The Safe Lambda Calculus

Long version of the TLCA07 paper

version of May 4, 2007

William Blum C.-H. Luke Ong

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, ENGLAND
{william.blum,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. 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.

1 Introduction

Background

The *safety condition* was introduced by Knapik, Niwiński and Urzyczyn at FoSSaCS 2002 [14] 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 [6] 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 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 [7] 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:

[1] See de Miranda's thesis [8] for a proof.

Levels 0, 1 and 2 of the hierarchy are respectively the regular, context-free, and indexed languages (in the sense of Aho [4]), although little is known about higher orders. Remarkably, for generating word languages, order-n *safe* grammars are equivalent to order-n pushdown automata [7], which are in turn equivalent to order-n indexed grammars [16, 17].

- *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 [7] 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 [5] has shown that the MSO theories of graphs generated[3] by *safe* grammars of every finite order are decidable. However, in a recent preprint [12], 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 [3]. Here we propose a more general and cleaner version where terms are no longer required to be homogeneously typed (see Section 2 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 of Schwichtenberg [24] 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* [21] over a certain abstract syntax tree of the η-long form of M (called *computation*

[2] It has been recently been shown [21] that trees generated by *unsafe* deterministic grammars (of every finite order) also have decidable MSO theories.

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

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 η-long β-normal λ-term is *safe* if and only if its strategy denotation is (innocent and) *P-incrementally justified.*

2 The safe lambda calculus

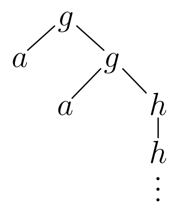
Higher-order safe grammars

We first present the safety restriction as it was originally defined [14]. 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. [14] for a formal definition.

Example 1. Let G be the following order-2 recursion scheme:



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

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 $g\,a\,(g\,a\,(h\,(h\,(h\,\cdots))))$.

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 [14]. We reproduce the following definition from [14].

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

Definition 1 (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 2. (i) Take $H : ((o,o),o)$, $f : (o,o,o)$; the following rewrite rules are unsafe (in each case we underline the unsafe subterm that occurs unsafely):

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

(ii) The order-2 grammar defined in Example 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 terms-in-context where Γ is the context and A is the type of M. For simplicity we write $(A_1, \cdots, A_n, B)$ to mean $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$, where B is not necessarily ground.

Definition 2. (i) The ***safe lambda calculus*** is a sub-system of the simply-typed lambda calculus defined by induction over the following rules:

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

$$(\mathsf{app})\ \frac{\Gamma \vdash_\Xi s : (A_1, \ldots, A_n, B)\ \ \Gamma \vdash_\Xi t_1 : A_1\ \ \ldots\ \ \Gamma \vdash_\Xi t_n : A_n}{\Gamma \vdash_\Xi s t_1 \ldots t_n : B}\ \mathsf{ord}(B) \sqsubseteq \mathsf{ord}(\Gamma)$$

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

where $\mathsf{ord}(\Gamma)$ denotes the set $\{\mathsf{ord}(y) : y \in \Gamma\}$ and "$c \sqsubseteq S$" means that c is a lower-bound of the set S. For convenience, we shall omit the subscript from $\vdash_\Xi$ whenever the generator-set Ξ is clear from the context.
(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***.

The safe lambda calculus deviates from the standard definition of the simply-typed lambda calculus in a number of ways. First the rules (app) and (abs) respectively can perform multiple applications and abstract several variables at once. (Of course this feature alone does not alter expressivity.) Crucially, the side-conditions in the application rule and abstraction rules require that variables in the typing context have order no smaller than that of the term being formed. We do not impose any constraint on types. In particular, type-homogeneity as used originally to define safe grammars [14] is not required here. Another difference is that we allow Ξ-constants to have arbitrary higher-order types.

Example 3 (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. *If $\Gamma \vdash M : A$ then every variable in Γ occurring free in M has order at least $ord(M)$.*

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. *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 $F z_1 \ldots z_m \to e$ in $\mathcal{R}$ is safe if and only if $z_1 : A_1, \cdots, z_m : A_m \vdash_{\Sigma \cup \mathcal{N}} e : o$ is a valid typing judgement of the* homogeneous *safe lambda calculus.*

In what sense is the safe lambda calculus safe? A basic idea in the lambda calculus is that when performing β-reduction, 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 2 (No variable capture). *There is no variable capture when performing capture-permitting substitution of N for x in M provided that $\Gamma, x : B \vdash M : A$ and $\Gamma \vdash N : B$ are valid judgments of the safe lambda calculus.*

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

If $x \notin \overline{y}$ then we have $M\{N/x\} = \lambda \overline{y}.R\{N/x\}$. By the induction hypothesis there is no variable capture in $R\{N/x\}$. Thus variable capture can only happen if the following two conditions are met: x occurs freely in R, and some variable y_i for $1 \leq i \leq p$ occurs freely in N. By Lemma 1, the latter condition implies $\mathsf{ord}(y_i) \geq \mathsf{ord}(N) = \mathsf{ord}(x)$. Since $x \notin \overline{y}$, the former condition implies that x occurs freely in the safe term $\lambda \overline{y}.R$ therefore Lemma 1 gives $\mathsf{ord}(x) \geq \mathsf{ord}(\lambda \overline{y}.R) \geq 1 + \mathsf{ord}(y_i) > \mathsf{ord}(y_i)$ which gives a contradiction. $\square$

Remark 1. 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 [14].

[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 $(\lambda y_1 \ldots y_n.M)\{N/x\} = \lambda y_1 \ldots y_n.M\{N/x\}$ when $x \notin \{y_1 \ldots y_n\}$.

Example 4. In order to contract the β-redex in the term

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

one should rename the bound variable x with 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. Indeed, it cannot be because $\mathsf{ord}(x) = 0 < 1 = \mathsf{ord}(fx)$.

Note that it is not the case that λ-terms that satisfy the No-variable-capture Lemma are 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.

Reductions and transformations preserving safety

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 3 (Substitution preserves safety). *If $\Gamma, x : B \vdash M : A$ and $\Gamma \vdash N : B$ then $\Gamma \vdash M[N/x] : A$.*

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 an alternative notion of reduction that performs simultaneous reduction of "consecutive" β-redexes. In order to define this reduction we first introduce the appropriate 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 3. Let $l \geq 1$ and $n \geq 1$. We use the abbreviations $\overline{x}$ and $\overline{x} : \overline{A}$ for $x_1 \ldots x_n$ and $x_1 : A_1, \ldots, x_n : A_n$ respectively.

A ***safe redex*** is a safe term of the form $(\lambda \overline{x}.M)N_1 \ldots N_l$ such that the variables $\overline{x}$ are abstracted altogether by one instance of the (abs) rule and the term $(\lambda \overline{x}.M)$ is applied to $N_1, \ldots, N_l$ by one instance of the (app) rule.

Thus M, the N_i's and the redex itself are all safe terms. For instance, in the case $n < l$, a safe redex has a derivation tree of the following form:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, \overline{x} : \overline{A} \vdash M : (A_{n+1}, \ldots, A_l, B)}}{\Gamma \vdash \lambda \overline{x}.M : (A_1, \ldots, A_l, B)}(\mathsf{abs}) \quad \dfrac{\cdots}{\Gamma \vdash N_1 : A_1} \quad \cdots \quad \dfrac{\cdots}{\Gamma \vdash N_l : A_l}}{\Gamma \vdash (\lambda \overline{x}.M)N_1 \ldots N_l : B}(\mathsf{app})$$

We are now in a position to define a notion of reduction for safe terms.

Definition 4. We use the abbreviations $\overline{x} = x_1 \ldots x_n$, $\overline{N} = N_1 \ldots N_l$. The relation β_s is defined on the set of safe redexes as:

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

where $M\left[R_1 \ldots R_k/z_1 \ldots z_k\right]$ denotes the simultaneous substitution of $R_1,\ldots,R_k$ for $z_1, \ldots, z_k$ in M. 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 β_s-reduction is a multi-step β-reduction *i.e.* $\rightarrow_\beta \subset \rightarrow_{\beta_s} \subset \twoheadrightarrow_\beta$.

Lemma 4 (β_s-reduction preserves safety). *If $\Gamma \vdash s : A$ and $s \rightarrow_{\beta_s} t$ then $\Gamma \vdash t : A$.*

Proof. It suffices to show that the relation β_s preserves safety. Suppose that $s\ \beta_s\ t$ where s is the safe-redex $(\lambda x_1 \ldots x_n.M)N_1 \ldots N_l$ with $x_1 : B_1, \ldots, x_n : B_n$ and M of type C. W.l.o.g we can assume that the last rule used to form the term s is (app) *i.e.* not the weakening rule (wk), thus we have $\Gamma = fv(s)$.

Suppose $n > l$ then $A = (B_{l+1}, \ldots, B_n, C)$. By Lemma 3 we can form the safe term $\Gamma, x_{l+1} : B_{l+1}, \ldots x_n : B_n \vdash M\left[\overline{N}/x_1 \ldots x_l\right] : C$. By Lemma 1, since s is safe, all the variables in Γ have order $\geq \mathsf{ord}(A)$. This ensures that the side-condition of the (abs) rule is verified if we abstract the variables $x_{l+1} \ldots x_n$, which gives us the judgement $\Gamma \vdash t : A$.

Suppose $n \leq l$. The substitution lemma gives $\Gamma \vdash M\left[N_1 \ldots N_n/\overline{x}\right] : C$ and using (app) we form $\Gamma \vdash t : A$. $\square$

In general, safety is not preserved by η-expansion; for instance we have $\vdash \lambda y^o z^o.y : (o, o, o)$ but $\not\vdash \lambda x^o.(\lambda y^o z^o.y)x : (o, o, o)$. However safety is preserved by η-reduction:

Lemma 5 (η-reduction preserves safety). *$\Gamma \vdash \lambda\varphi.s\varphi : A$ with φ not occurring free in s implies $\Gamma \vdash s : A$.*

Proof. Suppose $\Gamma \vdash \lambda\varphi.s\varphi : A$. If s is an abstraction then by construction of the safe term $\lambda\varphi.s\varphi$, s is necessarily safe. If $s = N_0 \ldots N_p$ with $p \geq 1$ then again, since $\lambda\varphi.N_0 \ldots N_p\varphi$ 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 $fv(s) \vdash N_0 \ldots N_p : A$. The weakening rules permits us to conclude $\Gamma \vdash s : A$. $\square$

The η-long normal form (or simply η-long form) of a term is obtained by hereditarily η-expanding every subterm occurring at an operand position. Formally the ***η-long form*** $\lceil t \rceil$ of a term $t : (A_1, \ldots, A_n, o)$ with $n \geq 0$ is defined by cases according to the syntactic shape of t:

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

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

Lemma 6 (η-long normalization preserves safety). *If $\Gamma \vdash s : A$ then $\Gamma \vdash \lceil s \rceil : A$.*

Proof. First we observe that for any variable or constant $x : A$ we have $x : A \vdash \lceil x \rceil : A$. 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 $\varphi_i : A_i \vdash \lceil \varphi_i \rceil : A_i$. Using (wk) we obtain $x : A, \overline{\varphi} : \overline{A} \vdash \lceil \varphi_i \rceil : A_i$. The application rule gives $x : A, \overline{\varphi} : \overline{A} \vdash x \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : o$ and the abstraction rule gives $x : A \vdash \lambda \overline{\varphi}.x \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil = \lceil x \rceil : A$.

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

- $s = x s_1 \ldots s_m$ 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 $\varphi_i : A_i \vdash \lceil \varphi_i \rceil : A_i$, the weakening rule then gives us $\Gamma, \overline{\varphi} : \overline{A} \vdash \lceil \varphi_i \rceil : A_i$. Since the judgement $\Gamma \vdash x s_1 \ldots s_m : A$ is formed using the (app) rule, each s_j must be safe for $1 \leq j \leq m$, thus by the induction hypothesis we have $\Gamma \vdash \lceil s_j \rceil : B_j$ and by weakening we get $\Gamma, \overline{\varphi} : \overline{A} \vdash \lceil s_j \rceil : B_j$. The (app) rule then gives $\Gamma, \overline{\varphi} : \overline{A} \vdash x \lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : o$. Finally the (abs) rule gives $\Gamma \vdash \lambda \overline{\varphi}.x \lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil = \lceil s \rceil : A$, the side-condition of (abs) being verified since $\mathsf{ord}(\lceil s \rceil) = \mathsf{ord}(s)$.
- $s = t s_0 \ldots s_m$ 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 $\Gamma \vdash \lambda \overline{\varphi}.\lceil t \rceil \lceil s_0 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil : A$.
- $s = \lambda \overline{\eta}.t$ where $\overline{\eta} : \overline{B}$ and $t : C$ is not an abstraction. The induction hypothesis gives $\Gamma, \overline{\eta} : \overline{B} \vdash \lceil t \rceil : C$ and using (abs) we get $\Gamma \vdash \lambda \overline{\eta}.\lceil t \rceil = \lceil s \rceil : A$. $\square$

Note that the converse does not hold in general, for instance $\lambda x^o.f^{(o,o,o)} x^o$ is unsafe although $\lceil \lambda x.fx \rceil = \lambda x^o y^o.fxy$ is safe.

Numeric functions representable in the safe lambda calculus

Natural numbers can be encoded into the simply-typed lambda calculus using the Church Numerals: each $n \in \mathbb{N}$ is encoded into the term $\overline{n} = \lambda sz.s^n z$ of type $I = ((o,o), o, o)$ where o is a ground type. In 1976 Schwichtenberg [24] showed the following:

Theorem 1 (Schwichtenberg 1976). *The numeric functions representable by simply-typed λ-terms of type $I \to \ldots \to I$ 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. *The functions representable by safe λ-expressions of type $I \to \ldots \to I$ are exactly the multivariate polynomials.*

Corollary 1. *The conditional operator $C : I \to I \to I \to I$ verifying $Ctyz \to_\beta y$ if $t \to_\beta \overline{0}$ and $Ctyz \to_\beta z$ if $t \to_\beta \overline{n+1}$ is not definable in the safe simply-typed lambda calculus.*

Proof. Natural numbers are encoded using Church Numerals: $\overline{n} = \lambda sz.s^n z$. Addition: For $n, m \in \mathbb{N}$, $\overline{n+m} = \lambda \alpha^{(o,o)} x^o.(\overline{n}\alpha)(\overline{m}\alpha x)$. Multiplication: $\overline{n.m} = \lambda \alpha^{(o,o)}.\overline{n}(\overline{m}\alpha)$. All 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 $n > 2$ is immediate (they correspond to polynomials with n variables). W.l.o.g we can assume that $U = \lambda xy\alpha z.u$ where u is a safe term of ground type in β-normal form with $fv(u) \subseteq \{x, y : I, z : o, \alpha : o \to o\}$.

Notation: Let T be a set of terms of type $\tau \to \tau$ and T' be a set of terms of type τ then $T \cdot T'$ denotes the set of terms $\{ss' : \tau \mid s \in T \wedge s' \in T'\}$. We also define $T^k \cdot T'$ recursively as follows: $T^0 \cdot T' = T'$ and for $k \geq 0$, $T^{k+1} \cdot T' = T \cdot (T^k \cdot T')$ (*i.e.* $T^k \cdot T'$ denotes $\{s_1(\ldots(s_k s')) \mid s_1, \ldots, s_k \in T \wedge s' \in T'\}$). We define $T^+ \cdot T' = \bigcup_{k>0} T^k \cdot T'$ and $T^* \cdot T' = (T^+ \cdot T') \cup T'$. For two sets of terms T and T', we write $T =_\beta T'$ to express that any term of T is β-convertible to some term t' of T' and reciprocally.

Let us write $\mathcal{N}^\tau$ for the set of β-normal terms of type τ where τ ranges in $\{o, o \to o, I\}$ and with free variables in $\{x, y : I, z : o, \alpha : o \to o\}$. We write $\mathcal{A}^\tau$ for the subset of $\mathcal{N}^\tau$ consisting of applications only (*i.e.* not abstractions). Let B be the set of terms of type (o, o) defined by $B = \{\alpha\} \cup \{\lambda a.b \mid b \in \{a, z\}, a \neq z\}$. It is easy to see that the following equations hold:

$$\begin{aligned}
\mathcal{A}^I &= \{x, y\} \\
\mathcal{N}^{(o,o)} &= B \cup \mathcal{A}^I \cdot \mathcal{N}^{(o,o)} = (\mathcal{A}^I)^* \cdot B \\
\mathcal{A}^{(o,o)} &= \{\alpha\} \cup (\mathcal{A}^I)^+ \cdot B \\
\mathcal{A}^o = \mathcal{N}^o &= \{z\} \cup \mathcal{A}^{(o,o)} \cdot \mathcal{N}^o = (\mathcal{A}^{(o,o)})^* \cdot \{z\}
\end{aligned}$$

Hence $\mathcal{A}^o = (\{\alpha\} \cup \{x, y\}^+ \cdot (\{\alpha\} \cup \{\lambda a.b \mid b \in \{a, z\}, a \neq z\}))^* \cdot \{z\}$. Since u is safe, it cannot contain terms of the form $\lambda a.z$ with $a \neq z$ occurring at an operand position, therefore since u belongs to $\mathcal{A}^o$ we have:

$$u \in \left(\{\alpha\} \cup \{x, y\}^+ \cdot \{\alpha, \underline{i}\}\right)^* \cdot \{z\} \tag{1}$$

where $\underline{i}$ is the identity term of type $o \to o$.

We observe that $\overline{k}\underline{i} =_\beta \underline{i}$ for all $k \in \mathbb{N}$ and for $l \geq 1$, for all $k_1, \ldots k_l \in \mathbb{N}$, $\overline{k_1} \ldots \overline{k_l}\alpha =_\beta \overline{k_1 \times \ldots \times k_l}\alpha$. Hence for all $m, n \in \mathbb{N}$ we have:

$$\begin{aligned}
\{\overline{m}, \overline{n}\}^+ \cdot \{\alpha, \underline{i}\} &=_\beta \{\underline{i}\} \cup \{\overline{m^i n^j}\alpha \mid i + j \geq 1\} \\
&= \{\overline{m^i n^j}\alpha \mid i, j \geq 0\} \qquad \text{(since } \underline{i} = \overline{0}\alpha)
\end{aligned} \tag{2}$$

therefore:

$$\begin{aligned}
u[\overline{m}, \overline{n}/x, y] &\in (\{\alpha\} \cup \{\overline{m}, \overline{n}\}^+ \cdot \{\alpha, \underline{i}\})^* \cdot \{z\} && \text{(by eq. 1)} \\
&=_\beta \left(\{\alpha\} \cup \{\overline{m^i n^j}\alpha \mid i, j \geq 0\}\right)^* \cdot \{z\} && \text{(by eq. 2)} \\
&=_\beta \left\{\overline{m^i n^j}\alpha \mid i, j \geq 0\right\}^* \cdot \{z\} && (\alpha z =_\beta \overline{1}\alpha z).
\end{aligned}$$

Furthermore, for all $m, n, r, i, j \in \mathbb{N}$ we have $\overline{m^i n^j}\alpha(\alpha^r z) =_\beta \alpha^{r+m^i n^j} z$, hence $u[\overline{mn}/x, y] =_\beta \alpha^{p(m,n)} z$ where $p(m,n) = \sum_{0 \leq k \leq d} m^{i_k} n^{j_k}$ for some $i_k, j_k \geq 0$, $k \in \{0, .., d\}$ and $d \geq 0$. Thus $U\overline{mn} =_\beta \overline{p(m,n)}$. □

For instance, the term $C = \lambda FGH\alpha x.H(\underline{\lambda y.G\alpha x})(F\alpha x)$ used by Schwichtenberg [24] to define the conditional operator is unsafe since the underlined subterm is of order 1, occurs at an operand position and contains an occurrence of x of order 0.

3 A game-semantic account of safety

Our aim here is to characterize safety, which is a syntactic property, by game semantics. Because of length restriction, we shall assume that the reader is familiar with the basics of game semantics. (For an introduction, we recommend [2]). 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 depends only on the *P-view* of a play. The main result (Theorem 3) of this section is that if a λ-term is safe, then its game semantics (is an innocent strategy that) is *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 3 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 certain 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 [13]) of plays in the strategy denotation.

The useful transference technique between plays and traversals was originally introduced by one of us [21] for studying the decidability of MSO theories of infinite structures generated by higher-order grammars (in which the Σ-constants 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 [21] 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 $l(t_1, \ldots, t_n)$ with

$n \geq 0$ to denote the tree with a root labelled l with n children subtrees $t_1, \ldots, t_n$. In the following, judgements of the form $\Gamma \vdash M : T$ refer to simply-typed terms not necessarily safe unless mentioned.

Definition 5. The ***computation tree*** $\tau(M)$ of a simply-typed term $\Gamma \vdash M : T$ with variable names in a countable set $\mathcal{V}$ is a tree with labels in $\{@\} \cup \mathcal{V} \cup \{\lambda x_1 \ldots x_n \mid x_1, \ldots, x_n \in \mathcal{V}\}$ defined from its η-long form as follows:

$$\begin{aligned}
\text{for } n \geq 0 \text{ and } s : o,\ \tau(\lambda x_1 \ldots x_n.s) &= \lambda x_1 \ldots x_n(t) \quad \text{where } \tau(s) = \lambda(t) \\
\text{for } m \geq 0 \text{ and } x \in \mathcal{V},\ \tau(x s_1 \ldots s_m : o) &= \lambda(x(\tau(s_1), \ldots, \tau(s_m))) \\
\text{for } m \geq 1,\ \tau((\lambda x.t) s_1 \ldots s_m : o) &= \lambda(@(\tau(\lambda x.t), \tau(s_1), \ldots, \tau(s_m))) \ .
\end{aligned}$$

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 6. 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 2. *(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 the η-long form of M is safe.*

The assumption that M is closed is necessary. For instance for $x, y : o$, the two identical computation trees $\tau(\lambda xy.x)$ and $\tau(\lambda y.x)$ are 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 is written $[\![\Gamma \vdash M : T]\!]$. 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 7. 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:

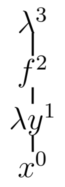
Lemma 7. *In P-incrementally justified strategies, pointers emanating from P-moves are superfluous.*

Example 5. 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.[6]

The Correspondence Theorem gives us the following equivalence:

Proposition 3. *For a β-normal term $\Gamma \vdash M : T$, $\tau(M)$ is incrementally-bound if and only if $[\![\Gamma \vdash M : T]\!]$ is P-incrementally justified.*

Example: Consider the β-normal term $\Gamma \vdash f(\lambda y.x) : o$ 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. Node x is not incrementally-bound therefore $\tau(f(\lambda y.x))$ is not incrementally-bound and by Proposition 3, $[\![\Gamma \vdash f(\lambda y.x) : o]\!]$ is not incrementally-justified (although $[\![\Gamma \vdash f : ((o,o),o)]\!]$ and $[\![\Gamma \vdash \lambda y.x : (o,o)]\!]$ are).



Propositions 2 and 3 allow us to show the following:

Theorem 3 (Safety and P-incremental justification).

(i) If $\Gamma \vdash M : T$ is safe then $[\![\Gamma \vdash M : T]\!]$ is P-incrementally justified.
(ii) If $\vdash M : T$ is a closed simply-typed term and $[\![\vdash M : T]\!]$ is P-incrementally justified then the η-long form of the β-normal form of M is safe.

Putting Theorem 3(i) and Lemma 7 together gives:

Proposition 4. *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 entirely superfluous at order 3 for safe terms. This is because for question moves in the first two levels of an arena, 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

[6] 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.

recoverable by P-incremental justification. This is not true anymore at order 4: Take the safe term $\psi : (((o^4, o^3), o^2), o^1) \vdash \psi(\lambda\varphi.\varphi a) : o^0$ 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.[7]

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. Using a PCF version of the Correspondence Theorem we can prove the following:

Theorem 4. *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) [23]. 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 [10] 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 [20, 19]. Restricting the problem to the safe fragment of IA_3 may lead to a lower complexity.

4 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

[7] 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. In the given example, the version of the play that is not O-incrementally justified is involved in the strategy composition $[\![\vdash M_2 : (((o, o), o), o)]\!] ; [\![\psi : (((o, o), o), o) \vdash \psi(\lambda\varphi.\varphi a) : o]\!]$ where M_2 denotes the unsafe Kierstead term.

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 [15]? Do incrementally-justified strategies compose? 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 [18] has shown that observational equivalence for IA_4 is undecidable; is observational equivalence for *safe* IA_4 decidable?

References

1. S. Abramsky. Semantics via game theory. In *Marktoberdorf International Summer School*, 2001. Lecture slides.
2. 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.
3. 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.
4. A. V. Aho. Indexed grammars – an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.
5. D. Caucal. On infinite terms having a decidable monadic theory. pages 165–176, 2002.
6. W. Damm. The IO- and OI-hierarchy. *TCS*, 20:95–207, 1982.
7. W. Damm and A. Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Information and Control*, 71(1-2):1–32, 1986.
8. J. G. de Miranda. *Structures generated by higher-order grammars and the safety constraint.* Dphil thesis, University of Oxford, 2006.
9. 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.
10. 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.
11. W. Greenland. *Game Semantics for Region Analysis.* PhD thesis, University of Oxford, 2004.
12. M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursive schemes. November 2006. 13 pages, preprint.
13. 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.
14. T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.
15. 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.
16. A. N. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet Math. Dokl.*, 15:1170–1174, 1974.
17. A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12:38–43, 1976.
18. 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.
19. 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.

20. C.-H. L. Ong. An approach to deciding observational equivalence of algol-like languages. *Ann. Pure Appl. Logic*, 130(1-3):125–171, 2004.
21. 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.
22. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes (technical report). Preprint, 42 pp, 2006.
23. 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.
24. H. Schwichtenberg. Definierbare funktionen im lambda-kalkul mit typen. *Archiv Logik Grundlagenforsch*, 17:113–114, 1976.

Appendix A – Computation tree, traversals and correspondence

In this section we adapt the notions of computation tree and traversals over the computation tree originally introduced in [21] to the context of the simply-typed lambda calculus without constants. Everything remains valid in the presence of *uninterpreted*[8] constants since we can just consider them as free variables. Moreover there is no restriction on the order of these constants, contrary to [21] where constants need to be of order at most one. We will also state the *Correspondence Theorem* (Theorem 5) used in Sec. 3.

In the following we fix a simply typed term $\Gamma \vdash M : T$ and we consider its computation tree $\tau(M)$ as defined in Def. 5.

4.1 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 (we write $m \vdash_i n$ to precise that n is the i^{th} variable bound by m), or $m = r$ and n is a free variable, or n is a λ-node and m is its parent node.

For any set of nodes S we write $S^{\upharpoonright r}$ for $\{n \in S \mid r \vdash^* n\}$ – the subset of S constituted of nodes hereditarily enabled by r. We call ***input-variables nodes*** the elements of $N_{var}^{\upharpoonright r}$.

A ***justified sequence of nodes*** is a sequence of nodes with pointers such that each variable or λ-node n different from the root has a pointer to a node m occurring before it the sequence such that $m \vdash n$. We represent the pointer in the sequence as follows $\overset{i}{\overparen{m \ldots n}}$. 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 nodes $n_1, \ldots, n_{p-1}$ in the sequence such that n_i points to n_{i+1} for all $i \in 0..p-1$.

[8] A constant f is *uninterpreted* if the small-step semantics of the language does not contain any rule of the form $f \ldots \rightarrow e$. f can be regarded as a data constructor.

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]

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

The O-view of s, written $\llcorner s \lrcorner$, is defined dually. We 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.

4.2 Computation tree with value-leaves

Let us first fix some notations. We write r for the root of $\tau(M)$ and N, $N_@$, N_λ and N_{var} for the set of nodes, @-labelled nodes, λ-nodes and variable nodes respectively. For $n \in N$, $\kappa(n)$ denotes the subterm of $\lceil M \rceil$ corresponding to the subtree rooted at n, in particular $\kappa(r) = \lceil M \rceil$. The ***type*** of a variable node labelled x is the type of x, the type of the root is $(A_1, \ldots, A_p, T)$ where $x_1 : A_1, \ldots, x_p : A_p$ are the free variables of M and the type of $n \in (N_\lambda \cup N_@) \setminus \{r\}$ is the type of $\kappa(n)$.

We now add another ingredient to the computation tree that was not originally used in [21]. 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 nodes and leaves of the computation tree. For \$ ranging in $\{@, \lambda, var\}$, we write $V_\$$ to denote the set $N_\$ \cup \{v_n \mid n \in N_\$, v \in \mathcal{D}\}$.

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: $\overset{v}{\overparen{n \ldots v_n}}$. 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 a node n is ***matched*** by v_n if there is an occurrence of v_n in the sequence that points to n, otherwise we say that n is ***unmatched***. The last unmatched 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 unmatched nodes.

4.3 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 8. The set $\mathcal{T}rv(M)$ of ***traversals*** over $\tau(M)$ is defined by induction over the following rules.

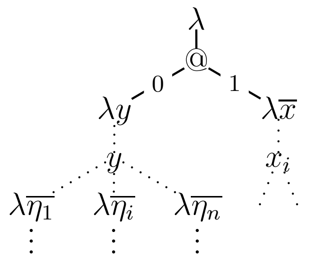
[9] The equalities in the definition determine pointers implicitly. For instance in the second clause, if in the left-hand side, n points to some node in s that is also present in $\ulcorner s \urcorner$ then in the right-hand side, n points to that occurrence of the node in $\ulcorner s \urcorner$.

(Empty) $\epsilon \in \mathcal{T}rv(M)$.
(Root) $r \in \mathcal{T}rv(M)$.
(Lam) If $t \cdot \lambda\overline{\xi}$ is a traversal then so is $t \cdot \lambda\overline{\xi} \cdot n$ where n is $\lambda\overline{\xi}$'s child and if n is a variable node then it points to the only[10] occurrence of its enabler that is still present in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$.
(App) If $t \cdot @$ is a traversal then so is $t \cdot @ \cdot \overset{0}{n}$.
(InputVar^{val}) If $t_1 \cdot x \cdot t_2$ is a traversal with $x \in N_{var}^{\restriction r}$ and $?(t_1 \cdot x \cdot t_2) = ?(t_1) \cdot x$ then so is $t_1 \cdot x \cdot t_2 \cdot \overset{v}{v_x}$ for all $v \in \mathcal{D}$.
(InputVar) If $t_1 \cdot x \cdot t_2$ is a traversal with $x \in N_{var}^{\restriction r}$ and $?(t_1 \cdot x \cdot t_2) = ?(t_1) \cdot x$ then so is $t_1 \cdot x \cdot t_2 \cdot n$ for any λ-node n whose parent occurs in $\llcorner t_1 \cdot x \lrcorner$, n pointing to some occurrence of its parent node in $\llcorner t_1 \cdot x \lrcorner$.
(Copycat$^@$) If $t \cdot @ \cdot \overset{0}{\lambda\overline{z}} \ldots \overset{v}{v_{\lambda\overline{z}}}$ is a traversal then so is $t \cdot @ \cdot \lambda\overline{z} \ldots \overset{v}{v_{\lambda\overline{z}}} \cdot \overset{v}{v_@}$.
(Copycat$^\lambda$) If $t \cdot \lambda\overline{\xi} \cdot x \ldots \overset{v}{v_x}$ is a traversal then so is $t \cdot \lambda\overline{\xi} \cdot x \ldots \overset{v}{v_x} \cdot \overset{v}{v_{\lambda\overline{\xi}}}$.
(Copycat^{var}) If $t \cdot y \cdot \lambda\overline{\xi} \ldots \overset{v}{v_{\lambda\overline{\xi}}}$ is a traversal for some variable y not in $N_{var}^{\restriction r}$ then so is

$t \cdot y \cdot \lambda\overline{\xi} \ldots \overset{v}{v_{\lambda\overline{\xi}}} \cdot \overset{v}{v_y}$.

(Var) If $t' \cdot n \cdot \lambda\overline{x} \ldots \overset{i}{x_i}$ is a traversal for some variable x_i not in $N_{var}^{\restriction r}$ then so is $t' \cdot n \cdot \lambda\overline{x} \ldots \overset{i}{x_i} \cdot \overset{i}{\lambda\overline{\eta_i}}$.

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 $\lambda \cdot @ \cdot \lambda y \ldots \overset{1}{y} \cdot \overset{1}{\lambda\overline{x}} \ldots \overset{i}{x_i} \cdot \overset{i}{\lambda\overline{\eta_i}} \ldots$ is an example of traversal of the computation tree shown on the right.


λ
@
0
1
λy
$\lambda\overline{x}$
y
x_i
$\lambda\overline{\eta_1}$
$\lambda\overline{\eta_i}$
$\lambda\overline{\eta_n}$

Proposition 5 (counterpart of proposition 6 from [22]). *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) $\ulcorner ?(t) \urcorner$ is the path in $\tau(M)$ from r to the last node in $?(t)$.

The ***reduction*** of a traversal t written $t \restriction r$, is the subsequence of t obtained by keeping only the nodes that are hereditarily justified by r. This has the effect of eliminating the "internal nodes" of the computation.

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 introduce the two notations $\mathcal{T}rv(M)^{-@} = \{t - @ \mid t \in \mathcal{T}rv(M)\}$ and $\mathcal{T}rv(M)^{\restriction r} = \{t \restriction r \mid t \in \mathcal{T}rv(M)\}$.

[10] This is justified *a posteriori* by the fact that P-views are paths in the computation tree.

Remark 2. Clearly if M is β-normal then τ does not contain any @-node therefore all nodes are hereditarily justified by r and we have $\mathcal{T}rv(M)^{-@} = \mathcal{T}rv(M) = \mathcal{T}rv(M)^{\upharpoonright r}$.

Lemma 8 (View of a traversal reduction). *If M is in β-normal form then for all $t \in \mathcal{T}rv(M)$ we have $\ulcorner ?(t) \upharpoonright r \urcorner = \ulcorner ?(t) \urcorner \upharpoonright r$.*

In the safe lambda calculus without interpreted constants this lemma follows immediately from the fact that $\mathcal{T}rv(M) = \mathcal{T}rv(M)^{\upharpoonright r}$. This remains valid in the presence of interpreted constants provided that the traversal rules implementing the constants are *well-behaved*.[11]

4.4 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". [12] We fix a term $\Gamma \vdash M : T$ and write $[\![\Gamma \vdash M : T]\!]$ for its strategy denotation. The answer moves of a question q are written v_q where v ranges in $\mathcal{D}$.

Definition 9 (Mapping from nodes to moves). Let q be a question move of $[\![T]\!]$ and $n \in N$ such that n and q are of type $(A_1, \ldots, A_p, o)$. Let $\{q^1, \ldots, q^p\} \cup \{v_q \mid v \in \mathcal{D}\}$ be the set of moves enabled by q where each q^i is of type A_i. The function $\psi_M^{n,q}$ from $V^{\upharpoonright n}$ to $[\![T]\!]$ is defined as:

$$\psi_M^{n,q} = \{n \mapsto q\} \cup \{v_n \mapsto v_q \mid v \in \mathcal{D}\}$$
$$\cup \begin{cases} \emptyset, & \text{if } p = 0 \ ; \\ \bigcup_{m \in N \mid n \vdash_i m} \psi_M^{m,q^i}, & \text{if } p \geq 1 \text{ and } n \in N_\lambda \ ; \\ \bigcup_{i=1..p} \psi_M^{n.i,q^i}, & \text{if } p \geq 1 \text{ and } n \in N_{var} \ . \end{cases}$$

Note that $\psi_M^{n,q}$ is only defined on nodes hereditarily enabled by n. For any $n \in N$ let A_n denote the type of $\kappa(n)$. We write $\psi_{\kappa(n)}$ for $\psi_{\kappa(n)}^{n,q}$ where q denotes the initial move of $[\![A_n]\!]$.[13]

For a closed term $\vdash M : T$, the total function φ_M from $V_\lambda \cup V_{var}$ to $[\![T]\!] \uplus \biguplus_{n \in N'_@} [\![A_n]\!]$ is defined as $\varphi_M = \psi_M \cup \bigcup_{n \in N'_@} \psi_{\kappa(n)}$ where $N'_@$ denotes the set of children nodes of @-nodes. For an open term $x_1 : X_1, \ldots, x_n : X_n \vdash M : T$, φ_M is defined as $\varphi_{\lambda x_1 \ldots x_n . M}$. When there is no ambiguity we omit the subscript in φ_M and ψ_M.

Remark 3. 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. Moreover φ maps nodes of a given order to moves of the same order.

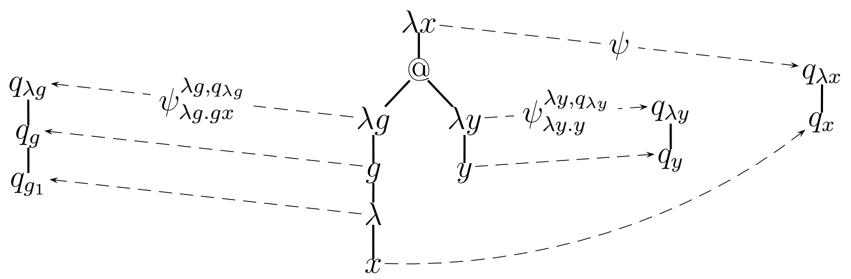
[11] A traversal rule is ***well-behaved*** if it can be stated under the form "$t = t_1 \cdot n \cdot t_2 \in \mathcal{T}rv \wedge ?(t) = ?(t_1) \cdot n \wedge n \in N_\Sigma \cup N_{var} \wedge P(t) \wedge m \in S(t) \Longrightarrow t_1 \cdot \overset{\curvearrowleft}{n \cdot t_2 \cdot m} \in \mathcal{T}rv$" 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.

[12] 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" version is more appropriate.

[13] Arenas involved in the game semantics of simply-typed lambda calculus are trees.

If $t = t_0 t_1 \ldots$ is a justified sequence of nodes in $V_\lambda \cup V_{var}$ 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 6. 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).



4.5 The Correspondence Theorem

In game semantics, strategy composition is achieved by performing a CSP-like "composition + hiding". If the internal moves are not hidden then we obtain an alternative semantics called ***revealed semantics*** in [11] and *interaction* semantics in [9]. The revealed semantics of a term $\Gamma \vdash M : T$, written $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$, is obtained by uncovering[14] the internal moves from $[\![\Gamma \vdash M : T]\!]$ that are generated by the composition with the evaluation map ev at each @-node of the computation tree. The inverse operation consists in filtering out the internal moves.

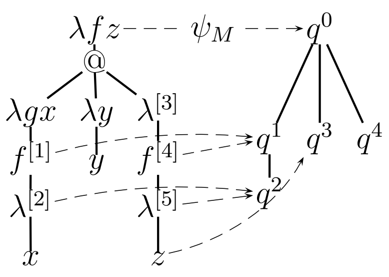
In the simply-typed lambda calculus, the set $\mathcal{T}rv(M)$ of traversals of the computation tree is isomorphic to the set of uncovered plays of the strategy denotation (this is the counterpart of the "Path-Traversal Correspondence" of [21]). Moreover the set of traversal reductions is isomorphic to the strategy denotation.

Theorem 5 (The Correspondence Theorem).

$$(i)\ \varphi_M : \mathcal{T}rv(M)^{-@} \xrightarrow{\cong} \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$$
$$(ii)\ \psi_M : \mathcal{T}rv(M)^{\upharpoonright r} \xrightarrow{\cong} [\![\Gamma \vdash M : T]\!]\ .$$

Example 7. Take $M = \lambda f z.(\lambda g x.f x)(\lambda y.y)(f z) : ((o, o), o, o)$. 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:

[14] An algorithm that uniquely recovers hidden moves is given in Part II of [13].



$$t = \lambda f z\ @\ \lambda g x\ f^{[1]}\ \lambda^{[2]}\ x\ \lambda^{[3]}\ f^{[4]}\ \lambda^{[5]}\ z$$

$$t \upharpoonright r = \lambda f z\ f^{[1]}\ \lambda^{[2]}\ f^{[4]}\ \lambda^{[5]}\ z$$

$$\psi_M(t \upharpoonright r) = q^0\ q^1\ q^2\ q^1\ q^2\ q^3 \in [\![M]\!]\ .$$

Appendix B – Proof of Theorem 3 and 4

4.6 Proof of Proposition 2

(i) Suppose that M is safe. By Lemma 6 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 $\mathsf{ord}(\lambda\overline{\xi})$. Reciprocally, 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. We assume that M is already 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 $M = \lambda\overline{\xi}.N_1 \ldots N_p$. 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 the (abs) which shows that N_i is safe. Finally we conclude $\vdash M = \lambda\overline{\xi}.N_1 \ldots N_p : T$ using the rules (app) and (abs). ☐

4.7 Proof of Theorem 3

(i) Let M be a safe simply-typed term. By Lemma 4, its β-normal form M' is also safe. By Proposition 2(i), $\tau(M')$ is incrementally-bound and by Proposition 3, $[\![M']\!]$ is an incrementally-justified. Finally the soundness of the game model gives $[\![M]\!] = [\![M']\!]$. (ii) is a consequence of Lemma 4, Proposition 3 and 2(ii) and soundness of the game model. ☐

4.8 Proof of Theorem 4

The computation tree of a PCF term is defined as the least upper-bound of the chain of computation trees of its *syntactic approximants* [2]. 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. It is trivial to show it for term approximants. The result is then lifted to any PCF term by observing that the function $\mathcal{T}rv^{\upharpoonright r}$ 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 is continuous.

Computation trees of Safe PCF terms are incrementally-bound. Moreover the traversal rules for PCF are *well-behaved* hence lemma 8 still holds and the game-semantic analysis of safety remains valid for PCF.