$. Again, using the induction hypothesis we can easily derive $\Gamma \vdash_{\mathsf{l}} \lambda\overline{\varphi}.\lceil t \rceil \lceil s_0 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : A$.
- $s = \lambda\overline{\eta}.t$ where $\overline{\eta} : \overline{B}$ and $t : C$ is not an abstraction. The induction hypothesis gives $\Gamma, \overline{\eta} : \overline{B} \vdash_{\mathsf{l}} \lceil t \rceil : C$ and using $(\mathsf{abs}_{\mathsf{l}})$ we get $\Gamma \vdash_{\mathsf{l}} \lambda\overline{\eta}.\lceil t \rceil = \lceil s \rceil : A$. □

Remark 1.21.

(1) The converse of this lemma does not hold in general: Performing η-reduction over a large abstraction does not in general preserve long-safety. This does not contradict Lemma 1.19 which states that safety is preserved when performing η-reduction on an abstraction of a *single* variable. The simplest counter-example is the term $f^{(o,o,o)} \vdash_{\mathsf{st}} \lambda x^o.f\underline{x}$ which is not long-safe and whose eta-long normal form $f^{(o,o,o)} \vdash_{\mathsf{l}} \lambda x^o y^o.fxy$ is long-safe. Even for closed terms the converse does not hold: $\lambda f^{(o,o,o)} g^{((o,o,o),o)}.g(\lambda x^o.f\underline{x})$ is not long-safe but its eta-normal form $\lambda f^{(o,o,o)} g^{((o,o,o),o)}.g(\lambda x^o y^o.fxy)$ is long-safe. In fact even the closed $\beta\eta$-normal term $\lambda f^{(o,(o,o),o,o)} g^{((o,o),o,o,o),o)}.g(\lambda y^{(o,o)} x^o.f\underline{xy})$ which is not long-safe has a long-safe η-long normal form!

(2) After performing η-long expansion of a term, all the occurrences of the application rule are made long-safe. Thus if a term remains not long-safe after η-long expansion, this means that some variable occurrence is not bound by the first following application of the (abs) rule in the typing tree.

Lemma 1.22. *A simply-typed term is safe if and only if its η-long normal form is long-safe.*

Proof. Let $\Gamma \vdash_{\mathsf{st}} M : T$. We want to show that $\Gamma \vdash_{\mathsf{l}} \lceil M \rceil : T$ if and only if $\Gamma \vdash_{\mathsf{s}} M : T$. The 'If' part can be proved by a trivial induction on the structure of $\Gamma \vdash_{\mathsf{s}} M : T$. For the 'Only if' part we proceed by induction on the structure of the simply-typed term $\Gamma \vdash_{\mathsf{st}} M : T$: The variable, constant and weakening cases are trivial. Suppose that M is an application of the form $xs_1 \ldots s_m : A$ for $m \geq 1$. Its η-long normal form is of the form $x\lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_m \rceil : o$ for some fresh variables $\varphi_1, \ldots \varphi_m$. By assumption this term

is long-safe term therefore we have $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$ and for $1 \leq i \leq m$, $\lceil s_i \rceil$ is also long-safe. By the induction hypothesis this implies that the s_is are all safe. We can then form the judgment $\Gamma \vdash_{\mathsf{s}} x s_1 \dots s_m : A$ using the rules (var) and (δ) followed by $m-1$ applications of the rule ($\mathsf{app}_{\mathsf{as}}$) and one application of (app) (this is allowed since we have $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$). The case $M \cong (\lambda x.s) s_1 \dots s_m$ for $m \geq 1$ is treated identically.

Suppose that $M \cong \lambda \overline{x}.s : A$. By assumption, its η-long n.f. $\lambda \overline{x}\overline{\varphi}.\lceil s \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_m \rceil : A$ (for some fresh variables $\varphi_1 \dots \varphi_m$) is long-safe. Thus we have $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$. Furthermore the long-safe subterm $\lceil s \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_m \rceil$ is precisely the eta-long expansion of $s\varphi_1 \dots \varphi_m : o$ therefore by the induction hypothesis we have that $s\varphi_1 \dots \varphi_m : o$ is safe. Since the φ_i's are all safe (by rule (var)), we can "peal-off" m applications of the rules ($\mathsf{app}_{\mathsf{as}}$)/($\mathsf{app}$) from the sequent $\Gamma, \overline{x}, \overline{\varphi} \vdash_{\mathsf{s}} s\varphi_1 \dots \varphi_m : o$ which gives us the sequent $\Gamma, \overline{x}, \overline{\varphi} \vdash_{\mathsf{as}} s : A$. Since the $\overline{\varphi}$ variables are fresh for s, we can further peal-off m applications of the weakening rule to obtain the judgment $\Gamma, \overline{x} \vdash_{\mathsf{s}} s : A$. Finally we obtain $\Gamma \vdash_{\mathsf{s}} \lambda \overline{x}.s : A$ using the rule (abs) (which is permitted since we have $\mathsf{ord}(A) \leq \mathsf{ord}(\Gamma)$). □

Proposition 1.23. *A term is safe if and only if its η-long normal form is safe.*

Proof. 'If':

$$\begin{aligned} \Gamma \vdash_{\mathsf{s}} \lceil M \rceil : T &\implies \Gamma \vdash_{\mathsf{l}} \lceil M \rceil : T && \text{By Lemma 1.22 (only if),} \\ &\implies \Gamma \vdash_{\mathsf{s}} \lceil M \rceil : T && \text{By Lemma 1.22 (if).} \end{aligned}$$

'Only if':

$$\begin{aligned} \Gamma \vdash_{\mathsf{s}} M : T &\implies \Gamma \vdash_{\mathsf{l}} \lceil M \rceil : T && \text{By Lemma 1.22 (only if),} \\ &\implies \Gamma \vdash_{\mathsf{s}} \lceil M \rceil : T && \text{By Lemma 1.18} \end{aligned}$$

□

The type inhabitation problem. It is well known that the simply-typed lambda calculus corresponds to intuitionistic implicative logic via the Curry-Howard isomorphism. The theorems of the logic correspond to inhabited types; further every inhabitant of a type represents a proof of the corresponding formula. Similarly, we can consider the fragment of intuitionistic implicative logic that corresponds to the safe lambda calculus under the Curry-Howard isomorphism. We call it the *safe fragment of intuitionistic implicative logic*.

An obvious first question is to compare the reasoning power of these two logics, in other words, to determine which types are inhabited in the lambda calculus but not in the safe lambda calculus.[6]

If types are generated from a single atom o, then there is a positive answer: every type generated from one atom that is inhabited in the lambda calculus is also inhabited in the safe lambda calculus. Indeed, one can transform any unsafe inhabitant M into a safe one of the same type as follows: Compute the eta-long beta normal form of M. Let x be an occurrence of a ground-type variable in a subterm of the form $\lambda \overline{x}.C[x]$ where $\lambda \overline{x}$ is the binder of x and for some context $C[\_\,]$ different from the identity $C[R] = R$. We replace the subterm $\lambda \overline{x}.C[x]$ by $\lambda \overline{x}.x$ in M. This transformation is sound because both $C[x]$ and x are of the same ground type. We repeat this procedure until the term stabilizes. This procedure clearly terminates since the size of the term decreases strictly after each step. The final term obtained is safe and of the same type as M.

[6]This problem was raised to our attention by Ugo dal Lago.

This argument cannot be generalized to types generated from multiple atoms. In fact there are order-3 types with only 2 atoms that are inhabited in the simply-typed lambda calculus but not in the safe lambda calculus. Take for instance the order-3 type $(((b,a),b),((a,b),a),a)$ for some distinct atoms a and b. It is only inhabited by the following family of terms which are all unsafe:

$$\begin{aligned}
&\lambda fg.g(\lambda x_1.f(\lambda y_1.x_1))\\
&\lambda fg.g(\lambda x_1.f(\lambda y_1.g(\lambda x_2.y_1)))\\
&\lambda fg.g(\lambda x_1.f(\lambda y_1.g(\lambda x_2.f(\lambda y_2.x_i)))) \qquad \text{where } i=1,2\\
&\lambda fg.g(\lambda x_1.f(\lambda y_1.g(\lambda x_2.f(\lambda y_2.g(\lambda x_3.y_i))))) \qquad \text{where } i=1,2\\
&\dots
\end{aligned}$$

Another example is the type of function composition. For any atom a and natural number $n \in \mathbb{N}$, we define the types n_a as follows: $0_a = a$ and $(n+1)_a = n_a \to a$. Take three distinct atoms a, b and c. For any $i,j,k \in \mathbb{N}$, we write $\sigma(i,j,k)$ to denote the type

$$\sigma(i,j,k) \cong (i_a \to j_b) \to (j_b \to k_c) \to i_a \to k_c \ .$$

For all i, j, k, this type is inhabited in the lambda calculus by the "function composition term":

$$\lambda xyzw.y(xz)$$

which is safe if and only if $i \geq j$. Suppose that $i < j$ then the type $\sigma(i,j,k)$ may still be safely inhabited. For instance $\sigma(1,3,4)$ is inhabited by the safe term

$$\lambda x^{1_a \to 3_b} y^{3_b \to 4_c} z^{1_c} w^{3_c}.y(x(\lambda u^a.u)) \ .$$

The order-4 type $\sigma(0,2,0)$, however, is only inhabited by the unsafe term $\lambda xyzw.y(xz)$.

Statman showed in [35] that the problem of deciding whether a type *defined over an infinite number of ground atoms* is inhabited (or equivalently of deciding validity of an intuitionistic implicative formula) is PSPACE-complete. The previous observations suggest that the validity problem for the safe fragment of implicative logic may not be PSPACE-hard.

2. Expressivity

2.1. **Numeric functions representable in the safe lambda calculus.** Natural numbers can be encoded in the simply-typed lambda calculus using the Church Numerals: each $n \in \mathbb{N}$ is encoded as the term $\overline{n} = \lambda sz.s^n z$ of type $I = ((o,o),o,o)$ where o is a ground type. We say that a p-ary function $f : \mathbb{N}^p \to \mathbb{N}$, for $p \geq 0$, is represented by a term $F : (I,\dots,I,I)$ (with $p+1$ occurrences of I) if for all $m_i \in \mathbb{N}$, $0 \leq i \leq p$ we have:

$$F\ \overline{m_1} \dots \overline{m_p} =_\beta \overline{f(m_1,\dots,m_p)} \ .$$

In 1976 Schwichtenberg [34] showed the following:

Theorem 2.1 (Schwichtenberg 1976)**.** *The numeric functions representable by simply-typed λ-terms of type $I \to \dots \to I$ using the Church Numeral encoding are exactly the multivariate polynomials* extended with the conditional function.

If we restrict ourselves to safe terms, the representable functions are exactly the multivariate polynomials:

Theorem 2.2. *The functions representable by safe λ-expressions of type $I \to \ldots \to I$ are exactly the multivariate polynomials.*

Proof. Natural numbers are encoded as the Church Numerals: $\overline{n} = \lambda sz.s^n z$ for each $n \in \mathbb{N}$. Addition: For $n, m \in \mathbb{N}$, $\overline{n+m} = \lambda\alpha^{(o,o)}x^o.(\overline{n}\alpha)(\overline{m}\alpha x)$. Multiplication: $\overline{n.m} = \lambda\alpha^{(o,o)}.\overline{n}(\overline{m}\alpha)$. These terms are safe and clearly any multivariate polynomial $P(n_1, \ldots, n_k)$ can be computed by composing the addition and multiplication terms as appropriate.

For the converse, let U be a safe λ-term of type $I \to I \to I$. The generalization to terms of type $I^n \to I$ for any $n \in \mathbb{N}$ is immediate (they correspond to polynomials with n variables). By Lemma 1.23, safety is preserved by η-long normal expansion therefore we can assume that U is in η-long normal form.

Let us write $\mathcal{N}_\Sigma^\tau$ for the set of safe η-long β-normal terms of type τ with free variables in Σ, and $\mathcal{A}_\Sigma^\tau$ for the set of β-normal terms of type τ with free variables in Σ and of the form $\varphi s_1 \ldots s_m$ for some variable $\varphi : (A_1, \ldots, A_m, o)$ where $m \geq 0$ and for all $1 \leq i \leq m$, $s_i \in \mathcal{N}_\Sigma^{A_i}$. Observe that the set $\mathcal{A}_\Sigma^o$ contains only safe terms but the sets $\mathcal{A}_\Sigma^\tau$ in general may contain unsafe terms. Let Σ denote the alphabet $\{x, y : I, z : o, \alpha : o \to o\}$. The sets $\mathcal{N}_\emptyset^0$ is given by the following grammar defined over the set of terminals $\Sigma \cup \{\lambda xy\alpha z., \lambda z.\}$:

$$\begin{array}{rcl}
\mathcal{N}_\emptyset^{(I,I,I)} & \to & \lambda xy\alpha z.\mathcal{A}_\Sigma^o \\
\mathcal{A}_\Sigma^o & \to & z \mid \mathcal{A}_\Sigma^{(o,o)}\mathcal{A}_\Sigma^o \\
\mathcal{A}_\Sigma^{(o,o)} & \to & \alpha \mid \mathcal{A}_\Sigma^I\ \mathcal{N}_\Sigma^{(o,o)} \\
\mathcal{N}_\Sigma^{(o,o)} & \to & \lambda z.\mathcal{A}_\Sigma^o \\
\mathcal{A}_\Sigma^I & \to & x \mid y
\end{array}$$

The key rule is the fourth one: if we had not imposed the safety constraint the right-hand side would instead be of the form $\lambda w^o.\mathcal{A}_{\Sigma\cup\{w:o\}}^{(o,o)}$. Here the safety constraint imposes to abstract all the ground type variables occurring freely, thus only one free variable of ground type can appear in the term and we can choose it to be named z up to α-conversion.

We extend the notion of representability to terms of type o, (o,o) and I with free variables in Σ as follows: a function $f : \mathbb{N}^2 \to \mathbb{N}$ is represented by a term $\Sigma \vdash_{\mathsf{st}} F : o$ if and only if for all $m, n \in \mathbb{N}$, $F[\overline{m}, \overline{n}/x, y] =_\beta \alpha^{\overline{f(m,n)}}z$, by a term $\Sigma \vdash_{\mathsf{st}} G : (o,o)$ iff $G[\overline{m}, \overline{n}/x, y] =_\beta \lambda z.\alpha^{\overline{f(m,n)}}z$, and by $\Sigma \vdash_{\mathsf{st}} H : I$ iff $H[\overline{m}, \overline{n}/x, y] =_\beta \lambda\alpha z.\alpha^{\overline{f(m,n)}}z$.

We now show by induction on the grammar rules that any term generated by the grammar represents some polynomial: The term x and y represent the projection functions $(m, n) \mapsto m$ and $(m, n) \mapsto n$ respectively. The term α and z represent the constant functions $(m, n) \mapsto 1$ and $(m, n) \mapsto 0$ respectively. If $F \in \mathcal{A}_\Sigma^o$ represents the functions f then so does $\lambda z.F$.

We make the following observations: let $m, p, p' \geq 0$

(1) $\overline{m}\,(\lambda z.\alpha^p z) =_\beta \lambda z.\alpha^{m\cdot p}z$

(2) $(\lambda z.\alpha^p z)(\alpha^{p'} z) =_\beta \alpha^{p+p'}z$

Now suppose that $F \in \mathcal{A}_\Sigma^I$ and $G \in \mathcal{N}_\Sigma^{(o,o)}$ represent the functions f and g respectively then by the previous observation, FG represents the function $f \times g$. And if $F \in \mathcal{A}_\Sigma^{(o,o)}$ and $G \in \mathcal{N}_\Sigma^o$ represent the functions f and g then FG represents the function $f + g$.

Thus U represents some polynomial as required, *i.e.* for all $m, n \in \mathbb{N}$ we have $U\ \overline{m}\ \overline{n} =_\beta \lambda\alpha z.\alpha^{p(m,n)}z$ where $p(m, n) = \sum_{0\leq k\leq d} m^{i_k} n^{j_k}$ for some $i_k, j_k \geq 0$, $d \geq 0$. □

Corollary 2.3. *The conditional operator $C : I \to I \to I \to I$ satisfying:*

$$C\ t\ y\ z \to_\beta \begin{cases} y, & \text{if } t \to_\beta \overline{0}; \\ z, & \text{if } t \to_\beta \overline{n+1}\ . \end{cases}$$

is not definable in the simply-typed safe lambda calculus.

Example 2.4. The term $\lambda FGH\alpha x.F(\underline{\lambda y.G\alpha x})(H\alpha x)$ used by Schwichtenberg [34] to define the conditional operator is unsafe since the underlined subterm, which is of order 1, occurs at an operand position and contains an occurrence of x of order 0.

Remark 2.5.

(1) This corollary tells us that the conditional function is not definable when numbers are represented by the Church Numerals. It may still be possible, however, to represent the conditional function using a different encoding for natural numbers. A possible way to compensate for the loss of expressivity caused by the safety constraint consists in introducing countably many domains of representation for natural numbers. This is the technique that is used to represent the predecessor function in the simply-typed lambda calculus [14].
(2) The boolean conditional can be represented in the safe lambda calculus as follows: We encode booleans by terms of type $B = ((o, o), o, o)$. The two truth values are then represented by $\lambda x^o y^o.x$ and $\lambda x^o y^o.y$ and the conditional by $\lambda F^B G^B H^B.F\ G\ H$.
(3) It is also possible to define a conditional operator behaving like the conditional operator C in the second-order lambda calculus [14]: natural numbers are represented by terms $\overline{n} \equiv \Lambda t.\lambda s^{t\to t} z^t.s^n(z)$ of type $J \equiv \Delta t.(t \to t) \to (t \to t)$ and the conditional is encoded by the term $\lambda F^J G^J H^J.F\ J\ (\lambda u^J.G)\ H$. Whether this term is safe or not cannot be answered just yet as we do not have a notion of safety for second-order typed term.

2.2. **Word functions definable in the safe lambda calculus.** Schwichtenberg's result on numeric functions definable in the lambda calculus was extended to richer structures: Zaionc studied the problem for words functions, then functions over trees and eventually the general case of functions over free algebras [20, 40, 39, 37, 41]. In this section we consider the case of word functions expressible in the safe lambda calculus.

Notations. We consider equality of terms modulo α, β and η conversion, and we write $M =_{\beta\eta} N$ to denote this equality. For any simple type τ, we write $\mathrm{Cl}(\tau)$ for the set of closed terms of type τ (modulo α, β and η conversion). We consider a binary alphabet $\Sigma = \{a, b\}$. The result naturally extends to all finite alphabets. We consider the set Σ^* of all words over Σ. The empty words is denoted ϵ. We write $|w|$ to denote the length of the word $w \in \Sigma^*$. For any $k \in \mathbb{N}$ we write $\mathbf{k}$ to denote the word $a \dots a$ with k occurrences of a, so that $|\mathbf{k}| = k$. For any $n \geq 1$ and $k \geq 0$, we write $c(n, k)$ for the n-ary function $(\Sigma^*)^n \to \Sigma^*$ that maps all inputs to the word $\mathbf{k}$. The function $\mathsf{app} : (\Sigma^*)^2 \to \Sigma^*$ is the usual concatenation function: $\mathsf{app}(x, y)$ is the word obtain by concatenating x and y. The substitution function $\mathsf{sub} : (\Sigma^*)^3 \to \Sigma^*$ is defined as follows: $\mathsf{sub}(x, y, z)$ is the word obtained from x by substituting the word y for all occurrences of a and z for all occurrences of b.

Take the type $\mathbf{B} = (o \to o) \to (o \to o) \to o \to o$, which is called *the binary word type* in [37]. There is a 1-1 correspondence between words over Σ and closed terms of type $\mathbf{B}$: the empty word ϵ is represented by $\lambda uvx.x$, and if $w \in \Sigma^*$ is represented by a term $W \in \mathrm{Cl}(\mathbf{B})$

then $a \cdot w$ is represented by $\lambda uvx.u(Wuvx)$ and $b \cdot w$ is represented by $\lambda uvx.v(Wuvx)$. The term representing the word w is denoted by $\underline{w}$. A closed term of type $\mathbf{B}^n \to \mathbf{B}$ is called a ***word function***. We say that the function on words $h : (\Sigma^*)^n \to \Sigma^*$ is ***represented*** by the term $H \in \mathrm{Cl}(\mathbf{B}^n \to \mathbf{B})$ just if for all $x_1, \ldots, x_n \in \mathbf{B}^*$, $H\underline{x_1} \ldots \underline{x_n} = \underline{hx_1 \ldots x_n}$.

It was shown in [37] that there is a finite base of word functions in the sense that every λ-definable word function is some composition of functions from the base:

Theorem 2.6 (Zaionc [37])**.** *The set of λ-definable word functions is the minimal set containing the following word functions and closed by compositions:*

- *concatenation* app*;*
- *substitution* sub*;*
- *extraction of the maximal prefix containing only a given letter;*
- *non-emptiness check: returns* $\mathbf{0}$ *if the word is* ϵ *and* $\mathbf{1}$ *otherwise, as well as emptiness check;*
- *occurrence checking: returns* $\mathbf{1}$ *if the word contain an occurrence of a given letter and* $\mathbf{0}$ *otherwise;*
- *first-occurrence check: test whether the word begins with a given letter;*
- *all the projections;*
- *all the constant functions.*

The lambda terms representing the base functions are:

$$\begin{aligned}
\mathrm{APP} &= \lambda cduvx.cuv(duvx), & \mathrm{SUB} &= \lambda xdeuvx.c(\lambda y.duvy)(\lambda y.euvy)x,\\
\mathrm{CUT}_a &= \lambda cuvx.cu(\lambda y.x)x, & \mathrm{CUT}_b &= \lambda cuvx.c(\lambda y.x)vx,\\
\mathrm{SQ} &= \lambda cuvx.c(\lambda y.ux)(\lambda y.ux)x, & \overline{\mathrm{SQ}} &= \lambda cuvx.c(\lambda y.x)(\lambda y.x)(ux),\\
\mathrm{BEG}_a &= \lambda cuvx.c(\lambda y.ux)(\lambda y.x)x, & \mathrm{BEG}_b &= \lambda cuvx.c(\lambda y.x)(\lambda y.ux)x,\\
\mathrm{OCC}_a &= \lambda cuvx.c(\lambda y.ux)(\lambda y.y)x, & \mathrm{OCC}_b &= \lambda cuvx.c(\lambda y.y)(\lambda y.ux)x.
\end{aligned}$$

where APP represents concatenation, SUB substitution, SQ and $\overline{\mathrm{SQ}}$ non-emptiness and emptiness checking, BEG_a and BEG_b first-occurrence test, and OCC_a and OCC_a occurrence test.

We observe that among these terms only APP and SUB are safe. All the other terms are unsafe because they contain terms of the form $N(\lambda y.x)$ where x and y are of the same order. It turns out that APP and SUB constitute a base of terms generating all the functions definable in the safe lambda calculus as the following theorem states:

Theorem 2.7. *Let* $\lambda^{safe}\mathrm{def}$ *denote the minimal set containing the following word functions and closed by compositions:*

- *concatenation* app*;*
- *substitution* sub*;*
- *all the projections;*
- *all the constant functions.*

The set of word-functions definable in the safe lambda calculus is precisely $\lambda^{safe}\mathrm{def}$.

The proof follows the same steps as Zaionc's proof. The first direction is immediate: the terms APP and SUB are safe and represent concatenation and substitution. Projections are represented by safe terms of the form $\lambda x_1 \ldots x_n.x_i$ for some $i \in \{1..n\}$, and constant functions by $\lambda x_1 \ldots x_n.\underline{w}$ for some $w \in \Sigma^*$. For composition, take a functions $g : (\Sigma^*)^n \to$

Σ^* represented by safe term $G \in \mathrm{Cl}(\mathbf{B}^n \to \mathbf{B})$ and functions $f_1, \ldots, f_n : (\Sigma^*)^p \to \Sigma^*$ represented by safe terms $F_1, \ldots F_n$ respectively then the function

$$(x_1, \cdots, x_p) \mapsto g(f_1(x_1, \ldots, x_p), \ldots, f_n(x_1, \ldots, x_p))$$

is represented by the term $\lambda c_1 \ldots x_p.G(F_1 c_1 \ldots c_p) \ldots (F_n c_1 \ldots c_p)$ which is also safe.

To show the other directions we need to introduce some more definitions. We will write $\mathrm{Op}(n,k)$ to denote the set of open terms of the form:

$$c_1 : \mathbf{B}, \ldots c_n : \mathbf{B}, u : (o,o), v : (o,o), x_{k-1} : o, \ldots, x_0 : o \vdash_{\mathsf{st}} M : o \ .$$

Thus we have the following equality (modulo α, β and η conversions) for $n, k \geq 1$:

$$\mathrm{Cl}(\tau(n,k)) = \{\lambda c_1^{\mathbf{B}} \ldots c_n^{\mathbf{B}} u^{(o,o)} v^{(o,o)} x_{k-1}^o \ldots x_0^o.M \mid M \in \mathrm{Op}(n,k)\}$$

writing $\tau(n,k)$ as a shorthand for the type $(\mathbf{B}^n, (o,o), (o,o), \overbrace{o, \ldots, o}^{k \text{ times}}, o)$. We generalized the notion of representability to terms of type $\tau(n,k)$ as follows:

Definition 2.8 (Function pair representation)**.** A closed term $T \in \mathrm{Cl}(\tau(n,k))$ ***represents the pair of functions*** (f,p) where $f : (\Sigma^*)^n \to \Sigma^*$ and $p : (\Sigma^*)^n \to \{\mathbf{0}, \ldots, \mathbf{k-1}\}$ if for all $w_1, \ldots, w_n \in \Sigma^*$ and for every $i \in \{0 \ldots, k-1\}$ we have:

$$T\underline{w_1} \ldots \underline{w_n} =_{\beta\eta} \lambda u v x_{k-1} \ldots x_0.\underline{f(w_1, \ldots, w_n)} u v x_{|p(w_1,\ldots,w_n)|} \ .$$

By extension we will say that an *open* term M from $\mathrm{Op}(n,k)$ represents the pair (f,p) iif $M[\underline{w_1} \ldots \underline{w_n}/c_1 \ldots c_n] =_{\beta\eta} \underline{f(w_1, \ldots, w_n)} u v x_{|p(w_1,\ldots,w_n)|}$.

We will call ***safe pair*** any pair of functions of the form $(w, c(n,i))$ where $0 \leq i \leq k-1$ and w is an n-ary function from λ^{safe}def.

Theorem 2.9 (Characterization of the representable pairs)**.** *The function pairs representable in the safe lambda calculus are precisely the safe pairs.*

Proof. (Soundness). Take a pair $(w, c(n,i))$ where $0 \leq i \leq k-1$ and w is an n-ary function from λ^{safe}def. As observed earlier, all the functions from λ^{safe}def are representable in the safe lambda calculus: let $\underline{w}$ be the representative of w. The pair $(w, c(n,i))$ is then represented by the term $\lambda c_1 \ldots c_n u v x_{k-1} \ldots x_0.\underline{w} c_1 \ldots c_n u v x_i$.

(Completeness) It suffices to consider safe β-η-long normal terms from $\mathrm{Op}(n,k)$ only. The result then immediately follows for any safe term in $\mathrm{Cl}(\tau(n,k))$. The subset of $\mathrm{Op}(n,k)$ constituted of β-η-long normal terms is generated by the following grammar (see [37]):

$$\begin{array}{llll}
(\alpha_i^k) & R^k & \to & x_i \\
(\beta^k) & & \mid & u R^k \\
(\gamma^k) & & \mid & v R^k \\
(\delta_j^k) & & \mid & c_j \overbrace{(\lambda z^k.R^{k+1}[z^k, x_0, \ldots, x_{k-1}/x_0, x_1, \ldots, x_k])}^{Q^k(R^{k+1})} \\
 & & & \quad (\lambda z^k.R^{k+1}[z^k, x_0, \ldots, x_{k-1}/x_0, x_1, \ldots, x_k]) \\
 & & & \quad R^k
\end{array}$$

for $k \geq 1$, $0 \leq i < k$, $0 \leq j \leq n$. The notation $M[\ldots/\ldots]$ denotes the usual simultaneous substitution. The name of each rule is given in parenthesis. The non-terminals are R^k for $k \geq 1$ and the set of terminals is $\{z^k, \lambda z^k \mid k \geq 1\} \cup \{x_i \mid i \geq 0\} \cup \{c_1, \ldots, c_n, u, v\}$.

We identify a rule name with the right-hand side of the corresponding rule, thus α_i^k belongs to $\mathrm{Op}(n,k)$, β^k and γ^k are functions from $\mathrm{Op}(n,k)$ to $\mathrm{Op}(n,k)$, and δ_j^k is a function from $\mathrm{Op}(n,k+1) \times \mathrm{Op}(n,k+1) \times \mathrm{Op}(n,k)$ to $\mathrm{Op}(n,k)$.

We now want to characterize the subset consisted of all *safe* terms generated by this grammar. The term α_i^k is always safe, $\beta^k(M)$ and $\gamma^k(M)$ are safe if and only if M is, and $\delta_j^k(F,G,H)$ is safe if and only if $Q^k(F)$, $Q^k(G)$ and H are safe. We observe that the free variables of $Q^k(F)$ all belong to $\{c_1, \ldots c_n, u, v, x_0, \ldots x_k\}$. All these variables have order greater than $\mathsf{ord}(z)$ except the x_is which have same order as z. Hence since the x_is are not abstracted together with z we have that $Q^k(F)$ is safe if and only if F is safe and the variables $x_0 \ldots x_k$ do not appear free in $F[z^k, x_0, \ldots, x_{k-1}/x_0, x_1, \ldots, x_k]$, which is the same as saying that the variables $x_1 \ldots x_k$ do not appear free in F. Similarly, $Q^k(G)$ is safe if and only if G is safe and variables $x_1 \ldots x_k$ do not appear free in G.

We therefore need to identify the subclass of terms generated by the non-terminal R^k which are safe and which do not have free occurrences of variables in $\{x_1 \ldots x_{k-1}\}$. By applying this requirement to the rules of the previous grammar we obtain the following specialized grammar characterizing the desired subclass:

$$
\begin{array}{lll}
(\overline{\alpha}_0^k) & \overline{R}^k & \rightarrow\ x_0 \\
(\overline{\beta}^k) & & \mid u\overline{R}^k \\
(\overline{\gamma}^k) & & \mid v\overline{R}^k \\
(\overline{\delta}_j^k) & & \mid c_j\ (\lambda z^k.\overline{R}^{k+1}[z^k/x_0])\ (\lambda z^k.\overline{R}^{k+1}[z^k/x_0])\ \overline{R}^k\ .
\end{array}
$$

For any term M, $Q^k(M)$ is safe if and only if M can be generated from the non-terminal $\overline{R}^k$. Thus the subset of $\mathrm{Cl}(\tau(n,k))$ consisting of safe beta-normal terms is given by the grammar:

$$
\begin{array}{lll}
(\widetilde{\pi}^k) & \widetilde{S} & \rightarrow \lambda c_1 \ldots c_n uv x_{k-1} \ldots x_0.\widetilde{R}^k \\
(\widetilde{\alpha}_i^k) & \widetilde{R}^k & \rightarrow\ x_i \\
(\widetilde{\beta}^k) & & \mid u\widetilde{R}^k \\
(\widetilde{\gamma}^k) & & \mid v\widetilde{R}^k \\
(\widetilde{\delta}_j^k) & & \mid c_j\ (\lambda z^k.\overline{R^{k+1}}[z^k/x_0])\ (\lambda z^k.\overline{R^{k+1}}[z^k/x_0])\ \widetilde{R}^k\ .
\end{array}
$$

Thus to conclude the proof, it suffices to show that every term that can be generated by this grammar starting with the non-terminal $\widetilde{S}$ represents a safe pair.

We proceed by induction and show that the non-terminal $\overline{R}^k$ generates terms representing pairs of the form $(w, c(n,0))$ while non-terminals $\widetilde{S}$ and $\widetilde{R}^k$ generate terms representing pairs of the form $(w, c(n,i))$ for $0 \leq i < k$ and $w \in^{safe}$def.

Base case: The term $\overline{\alpha}_0^k$ represents the safe pair $(c(n,0), c(n,0))$ while $\widetilde{\alpha}_i^k$ represents the safe pair $(c(n,0), c(n,i))$. *Step case:* Suppose $T \in \mathrm{Op}(n,k)$ represents a pair (w,p). Then $\overline{\alpha}^k(T)$ and $\widetilde{\alpha}^k(T)$ represent the pair $(\mathsf{app}(a,w),p)$, $\overline{\beta}^k(T)$ and $\widetilde{\beta}^k(T)$ represent the pair $(\mathsf{app}(b,w),p)$, and $\overline{\pi}^k(T) \in \mathrm{Cl}(\tau(n,k))$ represents the pair (w,p). Now suppose that E, F and G represent the pairs $(w_e, c(n,0))$, $(w_f, c(n,0))$ and $(w_g, c(n,i))$ respectively. Then

we have:

$$
\begin{aligned}
&\widetilde{\delta}^k_j(E,F,G)[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n]\\
&\qquad= \underline{w_j}\ (\lambda z^k.E[z^k/x_0])[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n]\\
&\qquad\qquad (\lambda z^k.F[z^k/x_0])[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n]\\
&\qquad\qquad G[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n]\\
&\qquad=_{\beta\eta} \underline{w_j}\ (\lambda z^k.E[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n][z^k/x_0])\\
&\qquad\qquad (\lambda z^k.F[\underline{w_1}\dots\underline{w_n}/c_1\dots c_n][z^k/x_0])\\
&\qquad\qquad (\underline{w_g(w_1\dots w_n)}\ u\ v\ x_i) && G \text{ represents } (h,c(n,i))\\
&\qquad=_{\beta\eta} \underline{w_j}\ (\lambda z^k.(\underline{w_e(w_1\dots w_n)}\ u\ v\ x_0)[z^k/x_0]) && E \text{ represents } (f,c(n,0))\\
&\qquad\qquad (\lambda z^k.(\underline{w_f(w_1\dots w_n)}\ u\ v\ x_0)[z^k/x_0]) && F \text{ represents } (g,c(n,0))\\
&\qquad\qquad (\underline{w_g(w_1\dots w_n)}\ u\ v\ x_i)\\
&\qquad=_{\beta\eta} \underline{w_j}\ (\lambda z^k.\underline{w_e(w_1\dots w_n)}\ u\ v\ z^k)\\
&\qquad\qquad (\lambda z^k.\underline{w_f(w_1\dots w_n)}\ u\ v\ z^k)\\
&\qquad\qquad (\underline{w_g(w_1\dots w_n)}\ u\ v\ x_i)\\
&\qquad=_{\eta} \underline{w_j}\ (\underline{w_e(w_1\dots w_n)}\ u\ v)\ (\underline{w_f(w_1\dots w_n)}\ u\ v)\ (\underline{w_g(w_1\dots w_n)}\ u\ v\ x_i)\\
&\qquad=_{\beta\eta} \underline{w}\ u\ v\ x_i
\end{aligned}
$$

where the word-function w is defined as

$$w : w_1,\dots,w_n \mapsto \mathsf{app}(\mathsf{sub}(w_j, w_e(w_1,\dots,w_n), w_f(w_1,\dots,w_n)), w_g(x_1,\dots,w_n))\ .$$

Hence $\widetilde{\delta}^k_j(E,F,G)$ represents the pair $(w,c(n,i))$.

The same argument shows that if E, F and G all represent safe pairs then so does $\overline{\delta}^k_j(E,F,G)$. □

By instantiating Theorem 2.9 with terms of types $\tau(n,1) = I^n \to I$ we obtain that every closed safe term of this type represents some n-ary function from λ^{safe}def. This concludes the proof of the characterization Theorem 2.7.

2.3. **Representability of functions over other structures.**

There is an isomorphism between binary trees and closed terms of type $\tau = (o \to o \to o) \to o \to o$. Thus any closed term of type $\tau \to \tau \to \ldots \to \tau$ represents an n-ary function over trees. Zaionc gave a characterization of the set of tree functions representable in the simply-typed lambda calculus [38]: it is precisely the minimal set containing constant functions, projections and closed under composition and limited primitive recursion. Zaionc showed that the same characterization holds for the general case of functions expressed over free algebras [40] (they are again given by the minimal set containing constant functions, projections and closed under composition and limited recursion). This result subsumes Schwichtenberg's result on definable numeric functions as well as Zaionc's own results on definable word and tree functions.

Among all these basic operations, only limited recursion is unsafe. We conjecture that the set of tree functions representable in the safe lambda calculus is given by the set

containing constant functions, projections and closed under composition (but not by limited recursion).

3. Complexity of the safe lambda calculus

This section is concerned with the complexity of the beta-eta equivalence problem for the safe lambda calculus: given two safe lambda terms, are they equivalent up to $\beta\eta$-conversion?

Let $\exp_h(m)$ denote the tower-of-exponential function defined by $\exp_0(m) = m$ and $\exp_{h+1}(m) = 2^{\exp_h(m)}$. Recall that a program is ***elementary recursive*** if its run-time can be bounded by $\exp_K(n)$ for some constant K where n is the length of the input.

3.1. **Statman's result.** A famous result by Statman states that deciding the $\beta\eta$-equality of two first-order typable lambda terms is not elementary recursive [36]. The proof proceeds by encoding the Henkin quantifier elimination of type theory in the simply-typed lambda calculus. Simpler proofs have subsequently been given, one by Mairson in [23] and another by Loader in [22]. Both proceed by encoding the Henkin quantifier elimination procedure in the lambda-calculus, as in the original proof, but their use of list iteration to perform quantifier elimination makes them much easier to understand.

It turns out that all these encodings rely on unsafe terms: Statman's encoding uses the conditional function sg which is not definable in the safe lambda-calculus [8]; Mairson's encoding uses unsafe terms to encode both quantifier elimination and set membership, and Loader's encoding uses unsafe term to build list iterators. We are thus led to conjecture that finite type theory (see definition in Sec. 3.2) is intrinsically unsafe in the sense that every encoding of it in the lambda calculus is necessarily unsafe. Of course this conjecture does not rule out the possibility that another non-elementary problem is encodable in the safe lambda calculus.

We start this section by presenting an adaptation of Mairson's encoding. We show that quantifier elimination can be safely encoded and explain why it is problematic to encode set-membership safely. We will then use this encoding to interpret the True Quantifier Boolean Formula (TQBF) problem in the safe lambda calculus, thus showing that deciding beta-eta equality is PSPACE-hard.

3.2. **Mairson's encoding.** We recall the definition of finite type theory. Let us define $\mathcal{D}_0 = \{\mathbf{true}, \mathbf{false}\}$ and $\mathcal{D}_{k+1} = powerset(\mathcal{D}_k)$. For $k \geq 0$, we write x^k, y^k and z^k to denote variables ranging over $\mathcal{D}_k$. Prime formulas are x^0, $\mathbf{true} \in y^1$, $\mathbf{false} \in y^1$, and $x^k \in y^{k+1}$. Formulae are built up from prime formulae using the logical connectives $\wedge$,$\vee$,$\rightarrow$,$\neg$ and the quantifiers $\forall$ and $\exists$. Meyer showed that deciding the validity of such formulae requires nonelementary time [26].

In Mairson's encoding, boolean values are encoded by terms of type $\mathsf{B} = \sigma \rightarrow \sigma \rightarrow \sigma$ for some type σ, and variables of order $k \geq 0$ are encoded by terms of type Δ_k defined as $\Delta_0 \equiv \mathsf{B}$ and $\Delta_{k+1} \equiv {\Delta_k}^*$ where for any type α, $\alpha^* = (\alpha \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$ for some type τ. Using this encoding, unsafety manifests itself in two different ways.

1. First in the encoding of set membership. The prime formula $x^k \in y^{k+1}$ is encoded as

$$x^k : \Delta_k, y^{k+1} : \Delta_{k+1} \vdash_{\mathsf{st}} y^{k+1}(\lambda y^k : \Delta_k.OR(eq_k\ \underline{x^k}\ y^k)\ F : \Delta_k \rightarrow \Delta_{k+1} \rightarrow \Delta_0 \qquad (3.1)$$

for some terms OR, F, eq_k. This term is unsafe because of the underline occurrence of x^k which is not abstracted together with y^k.

2. Secondly, quantifier elimination is performed using a list iterator $\mathbf{D}_{k+1}$ of type Δ_{k+2} which acts like the $fold\_right$ function (from functional programming) over the list of all elements of $\mathcal{D}_k$. Thus for instance the formula $\forall x^0.\exists y^0.x^0 \vee y^0$ is encoded as

$$\vdash_{\mathsf{st}} \mathbf{D}_0(\lambda x^0 : \Delta_0.AND(\mathbf{D}_0(\lambda y^0 : \Delta_0.OR(\underline{x^0} \vee y^0))F))\ T : \mathsf{B}$$

where the type τ is instanciated as B. This term is unsafe since the underlined occurrence is unsafely bound. This is due to the presence of two nested quantifiers in the formula, which are encoded as two nested list iterations. More generally, nested binding will be encoded safely if and only if every variable x in the formula is bound by the first quantifier $\exists z$ or $\forall z$ in the path to the root of the AST of the formula verifying $\mathsf{ord}(z) \geq \mathsf{ord}(x)$. For instance if set-membership could be encoded safely then the interpretation of $\forall x^k.\exists y^{k+1}.x^k \in y^{k+1}$ would be unsafe whereas the encoding of $\forall y^{k+1}.\exists x^k.x^k \in y^{k+1}$ would be safe.

Surprisingly, the 'unsafety' of the quantifier elimination procedure can be easily overcome. The idea is as follows. We introduce multiple domains of representation for a given formula. An element of $\mathcal{D}_k$ is thereby represented by countably many terms of type Δ_k^n where $n \in \mathbb{N}$ indicates the level of the domain of representation. The type Δ_k^n is defined in such a way that its order strictly increases as n grows. Furthermore, there exists a term that can lower the domain of representation of a given term. Thus each formula variable can have a different domain of representation, and since there are infinitely many such domains, it is always possible to find an assignment of representation domains to variables such that the resulting encoding term is safe.

For set-membership, however, there is no obvious way to obtain a safe encoding. In order to turn Mairson's encoding of set-membership (Eq. 3.1) into a safe term, we would need to have access to a function that changes the domain of representation of an encoded higher-order value of the type-hierarchy. Unfortunately, such transformation is intrinsically unsafe!

We now present the encoding in details.

3.2.1. *Encoding basic boolean operations.* Let o be a base type and define the family of types $\sigma_0 \equiv o$, $\sigma_{n+1} \equiv \sigma_n \to \sigma_n$ satisfying $\mathsf{ord}(\sigma_n) = n$. Booleans are encoded over domains $\mathsf{B}_n \equiv \sigma_n \to o \to o \to o$ for $n \geq 0$, each type B_n being of order $n+1$. We write $\underline{i}_{n+1}$ to denote the term $\lambda x^{\sigma_n}.x : \sigma_{n+1}$ for $n \geq 0$. The truth values **true** and **false** are represented by the following terms parameterized by $n \in \mathbb{N}$:

$$T^n \equiv \lambda u^{\sigma_n} x^o y^o.x : \mathsf{B}_n$$
$$F^n \equiv \lambda u^{\sigma_n} x^o y^o.y : \mathsf{B}_n$$

Clearly these terms are safe. Moreover the following relations hold for all $n, n' \geq 0$:

$$\lambda u^{\sigma_{n'}}.T^{n+1}\ \underline{i}_{n+1} \to_\beta T^{n'}$$
$$\lambda u^{\sigma_{n'}}.F^{n+1}\ \underline{i}_{n+1} \to_\beta F^{n'}$$

Hence it is possible to change the domain of representation of a Boolean value from a higher-level to another arbitrary level using the transformation:

$$\mathbf{C}_0^{n+1 \mapsto n'} \equiv \lambda m^{\mathsf{B}_{n+1}} u^{\sigma_{n'}}.m\ \underline{i}_{n+1} : \mathsf{B}_{n+1} \to \mathsf{B}_{n'}$$

so that if a term $M : \mathsf{B}_n$ for $n \geq 1$ is beta-eta convertible to T^n (resp. F^n) then $\mathbf{C}_0^{n \mapsto n'}\ M : \mathsf{B}_{n'}$ is beta-eta convertible to $T^{n'}$ (resp. $F^{n'}$).

Observe that although the term $\mathbf{C}_0^{n+1 \mapsto n'}$ is safe for all $n, n' \geq 0$, if we apply it to a variable then the resulting term

$$x : B_{n+1} \vdash_{\mathsf{st}} \mathbf{C}_0^{n+1 \mapsto n'}\ x : B_n$$

is safe if and only if the transformation decreases the domain of representation of x *i.e.* $\mathsf{ord}(B_{n+1}) \geq \mathsf{ord}(B_{n'})$.

Boolean functions are encoded by the following safe terms parameterized by n:

$$\begin{aligned} AND^n &\equiv \lambda p^{\mathsf{B}_n} q^{\mathsf{B}_n} u^{\sigma_n} x^o y^o . p\ u\ (q\ u\ x\ y)\ y : \mathsf{B}_n \to \mathsf{B}_n \to \mathsf{B}_n \\ OR^n &\equiv \lambda p^{\mathsf{B}_n} q^{\mathsf{B}_n} u^{\sigma_n} x^o y^o . p\ u\ x\ (q\ u\ x\ y) : \mathsf{B}_n \to \mathsf{B}_n \to \mathsf{B}_n \\ NOT^n &\equiv \lambda p^{\mathsf{B}_n} u^{\sigma_n} x^o \lambda y^o . p\ u\ y\ x : \mathsf{B}_n \to \mathsf{B}_n \to \mathsf{B}_n \end{aligned}$$

3.2.2. *Coding elements of the type hierarchy.* For any $n \in \mathbb{N}$ we define the hierarchy of type Δ_k^n as follows: $\Delta_0^n \equiv \mathsf{B}_n$ and $\Delta_{k+1}^n \equiv \Delta_k^{n*}$ where for any type α, $\alpha^* = (\alpha \to \tau \to \tau) \to \tau \to \tau$. An occurrence of a formula variable x^k will be encoded as a term variable x^k of type Δ_k^n for some level of domain representation $n \in \mathbb{N}$. Following Mairson's encoding, each set $\mathcal{D}_k$ is represented by a list $\mathbf{D}_k^n$ constituted of all its elements:

$$\begin{aligned} \mathbf{D}_0^n &\equiv \lambda c^{\mathsf{B}_n \to \tau \to \tau} e^\tau . c\ T^n\ (c\ F^n\ e) : \Delta_1^n \\ \mathbf{D}_{k+1}^n &\equiv powerset\ \mathbf{D}_k^n : \Delta_{k+2}^n \end{aligned}$$

where

$$\begin{aligned} powerset \equiv\ & \lambda A^{*(\alpha \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \to \alpha^{**}} . \\ & A^*\ double\ (\lambda c^{\alpha^* \to \tau \to \tau} b^\tau . c\ (\lambda c'^{\alpha \to \tau \to \tau} b'^\tau . b')\ b) \\ & : ((\alpha \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \\ double \equiv\ & \lambda x^\alpha\ l^{(\alpha^* \to \tau \to \tau) \to \tau \to \tau}\ c^{\alpha^* \to \tau \to \tau}\ b^\tau . \\ & \qquad l(\lambda e^{\alpha^*} . c\ (\lambda c'^{\alpha \to \tau \to \tau}\ b'^\tau . c'\ \underline{x}\ (e\ c'\ b')))(l\ c\ b) \\ & : \alpha \to \alpha^{**} \to \alpha^{**} \end{aligned}$$

In all these terms, the only variable occurrence that is potentially unsafe is the underlined occurrence x in *double*. This occurrence is safely bound just when $\mathsf{ord}(\alpha) \geq \mathsf{ord}(\tau)$. Consequently for all $k, n \geq 0$, $\mathbf{D}_k^n$ is safe if and only if $\mathsf{ord}(\alpha) \geq \mathsf{ord}(\tau)$.

3.2.3. *Quantifier elimination.* Terms of type Δ_{k+1}^n are now used as iterators over list of elements of type Δ_k^n and we set $\tau \equiv \mathsf{B}_n$ in the type Δ_{k+1}^n in order to iterate a level-n Boolean function. Since $\mathsf{ord}(\Delta_k^n) \geq \mathsf{ord}(\mathsf{B}_n)$ for all n, all the instantiations of the terms $\mathbf{D}_k^n$ will be safe. Following [23], quantifier elimination interprets the formula $\forall x^k . \Phi(x^k)$ as the iterated conjunction: $\mathbf{C}_0^{n \mapsto 0}\left(\mathbf{D}_k^n(\lambda x^k : \Delta_k^n . AND^n(\hat{\Phi}\ x^k))\ T^n\right)$ where $\hat{\Phi}$ is the interpretation of Φ and n is the representation level chosen for the variable x^k; similarly $\exists x^k . \Phi(x^k)$ is interpreted by the iterated disjunction $\mathbf{C}_0^{n \mapsto 0}\left(\mathbf{D}_k^n(\lambda x^k : \Delta_k^n . AND^n(\hat{\Phi}\ x^k))\ T^n\right)$.

Let $x_p^{k_p} \ldots x_1^{k_1}$ for $p \geq 1$ be the list of variables appearing in the formula. W.l.o.g. we can assume that they are given in the order of appearance of their binder in the formula *i.e.*

$x_p^{k_p}$ is bound by the leftmost binder. We fix the domain of representation of each variable as follows. The right-most variable $x_1^{k_1}$ will be encoded in the domain $\Delta^0_{k_1}$; suppose that for $1 \leq i < p$ the domain of representation of $x_i^{k_i}$ is $\Delta^l_{k_l}$ then the domain of representation of $x_{i+1}^{k_{i+1}}$ is defined as $\Delta^{l'}_{k_{i+1}}$ where l' is the smallest natural number such that $\mathsf{ord}(\Delta^{l'}_{k_{i+1}})$ is strictly greater than $\mathsf{ord}(\Delta^l_{k_i})$.

This way, since variables that are bound first have higher order, the variables that are bound in the nested list-iterations (corresponding to the nested quantifiers in the formula) are necessarily safely bound.

Example 3.1. The formula $\forall x^0.\exists y^0.x^0 \vee y^0$, which is encoded by an unsafe term in Mairson's encoding, is represented in our encoding by the safe term:

$$\vdash_{\mathsf{s}} \mathbf{C}_0^{1\mapsto 0}\left(\mathbf{D}_0^1\ (\lambda x^0 : \Delta_0^1.AND^0(\mathbf{D}_0^0\ (\lambda y^0 : \Delta_0^0.OR^0(OR^0\ (\mathbf{C}_0^{1\mapsto 0}\ x^0)\ y^0))\ F^0))\ T^1\right) : \mathsf{B}_0\ .$$

3.2.4. *Set-membership.* To complete the interpretation of prime formulae, we would need to show how to encode set membership. The use of multiple domains of representation does not suffice to turn Mairson's encoding into a safe term. We would further need to have a version of the Booleans conversion term $\mathbf{C}_0^{n+1\mapsto n'}$ generalized to higher-order sets. This transformation can be interpreted as the simply-typed term:

$$\mathbf{C}_{k+1}^{n\mapsto n'} \equiv \lambda m^{\Delta^n_{k+1}} u^{\Delta^n_k \to \tau \to \tau} v^\tau.m(\lambda z^{\Delta^n_k} w^\tau.\underline{u(\underline{\mathbf{C}_k^{n\mapsto n'} z})w})v : \Delta^n_{k+1} \to \Delta^{n'}_{k+1}$$

Unfortunately this term is safe if and only if $n = n'$ (the largest underlined subterm is safe just when $n \geq n'$ and the other underline subterm is safe just when $n' \geq n$), thus making this transformation useless in the safe lambda calculus.

This leads us to conjecture that the set-membership test function is intrinsically unsafe.

If $\mathbf{C}_{k+1}^{n\mapsto n'}$ were safely representable then the encoding would go as follows: We set $\tau \equiv \mathsf{B}_0$ in the types Δ^n_{k+1} for all $n, k \geq 0$ in order to iterate a level-0 Boolean function. Firstly, the formulae "$\mathbf{true} \in y^1$" and "$\mathbf{false} \in y^1$" can be encoded by the safe terms $y^1(\lambda x^0.OR^0\ x^0)F^0$ and $y^1(\lambda x^0.OR^0(NOT^0\ x^0))F^0$ respectively. For the general case "$x^k \in y^{k+1}$" we proceed as in [23] by introducing lambda-terms encoding set equality, set membership and subset tests, and we further parameterize these encoding by $n \in \mathbb{N}$.

$$\begin{aligned}
member^{n+1}_{k+1} &\equiv \lambda x^{\Delta^{n+1}_k} y^{\Delta^{n+1}_{k+1}}.(\mathbf{C}^{n+1\mapsto n}_{k+1}\ y)\ (\lambda z^{\Delta^n_k}.OR^0(eq^n_k\ (\mathbf{C}^{n+1\mapsto n}_k\ x)\ z))\ F^0 \\
&: \Delta^{n+1}_k \to \Delta^{n+1}_{k+1} \to \mathsf{B}_0 \\
subset^n_{k+1} &\equiv \lambda x^{\Delta^n_{k+1}} y^{\Delta^n_{k+1}}.x\ (\lambda x^{\Delta^n_k}.AND^0(member^n_{k+1}\ x\ y))\ T^0 \\
&: \Delta^n_{k+1} \to \Delta^n_{k+1} \to \mathsf{B}_0 \\
eq^n_0 &\equiv \lambda x^{\mathsf{B}_n} \lambda y^{\mathsf{B}_n}.\mathbf{C}^{n\mapsto 0}_0\ (OR^n(AND^n\ x\ y)(AND^n(NOT^n\ x)(NOT^n\ y))) \\
&: \mathsf{B}_n \to \mathsf{B}_n \to \mathsf{B}_0 \\
eq^n_{k+1} &\equiv \lambda x^{\Delta^n_{k+1}}\ y^{\Delta^n_{k+1}}.(\lambda op^{\Delta^n_{k+1}\to\Delta^n_{k+1}\to\mathsf{B}_0}.AND^0(op\ x\ y)(op\ y\ x))\ subset^n_{k+1} \\
&: \Delta^n_{k+1} \to \Delta^n_{k+1} \to \mathsf{B}_0
\end{aligned}$$

The variables in the definition of eq^n_{k+1} and $subset^n_{k+1}$ are safely bounds. Moreover, the occurrence of x in $member^{n+1}_{k+1}$ is now safely bound (this was not the case in Mairson's

original encoding) thanks to the fact that the representation domain of z is lower than that of x. The formula $x^k \in y^{k+1}$ can then be encoded as

$$x : \Delta_k^n, y : \Delta_{k+1}^{n'} \vdash_{\mathsf{st}} member_{k+1}^u \ (\mathbf{C}_k^{n \mapsto u} \ x) \ (\mathbf{C}_{k+1}^{n' \mapsto u} \ y) : \mathsf{B}_0$$

for some $n, n' \geq 2$ and $u = \min(n, n') + 1$.

Unfortunately, this encoding is not completely safe because it uses the unsafe conversion terms $\mathbf{C}_k^{n \mapsto n'}$ for $k \geq 1$.

3.3. **PSPACE-hardness.** We observe that instances of the True Quantified Boolean Formulae satisfaction problem (TQBF) are special instances of the decision problem for finite type theory. These instances corresponds to formulae in which set membership is not allowed and variables are all taken from the base domain $\mathcal{D}_0$. As we have shown in the previous section, such restricted formulae can be safely encoded in the safe lambda calculus. Therefore since TQBF is PSPACE-complete, deciding $\beta\eta$-equality of two terms of the safe lambda calculus is PSPACE-hard.

Example 3.2. Using the encoding where τ is set to B_0 in the types Δ_k^n for all $k, n \geq 0$, the formula $\forall x \exists y \exists z (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$ is represented by the safe term:

$$\begin{aligned}
\vdash_{\mathsf{s}} \ & \mathbf{D}_0^2(\lambda x^{\mathsf{B}_2}.AND^0 \\
& \quad (\mathbf{D}_0^1(\lambda y^{\mathsf{B}_1}.OR^0 \\
& \quad\quad (\mathbf{D}_0^0(\lambda z^{\mathsf{B}_0}.OR^0 \\
& \quad\quad\quad (AND^0(OR^0(OR^0 \ (\mathbf{C}_0^{2 \mapsto 0} \ x) \ (\mathbf{C}_0^{1 \mapsto 0} \ y))z) \\
& \quad\quad\quad\quad (OR^0(OR^0(NEG^0(\mathbf{C}_0^{2 \mapsto 0} \ x))(NEG^0(\mathbf{C}_0^{1 \mapsto 0} \ y)))(NEG^0 \ z))) \\
& \quad\quad)F^0) \\
& \quad)F^0) \\
& \quad)T^0 \\
& : \mathsf{B}_0
\end{aligned}$$

Remark 3.3. Since the Boolean satisfaction problem (SAT) can be reduced to TQBF (where formulae are restricted to use only existential quantifiers), the safe lambda calculus is also NP-hard. In [6], Asperti gave an interpretation of SAT in the simply-typed lambda calculus but his encoding relies on unsafe terms.

3.4. **Other complexity results.**

3.4.1. *Better lower bound?* Since the safety condition restricts the expressivity of the lambda calculus in a non-trivial way, one can reasonably expect the beta-eta equality problem (where types are not restricted) to have a lower complexity in the safe case than in the normal case. Our failed attempt to encode type theory in the safe lambda calculus suggests that the non-elementary lower bounds that holds in the simply-typed lambda calculus no longer applies in the safe lambda calculus. (Nevertheless, one may not rule out the possibility that another non recursive problem may be encodable in the safe lambda calculus.)

We have shown that the problem is PSPACE-hard but this is probably a coarse lower bound. It would be interesting to know whether it is also EXPTIME-hard.

3.4.2. *Upper bound.* At present, no upper bound is known for the equivalence problem for safe terms.

3.4.3. *Beta-eta equivalence for terms limited to a finite set of types.* Statman showed [36] that there exists a finite set of types such that the beta-eta equivalence problem restricted to terms of these types is PSPACE-hard.

The picture is different in the safe lambda calculus since our encoding of TQBF requires the full type hierarchy. It was indeed necessary to introduce variables of higher-order in order to eliminate 'unsafety'. Consequently, we had to use simple types of unbounded order (the order is linear in the size of the QBF formula). We suspect the decidability problem for safe terms restricted to any finite set of types to have a complexity lower than PSPACE.

3.4.4. *Normalization.* The *normalization problem* is: given a term M, what is its β-normal form? This problem is non-elementary even when restricted to safe terms as the following example shows. Let $\tau_{-2} \equiv o$ and for $n \geq -1$, $\tau_n \equiv \tau_{n-1} \to \tau_{n-1}$. For $k, n \in \mathbb{N}$ we write $\overline{k}^n$ to denote the kth Church Numeral parameterized by n as follows:

$$\overline{k}^n \equiv \lambda s^{\tau_{n-1}} z^{\tau_{n-2}}. \overbrace{s(\ldots(s(s}^{k \text{ times}} z)\ldots) : \tau_n \ .$$

Then for $n \geq 1$, the safe term $\overline{2}^{n-1}\ \overline{2}^{n-2} \ldots \overline{2}^0$ of type τ_0 has length $\mathcal{O}(n)$ whereas its normal form $\overline{\exp_n(1)}^0$ has length $\mathcal{O}(\exp_n(1))$.

Statman's result shows that in the simply-typed lambda calculus, the beta-eta equality problem is essentially as hard as the normalization problem i.e. they are both non-elementary. It is not known whether this is still the case in the safe lambda calculus. In particular, it may be the case that the beta-eta equivalence problem is elementary although we know that the normalization problem is not.

3.4.5. *The beta-reduction problem.* The *beta-reduction problem* is related to the beta-eta equivalence problem. It can be stated as follows: given a term M_1 in β-normal form and a term M_2 (possibly containing redexes), does M_2 β-reduces to M_1?

In [33], Schubert gave a PSPACE algorithm to decide the β-reduction problem for order-3 lambda terms. Since order-3 terms are sufficient to encode TQBF in the lambda calculus, Schubert shows that the problem is PSPACE-complete. No complexity result is known for restrictions of this problem to terms of order greater than 3. A natural question to ask is whether complexity characterizations can be obtained when restricting the problem to safe terms.

4. A game-semantic account of safety

Our aim is to characterize safety by game semantics. We shall assume that the reader is familiar with the basics of game semantics; For an introduction, we recommend [3]. Recall that a *justified sequence* over an arena is an alternating sequence of O-moves and P-moves such that every move m, except the opening move, has a pointer to some earlier occurrence of the move m_0 such that m_0 enables m in the arena. A *play* is just a justified sequence that satisfies Visibility and Well-Bracketing. A basic result in game semantics is that λ-terms are denoted by *innocent strategies*, which are strategies that depend only on the *P-view* of

a play. The main result (Theorem 4.11) of this section is that if a λ-term is safe, then its game semantics (is an innocent strategy that) is, what we call, *P-incrementally justified*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and pointers from the O-moves therein: specifically a P-question always points to the last pending O-question (in the P-view) of a greater order.

The proof of Theorem 4.11 depends on a Correspondence Theorem (see the Appendix) that relates the strategy denotation of a λ-term M to the set of *traversals* over a souped-up abstract syntax tree of the η-long form of M. In the language of game semantics, traversals are just (concrete representations of) the *uncovering* (in the sense of Hyland and Ong [18]) of plays in the strategy denotation.

The useful transference technique between plays and traversals was originally introduced by one of us [30] for studying the decidability of monadic second-order theories of infinite structures generated by higher-order grammars (in which the Σ-constants or terminal symbols are at most order 1, and *uninterpreted*). In the Appendix, we present an extension of this framework to the general case of the simply-typed lambda calculus with free variables of any order. A new traversal rule is introduced to handle nodes labelled with free variables. Also new nodes are added to the computation tree to account for the answer moves of the game semantics, thus enabling the framework to model languages with interpreted constants such as PCF (by adding traversal rules to handle constant nodes).

Incrementally-bound computation tree. In [30] the computation tree of a grammar is defined as the unravelling of a finite graph representing the *long transform* of a grammar. Similarly we define the computation tree of a λ-term as an abstract syntax tree of its η-long normal form. We write $l\langle t_1, \ldots, t_n\rangle$ with $n \geq 0$ to denote the ordered tree with a root labelled l with n child-subtrees $t_1, \ldots, t_n$. In the following we consider arbitrary simply-typed terms.

Definition 4.1. The ***computation tree*** $\tau(M)$ of a simply-typed term $\Gamma \vdash_{\mathsf{st}} M : T$ with variable names in a countable set $\mathcal{V}$ is a tree with labels in

$$\{@\} \cup \mathcal{V} \cup \{\lambda x_1 \ldots x_n \mid x_1, \ldots, x_n \in \mathcal{V}, n \in \mathbb{N}\}$$

defined from its η-long form as follows. Suppose $\overline{x} = x_1 \ldots x_n$ for $n \geq 0$ then

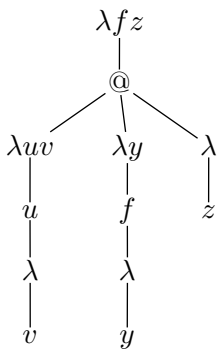
$$\begin{array}{rcl} \text{for } m \geq 0,\ z \in \mathcal{V}\text{: } \tau(\lambda\overline{x}.z s_1 \ldots s_m : o) & = & \lambda\overline{x}\langle z\langle \tau(s_1), \ldots, \tau(s_m)\rangle\rangle \\ \text{for } m \geq 1\text{: } \tau(\lambda\overline{x}.(\lambda y.t) s_1 \ldots s_m : o) & = & \lambda\overline{x}\langle @\langle \tau(\lambda y.t), \tau(s_1), \ldots, \tau(s_m)\rangle\rangle\ . \end{array}$$

Example 4.2. Take $\vdash_{\mathsf{st}} \lambda f^{o\to o}.(\lambda u^{o\to o}.u) f : (o \to o) \to o \to o$.

Its η-long normal form is:

$$\begin{array}{l} \vdash_{\mathsf{st}} \lambda f^{o\to o} z^{o}. \\ \qquad (\lambda u^{o\to o} v^{o}.u(\lambda.v)) \\ \qquad (\lambda y^{o}.fy) \\ \qquad (\lambda.z) \\ : (o \to o) \to o \to o \end{array}$$

Its computation tree is:

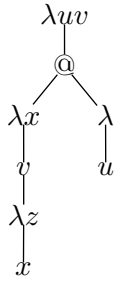


Example 4.3. Take $\vdash_{\mathsf{st}} \lambda u^o v^{((o\to o)\to o)}.(\lambda x^o.v(\lambda z^o.x))u : o \to ((o \to o) \to o) \to o$.

Its η-long normal form is:

$$\begin{array}{l}\vdash_{\mathsf{st}} \lambda u^o v^{((o\to o)\to o)}.\\ \qquad (\lambda x^o.v(\lambda z^o.x))u\\ : o \to ((o \to o) \to o) \to o\end{array}$$

Its computation tree is:



Even-level nodes are λ-nodes (the root is on level 0). A single λ-node can represent several consecutive variable abstractions or it can just be a *dummy lambda* if the corresponding subterm is of ground type. Odd-level nodes are variable or application nodes.

The ***order*** of a node n, written $\mathsf{ord}(n)$, is defined as follows: @-nodes have order 0. The order of a variable-node is the type-order of the variable labelling it. The order of the root node is the type-order of $(A_1, \ldots, A_p, T)$ where $A_1, \ldots, A_p$ are the types of the variables in the context Γ. Finally, the order of a lambda node different from the root is the type-order of the term represented by the sub-tree rooted at that node.

We say that a variable node n labelled x is ***bound*** by a node m, and m is called the ***binder*** of n, if m is the closest node in the path from n to the root such that m is labelled $\lambda\overline{\xi}$ with $x \in \overline{\xi}$.

We introduce a class of computation trees in which the binder node is uniquely determined by the nodes' orders:

Definition 4.4. A computation tree is ***incrementally-bound*** if for all variable node x, either x is *bound* by the first λ-node in the path to the root with order $> \mathsf{ord}(x)$, or x is a *free variable* and all the λ-nodes in the path to the root except the root have order $\leq \mathsf{ord}(x)$.

Proposition 4.5 (Safety and incremental-binding)**.**

(i) If M is safe then $\tau(M)$ is incrementally-bound.

(ii) Conversely, if M is a closed *simply-typed term and $\tau(M)$ is incrementally-bound then M is safe.*

Proof. (i) Suppose that M is safe. By Lemma 1.23 the η-long form of M is safe therefore $\tau(M)$ is the tree representation of a safe term.

In the safe lambda calculus, the variables in the context with the lowest order must be all abstracted at once when using the abstraction rule. Since the computation tree merges consecutive abstractions into a single node, any variable x occurring free in the subtree rooted at a node $\lambda\overline{\xi}$ different from the root must have order greater or equal to $\mathsf{ord}(\lambda\overline{\xi})$. Conversely, if a lambda node $\lambda\overline{\xi}$ binds a variable node x then $\mathsf{ord}(\lambda\overline{\xi}) = 1+\max_{z\in\overline{\xi}}\mathsf{ord}(z) > \mathsf{ord}(x)$.

Let x be a bound variable node. Its binder occurs in the path from x to the root, therefore, according to the previous observation, x must be bound by the first λ-node occurring in this path with order $> \mathsf{ord}(x)$. Let x be a free variable node then x is not bound by any of the λ-nodes occurring in the path to the root. Once again, by the previous observation, all these λ-nodes except the root have order smaller than $\mathsf{ord}(x)$. Hence τ is incrementally-bound.

(ii) Let M be a closed term such that $\tau(M)$ is incrementally-bound. W.l.o.g. we can assume that M is in η-long form. We prove that M is safe by induction on its structure. The base case $M = \lambda\overline{\xi}.x$ for some variable x is trivial. *Step case:* If $M = \lambda\overline{\xi}.N_1 \dots N_p$. Let i range over $1..p$. We have $N_i \equiv \lambda\overline{\eta_i}.N_i'$ for some non-abstraction term N_i'. By the induction hypothesis, $\lambda\overline{\xi}.N_i = \lambda\overline{\xi}\,\overline{\eta_i}.N_i'$ is a safe closed term, and consequently N_i' is necessarily safe. Let z be a free variable of N_i' not bound by $\lambda\overline{\eta_i}$ in N_i. Since $\tau(M)$ is incrementally-bound we have $\mathsf{ord}(z) \geq \mathsf{ord}(\lambda\overline{\eta_1}) = \mathsf{ord}(N_i)$, thus we can abstract the variables $\overline{\eta_1}$ using (abs) which shows that N_i is safe. Finally we conclude $\vdash_{\mathsf{s}} M = \lambda\overline{\xi}.N_1 \dots N_p : T$ using the rules (app) and (abs). □

The assumption that M is closed is necessary. For instance for $x, y : o$, the computation trees $\tau(\lambda xy.x)$ and $\tau(\lambda y.x)$ are both incrementally-bound but $\lambda xy.x$ is safe and $\lambda y.x$ is not.

P-incrementally justified strategy. We now consider the game-semantic model of the simply-typed lambda calculus. The strategy denotation of a term-in-context $\Gamma \vdash_{\mathsf{st}} M : T$ is written $[\![\Gamma \vdash_{\mathsf{st}} M : T]\!]$. We define the ***order*** of a move m, written $\mathsf{ord}(m)$, to be the length of the path from m to its furthest leaf in the arena minus 1. (There are several ways to define the order of a move; the definition chosen here is sound in the current setting where each question move in the arena enables at least one answer move.)

Definition 4.6. A strategy σ is said to be ***P-incrementally justified*** if for any play $s\,q \in \sigma$ where q is a P-question, q points to the last unanswered O-question in $\ulcorner s \urcorner$ with order strictly greater than $\mathsf{ord}(q)$.

Note that although the pointer is determined by the P-view, the choice of the move itself can be based on the whole history of the play. Thus P-incremental justification does not imply innocence.

The definition suggests an algorithm that, given a play of a P-incrementally justified denotation, uniquely recovers the pointers from the underlying sequence of moves and from the pointers associated to the O-moves therein. Hence:

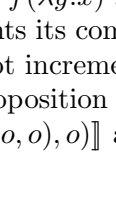
Lemma 4.7. *In P-incrementally justified strategies, pointers emanating from P-moves are superfluous.*

Example 4.8. Copycat strategies, such as the identity strategy id_A on game A or the evaluation map $ev_{A,B}$ of type $(A \Rightarrow B) \times A \to B$, are all P-incrementally justified.[7]

The Correspondence Theorem 5.10 gives us the following equivalence:

Proposition 4.9. *Let $\Gamma \vdash_{\mathsf{st}} M : T$ be a β-normal term. The computation tree $\tau(M)$ is incrementally-bound if and only if $[\![\Gamma \vdash_{\mathsf{st}} M : T]\!]$ is P-incrementally justified.*

Example 4.10. Consider the β-normal term $\Gamma \vdash_{\mathsf{st}} f(\lambda y.x) : o$ where $y : o$ and $\Gamma = f : ((o,o),o),\ x : o$. The figure on the right represents its computation tree with the node orders given as superscripts. The node x is not incrementally-bound therefore $\tau(f(\lambda y.x))$ is not incrementally-bound and by Proposition 4.9, $[\![\Gamma \vdash_{\mathsf{st}} f(\lambda y.x) : o]\!]$ is not incrementally-justified (although $[\![\Gamma \vdash_{\mathsf{st}} f : ((o,o),o)]\!]$ and $[\![\Gamma \vdash_{\mathsf{st}} \lambda y.x : (o,o)]\!]$ are).



[7] In such strategies, a P-move m is justified as follows: either m points to the preceding move in the P-view or the preceding move is of smaller order and m is justified by the second last O-move in the P-view.

Propositions 4.5 and 4.9 allow us to show the following:

Theorem 4.11 (Safety and P-incremental justification)**.**

(i) If $\Gamma \vdash_{\mathsf{s}} M : T$ *then* $[\![\Gamma \vdash_{\mathsf{s}} M : T]\!]$ *is P-incrementally justified.*

(ii) If $\vdash_{\mathsf{st}} M : T$ *is a closed simply-typed term and* $[\![\vdash_{\mathsf{st}} M : T]\!]$ *is P-incrementally justified then the* β*-normal form of* M *is safe.*

Proof. (i) Let M be a safe simply-typed term. By Lemma 1.15, its β-normal form M' is also safe. By Proposition 4.5(i), $\tau(M')$ is incrementally-bound and by Proposition 4.9, $[\![M']\!]$ is an incrementally-justified. Finally the soundness of the game model gives $[\![M]\!] = [\![M']\!]$. (ii) is a consequence of Lemma 1.15, Proposition 4.9 and 4.5(ii) and soundness of the game model. □

Putting Theorem 4.11(i) and Lemma 4.7 together gives:

Proposition 4.12. *In the game semantics of safe* λ*-terms, pointers emanating from P-moves are unnecessary i.e. they are uniquely recoverable from the underlying sequences of moves and from O-moves' pointers.*

In fact, as the last example highlights, pointers are superfluous at order 3 for safe terms whether from P-moves or O-moves. This is because for question moves in the first two levels of an arena (initial moves being at level 0), the associated pointers are uniquely recoverable thanks to the visibility condition. At the third level, the question moves are all P-moves therefore their associated pointers are uniquely recoverable by P-incremental justification. This is not true anymore at order 4: Take the safe term $\psi : (((o^4, o^3), o^2), o^1) \vdash_{\mathsf{s}} \psi(\lambda\varphi.\varphi a) : o^0$ for some constant $a : o$, where $\varphi : (o, o)$. Its strategy denotation contains plays whose underlying sequence of moves is $q_0\, q_1\, q_2\, q_3\, q_2\, q_3\, q_4$. Since q_4 is an O-move, it is not constrained by P-incremental justification and thus it can point to any of the two occurrences of q_3.[8]

Towards a fully abstract game model.

The standard game models which have been shown to be fully abstract for PCF [2, 18] are of course also fully abstract for the restricted language safe PCF. One may ask, however, whether there exists a fully abstract model with respect to safe context only.

Such model may be obtain by considering P-incrementally justified strategies - which have been shown to compose in [7]. Its is reasonable to think that O-moves also needs to be constrained by the symmetrical O-incremental justification, which corresponds to the requirement that contexts are safe. This line of work is still in progress.

[8]More generally, a P-incrementally justified strategy can contain plays that are not "O-incrementally justified" since it must take into account any possible strategy incarnating its context, including those that are not P-incrementally justified. For instance in the given example, there is one version of the play that is not O-incrementally justified (the one where q_4 points to the first occurrence of q_3). This play is involved in the strategy composition $[\![\vdash_{\mathsf{st}} M_2 : (((o, o), o), o)]\!]; [\![\psi : (((o, o), o), o) \vdash_{\mathsf{st}} \psi(\lambda\varphi.\varphi a) : o]\!]$ where M_2 denotes the unsafe Kierstead term.

Safe PCF and safe Idealised Algol. PCF is the simply-typed lambda calculus augmented with basic arithmetic operators, if-then-else branching and a family of recursion combinator $Y_A : ((A, A), A)$ for any type A. We define *safe* PCF to be PCF where the application and abstraction rules are constrained in the same way as the safe lambda calculus. This language inherits the good properties of the safe lambda calculus: No variable capture occurs when performing substitution and safety is preserved by the reduction rules of the small-step semantics of PCF.

4.0.6. *Correspondence.* The computation tree of a PCF term is defined as the least upper-bound of the chain of computation trees of its *syntactic approximants* [3]. It is obtained by infinitely expanding the Y combinator, for instance $\tau(Y(\lambda fx.fx))$ is the tree representation of the η-long form of the infinite term $(\lambda fx.fx)((\lambda fx.fx)((\lambda fx.fx)(\ldots$

It is straightforward to define the traversal rules modeling the arithmetic constants of PCF. Just as in the safe lambda calculus we had to remove @-nodes in order to reveal the game-semantic correspondence, in safe PCF it is necessary to filter out the constant nodes from the traversals. The Correspondence Theorem for PCF says that the interaction game semantics is isomorphic to the set of traversals disposed of these superfluous nodes. This can easily be shown for term approximants. It is then lifted to full PCF using the continuity of the function $\mathcal{T}rv(\_)^{\restriction\circledast}$ from the set of computation trees (ordered by the approximation ordering) to the set of sets of justified sequences of nodes (ordered by subset inclusion). Finally computation trees of safe PCF terms are incrementally-bound thus we have

Theorem 4.13. *Safe PCF terms have P-incrementally justified denotations.* □

Similarly, we can define safe IA to be safe PCF augmented with the imperative features of Idealized Algol (IA for short) [32]. Adapting the game-semantic correspondence and safety characterization to IA seems feasible although the presence of the base type `var`, whose game arena $\mathtt{com}^{\mathbb{N}} \times \mathtt{exp}$ has infinitely many initial moves, causes a mismatch between the simple tree representation of the term and its game arena. It may be possible to overcome this problem by replacing the notion of computation tree by a "computation directed acyclic graph".

The possibility of representing plays *without some or all of their pointers* under the safety assumption suggests potential applications in algorithmic game semantics. Ghica and McCusker [15] were the first to observe that pointers are unnecessary for representing plays in the game semantics of the second-order finitary fragment of Idealized Algol (IA_2 for short). Consequently observational equivalence for this fragment can be reduced to the problem of equivalence of regular expressions. At order 3, although pointers are necessary, deciding observational equivalence of IA_3 is EXPTIME-complete [29, 28]. Restricting the problem to the safe fragment of IA_3 may lead to a lower complexity.

5. Further work and open problems

The safe lambda calculus is still not well understood. Many basic questions remain. What is a (categorical) model of the safe lambda calculus? Does the calculus have interesting models? What kind of reasoning principles does the safe lambda calculus support, via the Curry-Howard Isomorphism? Does the safe lambda calculus characterize a complexity class, in the same way that the simply-typed lambda calculus characterizes the

polytime-computable numeric functions [21]? Is the addition of unsafe contexts to safe ones conservative with respect to observational (or contextual) equivalence?

With a view to algorithmic game semantics and its applications, it would be interesting to identify sublanguages of Idealised Algol whose game semantics enjoy the property that pointers in a play are uniquely recoverable from the underlying sequence of moves. We name this class PUR. IA_2 is the paradigmatic example of a PUR-language. Another example is *Serially Re-entrant Idealized Algol* [1], a version of IA where multiple uses of arguments are allowed only if they do not "overlap in time". We believe that a PUR language can be obtained by imposing the *safety condition* on IA_3. Murawski [27] has shown that observational equivalence for IA_4 is undecidable; is observational equivalence for *safe* IA_4 decidable?

Acknowledgment. We thank Ugo dal Lago for the insightful discussions we had during his visit at the Oxford University Computing Laboratory in March 2008.

References

[1] S. Abramsky. Semantics via game theory. In *Marktoberdorf International Summer School*, 2001. Lecture slides.
[2] S. Abramsky, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994.
[3] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School*. Springer-Verlag, 1998. Lecture notes.
[4] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. Technical report, University of Oxford, 2004.
[5] A. V. Aho. Indexed grammars – an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.
[6] A. Asperti. P = NP, up to sharing.
[7] W. Blum. *The Safe Lambda Calculus*. PhD thesis, University of Oxford. forthcoming.
[8] W. Blum and C.-H. L. Ong. The safe lambda calculus. In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2007.
[9] D. Caucal. On infinite terms having a decidable monadic theory. pages 165–176, 2002.
[10] W. Damm. The IO- and OI-hierarchy. *TCS*, 20:95–207, 1982.
[11] W. Damm and A. Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Information and Control*, 71(1-2):1–32, 1986.
[12] J. G. de Miranda. *Structures generated by higher-order grammars and the safety constraint.* Dphil thesis, University of Oxford, 2006.
[13] A. Dimovski, D. R. Ghica, and R. Lazic. Data-abstraction refinement: A game semantic approach. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *LNCS*, pages 102–117. Springer, 2005.
[14] S. Fortune, D. Leivant, and M. O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, 1983.
[15] D. R. Ghica and G. McCusker. Reasoning about idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, volume 1853 of *LNCS*, pages 103–116. Springer-Verlag, 2000.
[16] W. Greenland. *Game Semantics for Region Analysis.* PhD thesis, University of Oxford, 2004.
[17] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursive schemes. November 2006. 13 pages, preprint.
[18] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, December 2000.
[19] T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.

[20] D. Leivant. Functions over free algebras definable in the simply typed lambda calculus. *Theor. Comput. Sci.*, 121(1&2):309–322, 1993.
[21] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. In M. Bezem and J. F. Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 1993.
[22] R. Loader. Notes on simply typed lambda calculus, February 1998.
[23] H. Mairson. A Simple Proof of a Theorem of Statman. *TCS*, 103(2):387–394, 1992.
[24] A. N. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet Math. Dokl.*, 15:1170–1174, 1974.
[25] A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12:38–43, 1976.
[26] A. Meyer. The inherent computational complexity of theories of ordered sets. In *Proc. Int'l. Cong. of Mathematicians*, volume 2, pages 477–482, August 1974.
[27] A. Murawski. On program equivalence in languages with ground-type references. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 108–117, 22-25 June 2003.
[28] A. S. Murawski and I. Walukiewicz. Third-order idealized algol with iteration is decidable. In V. Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2005.
[29] C.-H. L. Ong. An approach to deciding observational equivalence of algol-like languages. *Ann. Pure Appl. Logic*, 130(1-3):125–171, 2004.
[30] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science.* Computer Society Press, 2006. Extended abstract.
[31] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes (technical report). Preprint, 42 pp, 2006.
[32] J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland, Amsterdam, 1981.
[33] A. Schubert. The complexity of beta-reduction in low orders. *Proceedings TLCA 2001*, pages 400–414.
[34] H. Schwichtenberg. Definierbare funktionen im lambda-kalkul mit typen. *Archiv Logik Grundlagenforsch*, 17:113–114, 1976.
[35] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, July 1979.
[36] R. Statman. The typed lambda -calculus is not elementary recursive. 9(1):73–81, July 1979.
[37] M. Zaionc. Word operation definable in the typed lambda-calculus. *Theor. Comput. Sci.*, 52:1–14, 1987.
[38] M. Zaionc. On the "lambda"-definable tree operations. In C. Bergman, R. D. Maddux, and D. Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *Lecture Notes in Computer Science*, pages 279–292. Springer, 1988.
[39] M. Zaionc. On the "lambda"-definable tree operations. pages 279–292, 1990.
[40] M. Zaionc. Lambda-definability on free algebras. *Ann. Pure Appl. Logic*, 51(3):279–300, 1991.
[41] M. Zaionc. Lambda representation of operations between fifferent term algebras. pages 91–105, 1995.

Appendix – Computation tree, traversals and correspondence

In [30], one of us introduced the notion of computation tree and traversals over a computation tree for the purpose of studying trees generated by higher-order recursion scheme. Here we extend these concepts to the simply-typed lambda calculus. Our setting allows the presence of free variables of any order and the term studied is not required to be of ground type. (This contrasts with [30]'s setting where the term is of ground type and contains only *uninterpreted constant*. Note that we automatically account for the presence of uninterpreted constants since they can just be regarded as free variables. We will then state the *Correspondence Theorem* (Theorem 5.10) that was used in Sec. 4.

In the following we fix a simply-typed term-in-context $\Gamma \vdash_{\mathsf{st}} M : T$ (not necessarily safe) and we consider its computation tree $\tau(M)$ as defined in Def. 4.1.

5.1. **Notations.** We first fix some notations. We write $\circledast$ to denote the root of the computation tree $\tau(M)$. The set of nodes of the computation tree is denoted by N. The sets $N_@$, N_λ and N_{var} are respectively the subset of @-nodes, λ-nodes and variable nodes. For $n \in N$ we will write $\kappa(n)$ to denote the subterm of $\lceil M \rceil$ corresponding to the subtree of $\tau(M)$ rooted at n. In particular $\kappa(\circledast) = \lceil M \rceil$. The ***type*** of a variable-labelled node is the type of the variable that labels it; the type of the root is $(A_1, \ldots, A_p, T)$ where $x_1 : A_1, \ldots, x_p : A_p$ are the variables in the context Γ; and the type of a node $n \in (N_\lambda \cup N_@) \setminus \{\circledast\}$ is the type of the corresponding subterm $\kappa(n)$.

5.2. **Pointers and justified sequences of nodes.** We define the ***enabling relation*** on the set of nodes of the computation tree as follows: m enables n, written $m \vdash n$, if and only if n is bound by m (and we sometimes write $m \vdash_i n$ to indicate that n is the i^{th} variable bound by m); or m is the root $\circledast$ and n is a free variable; or n is a λ-node and m is its parent node.

We say that a node n_0 of the computation tree is ***hereditarily enabled*** by $n_p \in N$ if there are nodes $n_1, \ldots, n_{p-1} \in N$ such that n_{i+1} enables n_i for all $i \in 0..p-1$.

For any set of nodes $S, H \subseteq N$ we write $S^{H\vdash}$ for $\{n \in S \mid \exists n_0 \in H \text{ s.t. } n_0 \vdash^* n\}$ – the subset of S consisting of nodes hereditarily enabled by some node in H. We will abbreviate $S^{\{n_0\}\vdash}$ into $S^{n_0\vdash}$.

We call ***input-variables nodes*** the elements of $V_{\mathsf{var}}^{\circledast\vdash}$ *i.e.* variables that are hereditarily enabled by the root of $\tau(M)$. Thus we have $V_{\mathsf{var}}^{\circledast\vdash} = V \setminus (V_{\mathsf{var}}^{N_@\vdash} \cup V_{\mathsf{var}}^{N_\Sigma\vdash})$.

A ***justified sequence of nodes*** is a sequence of nodes with pointers such that each occurrence of a variable or λ-node n different from the root has a pointer to some preceding occurrence m verifying $m \vdash n$. In particular, occurrences of @-nodes do not have pointer. We represent the pointer in the sequence as follows $m \overset{i}{\ldots} n.$ where the label indicates that either n is labelled with the ith variable abstracted by the λ-node m or that n is the i^{th} child of m. Children nodes are numbered from 1 onward except for @-nodes where it starts from 0. Abstracted variables are numbered from 1 onward. The i^{th} child of n is denoted by $n.i$.

We say that a node n_0 of a justified sequence is ***hereditarily justified*** by n_p if there are occurrences $n_1, \ldots, n_{p-1}$ in the sequence such that n_i points to n_{i+1} for all $i \in 0..p-1$. For any occurrence n in a justified sequence s, we write $s \restriction n$ to denote the subsequence of s consisting of occurrences that are hereditarily justified by n.

The notion of ***P-view*** $\ulcorner t \urcorner$ of a justified sequence of nodes t is defined the same way as the P-view of a justified sequences of moves in Game Semantics:[9]

$$\begin{array}{rcl@{\qquad\qquad}rcl} \ulcorner \epsilon \urcorner & = & \epsilon & \ulcorner s \cdot m \cdot \ldots \cdot \lambda\overline{\xi} \urcorner & = & \ulcorner s \urcorner \cdot m \cdot \lambda\overline{\xi} \\ \text{for } n \notin N_\lambda, \ulcorner s \cdot n \urcorner & = & \ulcorner s \urcorner \cdot n & \ulcorner s \cdot \circledast \urcorner & = & \circledast \end{array}$$

The O-view of s, written $\llcorner s \lrcorner$, is defined dually. We will borrow the game-semantic terminology: A justified sequences of nodes satisfies ***alternation*** if for any two consecutive nodes one is a λ-node and the other is not, and ***P-visibility*** if every variable node points to a node occurring in the P-view a that point.

[9]The equalities in the definition determine pointers implicitly. For instance in the second clause, if in the left-hand side, n points to some node in s that is also present in $\ulcorner s \urcorner$ then in the right-hand side, n points to that occurrence of the node in $\ulcorner s \urcorner$.

5.3. **Computation tree with value-leaves.** We now add another ingredient to the computation tree that was not originally used in [30]. We write $\mathcal{D}$ to denote the set of values of the base type o. We add ***value-leaves*** to $\tau(M)$ as follows: For each value $v \in \mathcal{D}$ and for each node $n \in N$ we attach the child leaf v_n to n. We write V for the set of vertices of the resulting tree (*i.e.* inner nodes and leaf nodes). For \$ ranging in $\{@, \lambda, var\}$, we write $V_\$$ to denote the set of inner nodes from $N_\$$ plus the leaf-nodes whose parent is in $N_\$$ *i.e.* $V_\$ = N_\$ \cup \{v_n \mid n \in N_\$, v \in \mathcal{D}\}$.

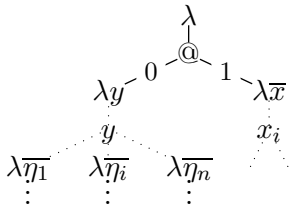
Everything that we have defined can be lifted to this new version of computation tree. A value-leaf has order 0. The enabling relation $\vdash$ is extended so that every leaf is enabled by its parent node. A link going from a value-leaf v_n to a node n is labelled by v: $n \overset{v}{\cdots} v_n$. For the definition of P-view and visibility, value-leaves are treated as λ-nodes if they are at an odd level in the computation tree, and as variable nodes if they are at an even level.

We say that an occurrence of an inner node $n \in N$ is ***answered*** by an occurrence v_n if v_n in the sequence that points to n, otherwise we say that n is ***unanswered***. The last unanswered node is called the ***pending node***. A justified sequence of nodes is ***well-bracketed*** if each value-leaf occurring in it is justified by the pending node at that point. If t is a traversal then we write $?(t)$ to denote the subsequence of t consisting only of unanswered nodes.

5.4. **Traversals of the computation tree.** A *traversal* is a justified sequence of nodes of the computation tree where each node indicates a step that is taken during the evaluation of the term.

Definition 5.1 (Traversals for simply-typed λ-terms)**.** The set $\mathcal{T}rv(M)$ of ***traversals*** over $\tau(M)$ is defined by induction over the rules of Table 1. A traversal that cannot be extended by any rule is said to be *maximal*.

A traversal always starts by visiting the root. Then it mainly follows the structure of the tree. The (Var) rule permits to jump across the computation tree. The idea is that after visiting a variable node x, a jump is allowed to the node corresponding to the subterm that would be substituted for x if all the β-redexes occurring in the term were reduced. The sequence $\lambda \cdot @ \cdot \lambda y \ldots y \cdot \lambda\overline{x} \ldots x_i \cdot \lambda\overline{\eta_i} \ldots$ is an example of traversal of the computation tree shown on the right.



Example 5.2. The following justified sequence is a traversal of the computation tree of example 4.2:

$$t = \lambda f z \cdot @ \cdot \lambda u v \cdot u \cdot \lambda y \cdot f \cdot \lambda \cdot y \cdot \lambda \cdot v \cdot \lambda \cdot z \ .$$

Proposition 5.3 (counterpart of proposition 6 from [31])**.** *Let t be a traversal. Then:*

(i) t is a well-defined and well-bracketed justified sequence;
(ii) t is a well-defined justified sequence verifying alternation, P-visibility and O-visibility;
(iii) If t's last node is not a value-leaf, then $\ulcorner t \urcorner$ is the path in the computation tree going from the root to t's last node.

Initialization rules

(Empty) $\epsilon \in \mathcal{T}rv(M)$.

(Root) The sequence constituted of a single occurrence of $\tau(M)$'s root is a traversal.

Structural rules

(Lam) If $t \cdot \lambda\overline{\xi}$ is a traversal then so is $t \cdot \lambda\overline{\xi} \cdot n$ where n denotes $\lambda\overline{\xi}$'s child and:
- If $n \in N_@ \cup N_\Sigma$ then it has no justifier;
- if $n \in N_{\mathsf{var}} \setminus N_{\mathsf{fv}}$ then it points to the only occurrence[a] of its binder in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$;
- if $n \in N_{\mathsf{fv}}$ then it points to the only occurrence of the root $\circledast$ in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$.

(App) If $t \cdot @$ is a traversal then so is $t \cdot @ \overset{0}{\cdot} n$.

Input-variable rules

(InputVar) If t is a traversal where $t^\omega \in N^{\circledast\vdash}_{\mathsf{var}} \cup L^{\circledast\vdash}_\lambda$ and x is an occurrence of a variable node in $\llcorner t \lrcorner$ then so is $t \cdot n$ for any child λ-node n of x, n pointing to x.

(InputValue) If $t_1 \cdot x \cdot t_2$ is a traversal with pending node $x \in N^{\circledast\vdash}_{\mathsf{var}}$ then so is $t_1 \cdot x \overset{v}{\cdot} t_2 \cdot v_x$ for all $v \in \mathcal{D}$.

Copy-cat rules

(Var) If $t \cdot n \cdot \lambda\overline{x} \ldots x_i \overset{i}{\cdot}$ is a traversal where $x_i \in N^{@\vdash}_{\mathsf{var}}$ then so is $t \cdot n \overset{i}{\cdot} \lambda\overline{x} \ldots x_i \overset{i}{\cdot} \lambda\overline{\eta_i}$.

(Value) If $t \cdot m \cdot n \ldots v_n \overset{v}{\cdot}$ is a traversal where $n \in N$ then so is $t \cdot m \overset{v}{\cdot} n \ldots v_n \overset{v}{\cdot} v_m$.

Table 1: Traversal rules for the simply-typed λ-calculus.

[a] Prop. 5.3 shows that P-views are paths in the tree thus n's enabler occurs exactly once in the P-view.

The ***reduction*** of a traversal t is the subsequence of t obtained by keeping only occurrences of nodes that are hereditarily enabled by the root $\circledast$. This has the effect of eliminating the "internal nodes" of the computation. If t is a non-empty traversal then the root $\circledast$ occurs exactly once in t thus the reduction of t is equal to $t \restriction r$ where r is the first occurrence in t (the only occurrence of the root). We write $\mathcal{T}rv(M)^{\restriction\circledast}$ for the set or reductions of traversals of M.

Example 5.4. The reduction of the traversal given in example 5.2 is:

$$t \restriction \lambda f z = \lambda f z \overset{f}{\cdot} \overset{\lambda}{\cdot} z \ .$$

Application nodes are used to connect the operator and the operand of an application in the computation tree but since they do not play any role in the computation of the term, we can remove them from the traversals. We write $t - @$ for the sequence of nodes-with-pointers obtained by removing from t all @-nodes and value-leaves of @-nodes, any link pointing to an @-node being replaced by a link pointing to the immediate predecessor of @ in t. We write $\mathcal{T}rv(M)^{-@}$ for the set $\{t - @ \mid t \in \mathcal{T}rv(M)\}$.

Example 5.5. Let t be the traversal given in example 5.2, we have:

$$t - @ = \lambda f z \cdot \lambda u v \cdot u \cdot \lambda y \ f \cdot \lambda \cdot y \cdot \lambda \cdot v \cdot \lambda \cdot z \ .$$

Remark 5.6. Clearly if M is β-normal then $\tau(M)$ does not contain any @-node therefore all nodes are hereditarily enabled by the root and we have $\mathcal{T}rv(M)^{-@} = \mathcal{T}rv(M) = \mathcal{T}rv(M)^{\restriction\circledast}$.

Lemma 5.7. *Suppose that M is a β-normal simply-typed term. Let t be a non-empty traversal of M and r denote the only occurrence of $\tau(M)$'s root in t. If t's last occurrence is not a leaf then*

$$\ulcorner t \urcorner \restriction r = \ulcorner ?(t) \restriction r \urcorner .$$

In the lambda calculus without interpreted constants this lemma follows immediately from the fact that $\mathcal{T}rv(M) = \mathcal{T}rv(M)^{\restriction\circledast}$. But this Lemma remains valid in the presence of interpreted constants provided that the traversal rules implementing the constants are *well-behaved*[10].

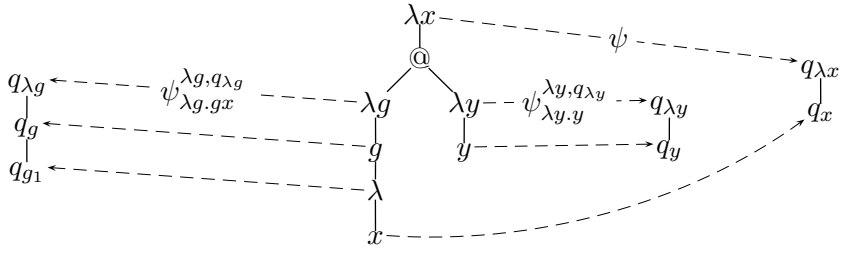
5.5. **Computation trees and arenas.** We consider the well-bracketed game model of the simply-typed lambda calculus. We choose to represent strategies using "prefix-closed set of plays".[11] We fix a term $\Gamma \vdash_{\mathsf{st}} M : T$ and write $[\![\Gamma \vdash_{\mathsf{st}} M : T]\!]$ for its strategy denotation. The answer moves of a question q are written v_q where v ranges in $\mathcal{D}$.

Proposition 5.8. *There exists a function φ_M, constructible from $\tau(M)$, that maps nodes from $V \setminus (V_@ \cup V_\Sigma)$ to moves of the interaction arena underlying the revealed strategy $\langle\!\langle \Gamma \vdash_{\mathsf{st}} M : T \rangle\!\rangle_{\mathsf{s}}$ and such that*

- *φ maps λ-nodes to O-questions, variable nodes to P-questions, value-leaves of λ-nodes to P-answers and value-leaves of variable nodes to O-answers.*
- *φ maps nodes of a given order to moves of the same order.*

If $t = t_0 t_1 \ldots$ is a justified sequence of nodes in $V_\lambda \cup V_{\mathsf{var}}$ then $\varphi(t)$ is defined to be the sequence of moves $\varphi(t_0)\ \varphi(t_1) \ldots$ equipped with the pointers of t.

Example 5.9. Take $\lambda x.(\lambda g.gx)(\lambda y.y)$ with $x, y : o$ and $g : (o, o)$. The diagram below represents the computation tree (middle), the arenas $[\![(o, o), o]\!]$ (left), $[\![o, o]\!]$ (right), $[\![o \to o]\!]$ (rightmost) and $\varphi = \psi \cup \psi^{\lambda g, q_{\lambda g}}_{\lambda g.gx} \cup \psi^{\lambda y, q_{\lambda y}}_{\lambda y.y}$ (dashed-lines).



[10]A traversal rule is ***well-behaved*** if it can be stated under the form "$t = t_1 \cdot n \cdot t_2 \in \mathcal{T}rv(M) \wedge ?(t) = ?(t_1) \cdot n \wedge n \in N_\Sigma \cup N_{\mathsf{var}} \wedge P(t) \wedge m \in S(t) \Longrightarrow t_1 \cdot n \cdot t_2 \cdot m \in \mathcal{T}rv(M)$" for some expression P expressing a condition on t and function S mapping traversals of the form of t to a subset of the children of n.

[11]In the literature, a strategy is commonly defined as a set of plays closed by taking a prefix of *even* length. However for the purpose of showing the correspondence with traversals, the "prefix-closed"-based definition is more adequate.

5.6. **The Correspondence Theorem.** In game semantics, strategy composition is performed using a CSP-like "composition + hiding". If some of the internal moves are not hidden then we obtain alternative denotations called ***revealed semantics*** in [16] and *interaction* semantics in [13]. We obtain different notions of revealed semantics depending on the choice of internal moves that we hide. For instance the ***fully revealed denotation*** of $\Gamma \vdash_{\mathsf{st}} M : T$, written $\langle\!\langle \Gamma \vdash_{\mathsf{st}} M : T \rangle\!\rangle$, is obtained by uncovering all the internal moves from $[\![\Gamma \vdash_{\mathsf{st}} M : T]\!]$ that are generated during composition.[12] The inverse operation consists in filtering out the internal moves.

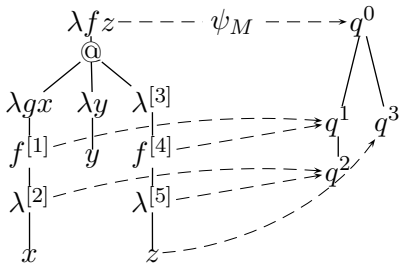
The ***syntactically-revealed denotation***, written $\langle\!\langle \Gamma \vdash_{\mathsf{st}} M : T \rangle\!\rangle_{\mathsf{s}}$, differs from the fully-revealed one in that only certain internal moves are preserved during composition: when computing the denotation of an application joint by an @-node in the computation tree, all the internal moves are preserved. When computing the denotation of $\langle\!\langle y_i N_1 \dots N_p \rangle\!\rangle$ for some variable y_i, however, we only preserve the internal moves of $N_1, \dots, N_p$ while omitting the internal moves produced by the copy-cat projection strategy denoting y_i.

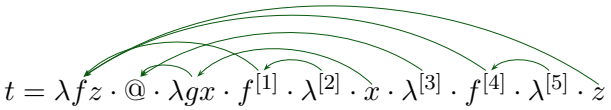
In the simply-typed lambda calculus, the set $\mathcal{T}rv(M)$ of traversals of the computation tree is isomorphic to the syntactically-revealed denotation, and the set of traversal reductions is isomorphic to the standard strategy denotation:

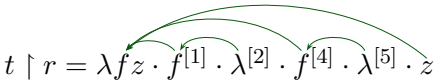
Theorem 5.10 (The Correspondence Theorem)**.** *φ_M gives us the following two isomorphisms:*

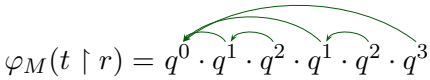
$$(i)\ \varphi_M \ : \mathcal{T}rv(M)^{-@} \xrightarrow{\cong} \langle\!\langle \Gamma \vdash_{\mathsf{st}} M : T \rangle\!\rangle_{\mathsf{s}}$$
$$(ii)\ \varphi_M \ : \mathcal{T}rv(M)^{\restriction \circledast} \xrightarrow{\cong} [\![\Gamma \vdash_{\mathsf{st}} M : T]\!] \ .$$

Example 5.11. Take $M = \lambda fz.(\lambda gx.fx)(\lambda y.y)(fz) : ((o,o),o,o)$. The figure below represents the computation tree (left tree), the arena $[\![((o,o),o,o)]\!]$ (right tree) and ψ_M (dashed line). (Only question moves are shown for clarity.) The justified sequence of nodes t defined hereunder is an example of traversal:









[12] An algorithm that uniquely recovers hidden moves from $[\![\Gamma \vdash_{\mathsf{st}} M : T]\!]$ is given in Part II of [18].