Transfer thesis

William Blum

August 28, 2006

Oxford University Computing Laboratory

CONTENTS

Part I

ACADEMIC ACTIVITIES

1. FIRST-YEAR WORK

1.1 Coursework

I have attended the following courses: *Automata Logic and Games* in Hilary term 2005, *Domain theory* in Michaelmas term 2005 and *Categories Proofs and Programs* in Hilary term 2006.

1.2 Teaching

I was the demonstrator for *Network and Operating Systems* practicals in Hilary term 2005, I tutored two groups of students for the *Introduction to Specification* classes (Hilary 2006) and I was the marker for one group.

1.3 Meetings and conferences

- I attended Bonn spring school on GAMES in March 2005;
- I attended BCTCS (British Colloquium in Theoretical Computer Science) in Nottingham in March 2005 where I gave a presentation based on my MSc dissertation "Termination analysis of a subset of CoreML";
- I attended PAT *Program transformation and Analysis* in Copenhagen, July 2005;
- Marktoberdorf Summer School;
- CSL (Computer Science Logic) August 2005: I helped to organise the conference;
- I visited the Isaac Newton Institute in Cambridge in February 2006.

I have also done a presentation during the Computer Laboratory open days.

1.4 Research

1.4.1 Game semantics

During the past months, I have studied a restriction of lambda-calculus called "safe lambda-calculus". *Safety* is a syntactic property originally defined in Knapik et al. [2002] for higher-order recursion schemes (grammars). In their paper they proved that the MSO theory of the term tree generated by a safe recursion scheme of level n is decidable. More recently, Ong proved in Ong [2006b] that the safety assumption is in fact not necessary for the decidability of MSO theories.

I am interested in the transposition of the safety property from grammars to lambda terms. A definition of the safe λ-Calculus was first given in a technical report by Aehlig, de Miranda and Ong in Aehlig et al. [2004]. One interesting property is that performing substitution on safe terms does not require a renaming of the variable.

I have investigated different possible definitions of a safe lambda calculus and have proposed a more general notion of safety that does not assume homogeneity of types while still preserving the "no variable renaming" property.

I also tried to relate the safety restriction and the *size-change termination* property defined in Lee et al. [2001], Jones and Bohr. [2004]. Jones conjectured that any simply-typed term is

size-change terminating, however Damien Sereni disproved this conjecture by exhibiting a class of counter-examples (Sereni [2005]). It turns out that the simply-typed terms of this class are all safe (but not necessary of homogeneous type) and not size-change terminating. This suggests that there is no real interesting relation between safety and size-change termination.

Recently, inspired by my reading on game semantics [Abramsky and McCusker, 1998b] and by the techniques developed by Luke Ong in [Ong, 2006b], I have proved a result on the game semantics of safe terms: the pointers in the game semantics of safe simply-typed terms can be recovered uniquely from the sequence of moves. This result is similar to the standard result in game semantics which says that pointers of strategies can be recovered uniquely for arena of order 2 at most.

1.4.2 Verification

In parallel, I worked on a separate project with Matthew Hagues and Luke Ong. We developed a SAT-based model checker for verifying Linear Temporal Logic (LTL) formulae on programs expressed as finite state machines. Our approach combines techniques presented in two papers: Hammer et al. [2005], McMillan [2003].

In McMillan [2003], McMillan describes an acceleration technique for the SAT-based Bounded Model Checking problem based on Craig interpolants. His algorithm significantly improves the performance of the standard SAT-based model checking method in the case of positive instances.

In [Hammer et al., 2005], Hammer *et al.* introduced a new kind of automata called *Linearly Weak Alternating Automata*, abbreviated LWAA. The set of languages recognized by these automata are exactly the set of languages definable in LTL. There is a straightforward translation from LTL formulae to LWAAs. The size of the resulting automaton is linear in the size of the LTL formula. Checking emptiness of LWAA then amounts to searching the configuration graph for a lasso verifying certain conditions.

Our approach can be summarized as follows: we translate the model checking problem into an emptiness checking of LWAA. The automata is empty if and only if the formula is true. The emptiness of the automaton is expressed in term of a reachability problem. As in the traditional SAT-based bounded-model checking approach (Biere et al. [1999]), we construct a boolean formula which is satisfiable if and only if the desired configuration is reachable in at most k steps (i.e. there is a counter-example of length k at most).

Furthermore, instead of using the traditional SAT-solver technique, which iterates k until the completeness threshold is reached, we use the acceleration method described in McMillan [2003]. The principle is the following: for every iteration of k, if the formula is not satisfiable then we perform some over-approximation of the set of initial configuration.

Suppose that the final configuration becomes reachable in k steps from the over-approximated initial configuration then we are still uncertain whether the formula has a valid counter-example because the counter-example obtained may be spuriously created by the over-approximation. We therefore increase k and move on to the next iteration. However, if after performing several over-approximations we reach a fixed point and the formula is still not satisfiable (not counter-example of length k at most) then we know that there cannot be any counter-example of any length. We have therefore reached the completeness threshold and we know that the formula is true.

There are two reasons why we think that our approach may lead to a gain of performance. Firstly, although determining emptiness of a LWAA is more costly than determining emptiness of a Büchi automaton, we save time during the construction of the automaton because the size of a LWAA is linear in the length of the formula as opposed to the standard translation which produces a Büchi automaton of size exponential in the length of the formula. Secondly, in the case where there is no counter-example, McMillan's acceleration method based on over-approximation permits quick detection of attaintment of the completeness threshold.

We have produced an experimental implementation in OCaml and C. The program parses a file in the NuSMV format (Cimatti et al. [2002]) containing the kripke structure of the model and the set of LTL properties to verify. Our tools can be interfaced with two SAT solvers: ZChaff [Princeton University] and MiniSat [N. Eén and N. Sörensson, 2003]. We also use BDD to perform

simplification on the propositional formula and to generate the CNF representation that the SAT solver takes as input.

Compared to the LWAASpin LTL model checker (Hammer et al. [2005]), our tool performs quite poorly. As soon as a model is taken into account, our procedure generates increasingly bigger propositional formulae that the SAT solver struggles to solve. However, for pure LTL emptiness checking, our tool performs quite well.

It seems disappointing that our approach does not give good results for model checking, however the reason seems to be that the SAT-solvers we are using produce bad interpolants. In the future, we would like to interface our model checking tool with other SAT solvers and interpolers.

Furthermore, there are optimizations that we have not finished to implement. These include the optimization of the encoding of the bounded model checking problem into a propositional formula. We propose to do some experimental tests do discover the encoding giving the best performance.

2. RESEARCH PLAN

My research plan for the coming year is as follows: first I will continue to work on the Safe λ-Calculus. My immediate goal is to extend the result I obtained about the unique recoverability of pointers in the game semantics of Safe simply-typed λ-calculus to the case of other languages like Safe Idealized Algol. I also wish to investigate applications in algorithmic game semantics. There are also further questions about Safe λ-Calculus that have to be addressed: what is the categorical interpretation of Safe λ-Calculus? What kind of proof theory do we obtain by the Curry-Howard isomorphism? Which complexity class is characterised by the Safe λ-Calculus?

In parallel to that line of research, I will continue to work with Matthew Hagues and Luke Ong on the LTL model checking problem.

Part II

SUMMARY OF WORK SO FAR

The first chapter of this part is devoted to the presentation of the basics and main results of game semantics. The categorical interpretation of game semantics is presented as well as the full abstraction result for PCF. We also give a brief summary of the main results in algorithmic game semantics. There is no personal contribution in this chapter.

In the second chapter we present the *Safe λ-Calculus*. Originally, *safety* has been introduced as a syntactical restriction on higher-order grammars in order to show a decidability result about MSO theory of infinite trees [Knapik et al., 2002]. In Aehlig et al. [2004], Aehlig, de Miranda and Ong proposed an adaptation of the safety restriction to the λ-calculus. This restriction gives rise to the Safe λ-Calculus. We first present this calculus and then give a more general definition which does not make any assumption on the types of the terms.

In the third chapter, following ideas described in Ong [2006b], we introduce the notions of computation tree of a simply-typed term and traversal over a computation tree. We prove a theorem showing a correspondence between traversals of the computation tree and the game semantics of a term. Based on that correspondence, we give a characterisation of the game semantics of safe terms by a property called "incremental justification". In incrementally-justified strategies, pointers are superfluous (i.e. they can be recovered uniquely from the underlying sequence of moves). This simplification of the game semantics suggests some potential applications in algorithmic game semantics. We finish the chapter by extending the result to Safe PCF and by giving the key elements for an extension to full Safe Idealized Algol.

Chapter 1

GAME SEMANTICS

The aim of this chapter is to introduce game semantics. It starts with a history of game semantics and a presentation of the full abstraction problem for PCF which has been solved using game semantics. It proceeds to introduce the basic notions of game semantics and give a categorical interpretation of games. Finally we show how games are used to define a syntax-independent model of programming languages like PCF and Idealized Algol (IA).

This chapter is largely based on the tutorial by Samson Abramsky on Game Semantics [Abramsky and McCusker, 1998b]. Many details and proofs will be omitted and we refer the reader to Hyland and Ong [2000], Abramsky et al. [1994] for a complete description of game semantics.

1.1 History

1.1.1 Game semantics

In the 1950s, Paul Lorenzen invented Game semantics as a new approach to study semantics of intuitionistic logic [Lorenzen, 1961]. In this setting, the notion of logical truth is modeled using game theoretic concepts (mainly the existence of winning strategy).

Four decades later, game semantics is used to prove the full completeness of Multiplicative Linear Logic (MLL) [Abramsky and Jagadeesan, 1992, Hyland and Ong, 1993]. Shortly after, a connection between games and linear logic was established. Game semantics has then emerged as a new paradigm for the study of formal models for programming languages. The idea is to model the execution of a program as a game played by two protagonists. The Opponent represents the environment and the Proponent represents the system. The meaning of the program is then modeled by a strategy for the Proponent.

Subsequently, these game-based model have been used to give a solution to the long-standing problem of "Full abstraction of PCF" [Abramsky et al., 1994, Hyland and Ong, 2000, Nickau, 1994].

Based on that major result, and in a more applied direction, games semantics has been used as a new tool for software verification Ghica and McCusker [2000]. This opened-up a new field called Algorithmic Game Semantics [Abramsky, 2001a].

1.1.2 Model of programming languages

Before the 1980s, there were many approaches to define models for programming languages. Among the successful ones, there were the axiomatic, operational and denotational semantics.

- Operational semantics gives a meaning to a program by describing the behaviour of a machine executing the program. It is defined formally by giving a state transition system.
- Axiomatic semantics defines the behaviour of the program through the use of axioms and is used to prove program correctness by static analysis of the code of the program.
- The denotational semantics approach consists in mapping a program to a mathematical structure having good properties such as compositionality. This mapping is achieved by structural induction on the syntax of the program.

In the 1990s, three different independent research groups: Samson Abramsky, Radhakrishnan Jagadeesan and Pasquale Malacaria [Abramsky et al., 1994], Martin Hyland and Luke Ong [Hyland and Ong, 2000] and Nickau [Nickau, 1994] introduced game semantics, a new kind of semantics, in order to solve a long standing problem in the semanticists community – finding a fully abstract model for PCF.

1.1.3 The problem of full abstraction for PCF

PCF is a simple programming language introduced in a classical paper by Plotkin "LCF considered as a programming language" (Plotkin [1977]). PCF is based on LCF, the Logic of Computable Functions devised by Dana Scott in Scott [1969]. It is a simply-typed lambda calculus extended with arithmetic operators, conditional and recursion.

The problem of the Full Abstraction for PCF goes back to the 1970s. In [Scott, 1993], Scott gave a model for PCF based on domain theory. This model gives a sound interpretation of observational equivalence – if two terms have the same domain theoretic interpretation then they are observationally equivalent. However the converse is not true – there exist two PCF terms which are observationally equivalent but have different domain theoretic denotations. As a result, we say that the model is not fully abstract.

The key reason why the domain theoretic model of PCF is not fully abstract is that the parallel-or operator defined by the following truth table

p-or	$\bot$	tt	ff
$\bot$	$\bot$	tt	$\bot$
tt	tt	tt	tt
ff	$\bot$	tt	ff

is not definable as a PCF term. It is possible to create two different PCF terms that always behave in the same way except when applied to a term computing p-or. Since p-or is not definable in PCF, these two terms will have the same denotation. This implies that the model is not fully abstract.

It is possible to patch PCF by adding the operator p-or, the resulting language "PCF+p-or" becomes fully-abstracted by Scott domain theoretic model [Plotkin, 1977]. However, the language we are now dealing with is strictly more powerful than PCF since it allows parallel execution of commands whereas PCF only permits sequential execution.

Another approach involves the elimination of the undefinable elements (like p-or) by strengthening the conditions on the function used in the model. This approach has been followed by Berry in Berry [1978, 1979] where he gives a model based on stable functions. Stable functions are a class of functions smaller than the class of strict and continuous function. Unfortunately this approach did not succeed.

The only successful approaches to obtain a fully abstract model for PCF were the ones taken by Ambramsky, Jagadeesan and Malacaria [Abramsky et al., 1994], Hyland and Ong [Hyland and Ong, 2000] and Nickau [Nickau, 1994]. They are all based on game semantics.

The game semantics approach has then been adapted to other varieties of programming paradigms including languages with stores (Idealized Algol), call-by-value [Honda and Yoshida, 1999, Abramsky and McCusker, 1998a] and call-by-name, general referencees [Abramsky et al., 1998], polymorphism [Abramsky and Jagadeesan, 2005], control features (continuation and exception), non determinism, concurrency. In all these cases, the game semantics model led to a syntax-independent fully abstract model of the corresponding language.

1.2 Games

We now introduce formally the notion of game that will be used in the following section to give a model of the programming languages PCF and Idealized Algol. The definitions are taken from Abramsky and McCusker [1998b], Hyland and Ong [2000], Abramsky et al. [1994].

The games we are interested in are two-players games. The players are named O for Opponent and P for Proponent. The game played by O and P is constrained by the *arena*. The arena defines

the possible moves of the game. By analogy with real board games, the arena represents the board together with the rules that tell players how they can make their moves on the board. The analogy with board game will not go beyond that. Instead, it is better to regard our games as dialogs between two players: player O interviews player P. P's goal is to answer the initial question asked by O. P can also ask questions to O if he needs more precision about O's initial question. Again, O can ask further question to P. This induces a flow of questions and answers between O and P that can continue possibly forever. In game semantics the attention is given to the study of this flow of questions and answers, the notion of winner of the game is not a concern.

1.2.1 Arenas

Our games have two kinds of moves – the questions and the answers. We also distinguish moves made by O and those made by P. An arena is represented by a directed acyclic graph (DAG) whose nodes correspond to question moves and leaves correspond to answer moves. It is formally defined as follows.

Definition 1.2.1 (Arena)**.** An arena is a structure $\langle M, \lambda, \vdash \rangle$ where:

- M is the set of possible moves;
- $(M, \vdash)$ is a directed acyclic graph;
- $\lambda : M \to \{O, P\} \times \{Q, A\}$ is a labelling function indicating whether a given move is a question or an answer and whether it can be played by O or P.
  $\lambda = [\lambda^{OP}, \lambda^{QA}]$ where $\lambda^{OP} : M \to \{O, P\}$ and $\lambda^{QA} : M \to \{Q, A\}$.
  - If $\lambda^{OP}(m) = O$, we call m an O-move otherwise m is a P-move. $\lambda^{QA}(m) = Q$ indicates that m is a question otherwise m is an answer.
  - a leaf l of the DAG $(M, \vdash)$ verifies $\lambda^{QA}(l) = A$ and a node n verifies $\lambda^{QA}(n) = Q$.
- The DAG $(M, \vdash)$ respects the following condition:
  - (e1) The roots are O-moves: for any root r of $(M, \vdash)$, $\lambda^{OP}(r) = O$;
  - (e2) enablers are questions: $m \vdash n \Rightarrow \lambda^{QA}(m) = Q$;
  - (e3) a player's move must be justified by a move played by the other player: $m \vdash n \Rightarrow \lambda^{OP}(m) \neq \lambda^{OP}(n)$.

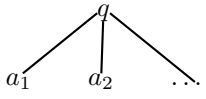
For the sake of convenience we write the set $\{O, P\} \times \{Q, A\}$ as $\{OQ, OA, PQ, PA\}$. $\overline{\lambda}$ denotes the labelling function obtained from λ after swapping players:

$$\begin{aligned} \overline{\lambda(m)} &= OQ \iff \lambda(m) = PQ \\ \text{and } \overline{\lambda(m)} &= OA \iff \lambda(m) = PA \end{aligned}$$

The roots of the DAG $(M, \vdash)$ are called the *initial moves*. Other moves must be enabled by some other question move. The edges of the DAG induces the enabling relation between moves.

The simplest possible arena, written **1**, is the arena with an empty set of moves.

Example 1.2.2 (The flat arena). Let A be any countable set, then the flat arena over A is defined to be the arena $\langle M, \lambda, \vdash \rangle$ such that M has one move q with $\lambda(q) = OQ$ and for each element in A, there is a corresponding move a_i in M with $\lambda(a_i) = PA$ for some $i \in \mathbb{N}$. The enabling relation $\vdash$ is defined to be $\{q \vdash a_i \mid i \in \mathbb{N}\}$. This arena is represented by the following tree whose vertices represent the moves and edges represent the enabling relation:



The flat arena over $\mathbb{N}$ and $\mathbb{B}$ is written **int** and **bool** respectively.

Once the arena has been defined, the bases of the game are set and the players have something to play with. We now need to describe the state of the game, for that purpose we introduce *justified sequences of moves*:

Definition 1.2.3 (Justified sequence of moves)**.** A justified sequence is a sequence of moves s together with an associated sequence of pointers. Any move m in the sequence that is not initial has a pointer that points to a previous move n that justifies it (i.e. $n \vdash m$).

Since initial moves are all O-moves, the first move of a justified sequence is necessarily an O-move

A justified sequence can be encoded as a sequence of pairs – a pair encodes an element of the sequence together with an index indicating the position where the element points to.

The pointers of a justified sequence are represented with arrows. The following is an example of justified sequence of moves:

$$\overset{\curvearrowleft}{q^4}\ \overset{\curvearrowleft}{q^3}\ \overset{\curvearrowleft}{q^2}\ \overset{\curvearrowleft}{q^3}\ \overset{\curvearrowleft}{q^2}\ q^1$$

Sequences of moves will be used to record the history of all the moves that have been played.

Notation: we write st or sometimes $s \cdot t$ to denote the sequences obtained by concatenating s and t. The empty sequence is written ϵ. Given a sequence $s = m_1 \cdot m_2 \dots m_n$ we write $s_{\leq m_i}$ for $m_1 \cdot m_2 \dots m_i$, the prefix sequence of s up to the move m_i. We write $s_{<m_i}$ for $m_1 \cdot m_2 \dots m_{i-1}$.

A justified sequence has two particular subsequences called the P-view and the O-view of the sequence. The idea is that a view describes the local context of the game. Here is the formal definition:

Definition 1.2.4 (View)**.** Given a justified sequence of moves s, we define the proponent view (P-view) written $\ulcorner s \urcorner$ by induction:

$$\begin{aligned}
\ulcorner \epsilon \urcorner &= \epsilon, && \\
\ulcorner s \cdot m \urcorner &= \ulcorner s \urcorner \cdot\ m && \text{if } m \text{ is a P-move,} \\
\ulcorner s \cdot m \urcorner &= m && \text{if } m \text{ is initial (O-move) ,} \\
\ulcorner s \cdot \overset{\curvearrowleft}{m} \cdot t \cdot \overset{\curvearrowleft}{n} \urcorner &= \ulcorner s \urcorner \cdot \overset{\curvearrowleft}{m} \cdot n && \text{if } n \text{ is a non initial O-move .}
\end{aligned}$$

The O-view $\llcorner s \lrcorner$ is defined similarly:

$$\begin{aligned}
\llcorner \epsilon \lrcorner &= \epsilon, && \\
\llcorner s \cdot m \lrcorner &= \llcorner s \lrcorner \cdot\ m && \text{if } m \text{ is a O-move,} \\
\llcorner s \cdot \overset{\curvearrowleft}{m} \cdot t \cdot \overset{\curvearrowleft}{n} \lrcorner &= \ulcorner s \urcorner \cdot \overset{\curvearrowleft}{m} \cdot n && \text{if } n \text{ is a P-move .}
\end{aligned}$$

1.2.2 Games

Not all justified sequences will be of interest for the games that we will use. We call *legal position* justified sequences that verify two additional conditions: alternation and visibility. Alternation says that players O and P play alternatively. Visibility expresses that each non-initial move is justified by a move situated in the local context at that point. The visibility condition gives some coherence to the justification pointers of the sequence.

Definition 1.2.5 (Legal position)**.** A legal position is a justified sequence of move s respecting the following constraints:

- *Alternation*: For any subsequence $m \cdot n$ of s, $\lambda^{OP}(m) \neq \lambda^{OP}(n)$.
- *Visibility*: For any subsequence tm of s where m is not initial, if m is a P-move then m points to a move in $\ulcorner s \urcorner$ and if m is a O-move then m points to a move in $\llcorner s \lrcorner$.

The set of legal position of an arena A is denoted by L_A.

We say that a move n is hereditarily justified by a move m if there is a sequence of move $m_1, \ldots, m_q$ such that:

$$m \vdash m_1 \vdash m_2 \vdash \ldots m_q \vdash n$$

If a move has no justification pointer, we says that it is an *initial move* (in that case it must be a root of the DAG of the arena).

Suppose that n is an occurrence of a move in the sequence s then $s \upharpoonright n$ denotes the subsequence of s containing all the moves hereditarily justified by n. Similarly, $s \upharpoonright I$ denotes the subsequence of s containing all the moves hereditarily justified by moves in I.

Definition 1.2.6 (Game)**.** A game is a structure $\langle M, \lambda, \vdash, P\rangle$ such that

- $\langle M, \lambda, \vdash\rangle$ is an arena;
- P is called the set of valid positions, it is:
  - a non-empty prefix closed subset of the set of legal position,
  - closed by initial hereditary filtering: if s is a valid position then for any set I of occurrences of initial moves in s, $s \upharpoonright I$ is also a valid position.

Example 1.2.7. Consider the flat arena **int**. The set of valid position $P = \{\epsilon, q\} \cup \{q \cdot a_i \mid i \in \mathbb{N}\}$ defines a game on the arena **int**.

1.2.3 Constructions on games

We now define game constructors that will be useful later on.

Consider the two functions $f : A \to C$ and $g : B \to C$, we write $[f, g]$ to denote the pairing of f and g defined on the direct sum $A + B$. Given a game A with a set of moves M_A, we use the filtering operator $s \upharpoonright A$ to denote the subsequence of s consisting of all moves in M_A. Although this notation conflicts with the hereditarily filtering operator, it should not cause any confusion.

Tensor product

Given two games A and B we define the tensor product constructor $A \otimes B$ as follows:

$$\begin{array}{rcl} M_{A\otimes B} & = & M_A + M_B \\ \lambda_{A\otimes B} & = & [\lambda_A, \lambda_B] \\ \vdash_{A\otimes B} & = & \vdash_A \ \cup \ \vdash_B \\ P_{A\otimes B} & = & \{s \in L_{A\otimes B} | s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}. \end{array}$$

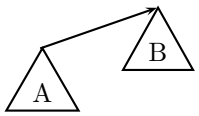
In particular, n is initial in $A \otimes B$ if and only if n is initial in A or B. And $m \vdash_{A\otimes B} n$ holds if and only if $m \vdash_A n$ or $m \vdash_B n$ holds.

Function space

The game $A \multimap B$ is defined as follows:

$$\begin{array}{rcl} M_{A\multimap B} & = & M_A + M_B \\ \lambda_{A\multimap B} & = & [\overline{\lambda_A}, \lambda_B] \\ \vdash_{A\multimap B} & = & \vdash_A \ \cup \ \vdash_B \ \cup \{(m, n) \mid m \text{ initial in } B \wedge n \text{ initial in } A\} \\ P_{A\otimes B} & = & \{s \in L_{A\otimes B} | s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}. \end{array}$$

Graphically if we draw a triangle to represent an arena A then the arena for $A \multimap B$ is represented as follows:



Cartesian product

The game $A\&B$ is defined as follows:

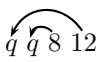
$$\begin{array}{rcl} M_{A\&B} & = & M_A + M_B \\ \lambda_{A\&B} & = & [\lambda_A, \lambda_B] \\ \vdash_{A\&B} & = & \vdash_A \ \cup \ \vdash_B \\ P_{A\&B} & = & \{s \in L_{A\otimes B} | s \upharpoonright A \in P_A \wedge s \ \upharpoonright B = \epsilon\} \\ & & \cup \{s \in L_{A\otimes B} | s \upharpoonright A \in P_B \wedge s \ \upharpoonright A = \epsilon\}. \end{array}$$

Note that a play of the game $A\&B$ is either a play of A or a play of B, whereas a play of the game $A \otimes B$ may be an interleaving of plays on A and plays on B.

1.2.4 Representation of plays

Plays of the game are usually represented in a table diagram. The columns of the table correspond to the different components of the arena and each row corresponds to one move in the play. The first row always represents an O-move, this is because O is the only player who can open a game (since roots of the arena are O-moves).

For example the play



on the game **int** $\multimap$ **int** is represented by the following diagram:

int	$\Rightarrow$	**int**	
		q	O
q			P
8			O
		12	P

When it is necessary, the justification pointers of the play are also shown on the diagram.

1.2.5 Strategy

During a game, the player who has to play may have several choices for his next move. A strategy is a guide telling the player which move to make when the game is in a given position. There is no notion of winning strategy since this is not relevant for the games that we are considering.

Definition

Formally, a strategy is a partial function mapping legal positions where P has to play to P-moves.

Definition 1.2.8 (Strategy)**.** A strategy for player P on a given game $\langle M, \lambda, \vdash, P\rangle$ is a non-empty set of even-length positions from P such that:

1. if $sab \in \sigma$ then $s \in \sigma$ (*no unreachable position*);
2. if $sab, sac \in \sigma$ then $b = c$ and b has the same justifier as c (*determinacy*).

The idea is that the presence of the even-length sequence sab in σ tells the player P that whenever the game is in position s and player O plays the move a then it must respond by playing the move b.

The first condition ensures that the strategy σ only considers positions that the strategy itself could have led to in a previous move. The second condition in the definition requires that this choice of move is deterministic (i.e. there is a function f from the set of odd length position to the set of moves M such that $f(sa) = b$).

For any game A, the smallest possible strategy is called the *empty strategy* and written $\bot$. It is formally defined by $\{\epsilon\}$, which corresponds to a strategy that never responds.

Remark 1.2.9. There is an alternative definition of a strategy. If we regard a strategy as an appropriated sub-tree of the game tree then it can be represented as the collection of all paths in this sub-tree, that is to say a certain prefix-closed set (as opposed to the *even-length prefix*-closed set of the above definition).

If σ denotes a strategy in the sense of definition 1.2.8 then the corresponding strategy in the alternative definition would be $\sigma \cup \mathsf{dom}(\sigma)$ where

$$\mathsf{dom}(\sigma) = \{sa \in P_A^{odd} | \exists b.sab \in \sigma\}.$$

Copy-cat strategy

For any arena A there is a strategy on the game $A \multimap A$ called the *copy-cat strategy*. We write A_1 and A_2 to denote the first and second copies of the arena A in the game $A \multimap A$. If A is the arena A_1 then $A^\bot$ denotes the arena A_2 and reciprocally.

Let A be one of the arena A_1 or A_2. The copy-cat strategy operates as follows: whenever P has to respond to an O-move played in A, it first replicates this move into the arena $A^\bot$. O then responds in $A^\bot$ and finally P replicates O's response back to A.

More formally, the copy-cat strategy is defined by:

$$\mathsf{id}_A = \{s \in P^{\mathsf{even}}_{A \multimap A} \mid \forall t \sqsubseteq^{\mathsf{even}} s \;.\; t \upharpoonright A_1 = t \upharpoonright A_2\}$$

where P_A^{even} denotes the set of valid positions of even length in the game A and $t \sqsubseteq^{\mathsf{even}} s$ denotes that t is an even length prefix of s.

The copy-cat strategy is also called *identity strategy* since it is the identity for strategy composition as we will see in the next paragraph.

Example 1.2.10. The copy-cat strategy on **int** is given by the following generic play:

int	$\Rightarrow$	**int**
		q
q		
n		
		n

Note that we introduced this type of diagram in the first place in order to represent plays but, as we can see here, whenever the represented play is general enough, the diagram can be used to represent strategies.

The copy-cat strategy on **int** $\Rightarrow$ **int** is given by the following diagram:

(**int**	$\Rightarrow$	**int**)	$\Rightarrow$	(**int**	$\Rightarrow$	**int**)
						q
		q				
q						
				q		
				m		
m						
		n				
						n

Composition

It is well-known that any model of the simply-typed lambda-calculus is a cartesian closed category [Crole, 1993]. Games are used to give a fully-abstract model of PCF – an extended simply-typed lambda calculus – therefore the game model should fit into a cartesian closed category. This category will have games as objects and strategies as morphisms. In a category, morphisms should be able to compose together, therefore there should be an appropriate notion of strategy composition.

In the following section we will show how strategies can be used to represent programs. A remarkable feature of the game model, called compositionality, is that obtaining the model of a composed program boils down to composing the strategies of the composing programs. Composition of strategies is therefore an essential feature of game semantics.

The way composition is defined for strategies is similar to "parallel composition plus hiding" in the trace semantics of CSP [Hoare, 1983]. Consider two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ that we wish to compose. For any sequence of moves u on three arenas A, B, C, we call projection of s on the game $A \multimap B$ and we write $u \upharpoonright A, B$ for the subsequence of s obtained by removing from u the moves in C and pointers to moves in C. The projection on $B \multimap C$ is defined similarly.

The definition of the projection on $A \multimap B$ differs slightly: $u \upharpoonright A, C$ is the subsequence of u consisting of the moves from A and C with some additional pointers. We add a pointer from $a \in A$ to $c \in C$ whenever a points to some move $b \in B$ itself pointing to c. All the pointers to moves in B are removed.

First we remark that for a given legal position s in the game $A \multimap C$, there is what is called an *uncovering* of s. The uncovering of s is the maximal justified sequence of moves u from the games A, B and C such that:

- The sequence s, considered as a pointer-less sequence, is a subsequence of u;
- the projection of u on the game $A \multimap B$ belongs to the strategy σ;
- the projection of u on the game $B \multimap C$ belongs to the strategy τ;
- and the projection of u on the game $A \multimap C$ is a subsequence of s (here the term "subsequence" refers to the sequence of nodes together with the auxiliary sequence of pointers).

This uncovering, written $uncover(s, \sigma, \tau)$, is defined uniquely for given strategies σ, τ and legal position s (this is proved in part II of Hyland and Ong [2000]).

We define $\sigma \| \tau$ to be the set of uncovering of legal positions in $A \multimap C$:

$$\sigma \| \tau = \{uncover(s, \sigma, \tau) \mid s \text{ is a legal position in } A \multimap C\}$$

The composition of σ and τ is defined to be the set of projections of uncovering of legal positions in $A \multimap C$:

Definition 1.2.11 (Strategy composition)**.** Let $\sigma : A \multimap B$ and $\tau : B \multimap C$ be two strategies. We define $\sigma ; \tau$ to be:

$$\sigma ; \tau = \{u \upharpoonright A, C \mid u \in \sigma \| \tau\}$$

It can be verified that composition is well-defined and associative [Hyland and Ong, 2000] and that the copy-cat strategy id_A is the identity for composition.

Constraint on strategies

Different classes of strategies will be considered depending on the features of the language that we want to model. Here is a list of common restrictions that we will consider:

- *Well-bracketing:* We call *pending question* the last question in a sequence that has not been answered. A strategy σ is well-bracketed if for every play $s \cdot m \in \sigma$ where m is an answer, m points to the pending question in s.

- *History-free strategies:* a strategy is history-free if the Proponent's move at any position of the game where he has to play is determined by the last move of the Opponent. In other words, the history prior to the last move is ignored by the Proponent when deciding how to respond.
- *History-sensitive strategies:* The Proponent follows a history-sensitive strategy if he needs to have access to the full history of the moves in order to decide which move to make.
- *Innocence:* a strategy is innocent if the determination of the Proponent's move depends only on a restricted view of the history of the play, mainly the P-view at that point. Such strategies can be specified by a partial function mapping P-views to P-moves called the *view function*. However not every partial function from P-views to P-moves gives rise to an innocent strategy (a sufficient condition is given in Hyland and Ong [2000]).

The formal definition of innocence follows:

Definition 1.2.12 (Innocence)**.** Given positions $sab, ta \in L_A$ where sab has even length and $\ulcorner sa \urcorner = \ulcorner ta \urcorner$, there is a unique extension of ta by the move b together with a justification pointer such that $\ulcorner sab \urcorner = \ulcorner tab \urcorner$. We write this extension $\mathsf{match}(sab, ta)$.

The strategy $\sigma : A$ is *innocent* if and only if:

$$\begin{pmatrix} \ulcorner sa \urcorner = \ulcorner ta \urcorner \\ sab \in \sigma \\ t \in \sigma \wedge ta \in P_A \end{pmatrix} \quad \Rightarrow \quad \mathsf{match}(sab, ta) \in \sigma$$

1.2.6 Categorical interpretation

In this section we recall some results about the categorical representation of games. These results with complete details and proofs can be found in McCusker [1996], Hyland and Ong [2000], Abramsky et al. [1994]. We refer the reader to Crole [1993] for more information about category theory.

We consider the category $\mathcal{G}$ whose objects are games and morphisms are strategies. A morphism from A to B is a strategy on the game $A \multimap B$.

Three other sub-categories of $\mathcal{G}$ are considered, each of them corresponds to some restriction on strategies: $\mathcal{G}_i$ is the sub-category of $\mathcal{G}$ whose morphisms are the innocent strategies, $\mathcal{G}_b$ has only the well-bracketed strategies and $\mathcal{G}_{ib}$ has the innocent and well-bracketed strategies.

Proposition 1.2.13. *$\mathcal{G}$, $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}_{ib}$ are categories.*

Proving this requires us to prove that composition of strategies is well-defined, associative, has a unit (the copy-cat strategy), preserves innocence and well-bracketedness. See Hyland and Ong [2000], Abramsky et al. [1994] for a proof.

Monoidal structure

We have already defined the tensor product on games in section 1.2.3. We now define the corresponding transformation on morphisms. Given two strategies $\sigma : A \multimap B$ and $\tau : C \multimap D$ the strategy $\sigma \otimes \tau : (A \otimes C) \multimap (B \otimes D)$ is defined by:

$$\sigma \otimes \tau = \{s \in L_{A \otimes C \multimap B \otimes D} \mid s \upharpoonright A, B \in \sigma \wedge s \upharpoonright C, D \in \tau\}$$

It can be shown that the tensor product is associative, commutative and has $I = \langle \emptyset, \emptyset, \emptyset, \{\epsilon\} \rangle$ as identity. Hence the game category $\mathcal{G}$ is a symmetric monoidal category. Moreover $\mathcal{G}_i$ and $\mathcal{G}_b$ are sub-symmetric monoidal categories of $\mathcal{G}$, and $\mathcal{G}_{ib}$ is a sub-symmetric monoidal category of $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}$.

Closed structure

Given the games A, B and C, we can transform strategies on $A \otimes B \multimap C$ to strategies on $A \multimap (B \multimap C)$ by retagging the moves to the appropriate arenas. This transformation defines an isomorphism written Λ_B and called currying. Therefore the hom-set $\mathcal{G}(A \otimes B, C)$ is isomorphic to the hom-set $\mathcal{G}(A, B \multimap C)$ which makes $\mathcal{G}$ an autonomous (i.e. symmetric monoidal closed) category.

We write $ev_{A,B} : (A \multimap B) \otimes A \to B$ to denote the *evaluation strategy* obtained by uncurrying the identity map on $A \to B$. $ev_{A,B}$ is in fact the copycat strategy for the game $(A \multimap B) \otimes A \to B$.

$\mathcal{G}_i$ and $\mathcal{G}_b$ are sub-autonomous categories of $\mathcal{G}$, and $\mathcal{G}_{ib}$ is a sub-autonomous category of $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}$.

Cartesian product

The cartesian product defined in section 1.2.3 is indeed a cartesian product in the category $\mathcal{G}$, $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}_{ib}$.

The projections $\pi_1 : A\&B \to A$ and $\pi_1 : A\&B \to B$ are given by the obvious copy-cat strategies. Given two category morphisms $\sigma : C \to A$ and $\tau : C \to B$, the pairing function $\langle \sigma, \tau \rangle : C \to A\&B$ is given by:

$$\begin{array}{rcl} \langle \sigma, \tau \rangle & = & \{s \in L_{C \multimap A\&B} \mid s \upharpoonright C, A \in \sigma \wedge s \upharpoonright B = \epsilon\} \\ & \cup & \{s \in L_{C \multimap A\&B} \mid s \upharpoonright C, A \in \sigma \wedge s \upharpoonright B = \epsilon\} \end{array}$$

Cartesian closed structure

Defining the cartesian product is not enough to turn $\mathcal{G}$ into a cartesian closed category : we also need to define a terminal object I and the exponential construct $A \Rightarrow B$ for any two games A and B. In fact, this cannot be done in the current category $\mathcal{G}$ and we have to move on to another category of games written $\mathcal{C}$ whose objects and morphisms are certain sub-classes of games and strategies.

Before defining the category $\mathcal{C}$ we need to introduce some definitions.

For any game A we define the exponential game denoted by $!A$. The game $!A$ corresponds to a repeated version of the game A. Plays of $!A$ are interleavings of plays of A. It is defined as follows:

$$\begin{array}{rcl} M_{!A} & = & M_A \\ \lambda_{!A} & = & \lambda_A \\ \vdash_{!A} & = & \vdash_A \\ P_{!A} & = & \{s \in L_{!A} | \text{ for each initial move } m,\ s \upharpoonright m \in P_A\} \end{array}$$

The following equalities hold:

$$\begin{array}{rcl} !(A\&B) & = & !A \otimes !B \\ I & = & !I \end{array}$$

Definition 1.2.14 (Well-opened games)**.** A game A is well-opened if for any position $s \in P_A$ the only initial move is the first one.

Well-opened games have single thread of dialog. They can be turned into games with multiple-thread of dialog using the promotion operator:

Definition 1.2.15 (Promotion)**.** Consider a well-opened game B. Given a strategy on $!A \multimap B$, we define its promotion $\sigma^\dagger : !A \multimap !B$ to be the strategy which plays several copies of σ. It is formally defined by:

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{ for all initial } m,\ s \upharpoonright m \in \sigma\}.$$

It can be shown that promotion is well-defined (it is indeed a strategy) and that it preserves innocence and well-bracketedness.

We now introduce the category of well-opened games.

Definition 1.2.16 (Category of well-opened games)**.** The category $\mathcal{C}$ of well-opened games is defined as follows:

1. The objects are the well-opened games,
2. a morphism $\sigma : A \to B$ is a strategy for the game $!A \multimap B$,
3. the identity map for A is the copy-cat strategy on $!A \multimap A$ (which is well-defined for well-opened games). It is called dereliction, denoted by der_A and defined formally by:

$$\mathsf{der}_A = \{s \in P^{\mathsf{even}}_{!A \multimap A} \mid \forall t \sqsubseteq^{\mathsf{even}} s \;.\; t \upharpoonright !A = t \upharpoonright A\},$$

4. composition of morphisms $\sigma : !A \multimap B$ and $\tau : !B \multimap C$ denoted by $\sigma \fatsemi \tau : !A \multimap C$ is defined as $\sigma^\dagger ; \tau$.

$\mathcal{C}$ is a well-defined category and the three sub-categories $\mathcal{C}_i$, $\mathcal{C}_b$, $\mathcal{C}_{ib}$ corresponding to sub-category with innocent strategies, well-bracketed strategies and innocent and well-bracketed strategies respectively.

The category $\mathcal{C}$ has a terminal object I, for any two games A and B a product $A\&B$ and an exponential $A \Rightarrow B$ defined to be $!A \multimap B$. The hom-sets $\mathcal{C}(A\&B, C)$ and $\mathcal{C}(A, !B \multimap C)$ are isomorphic. Indeed:

$$\begin{array}{rcll} \mathcal{C}(A\&B, C) & = & \mathcal{G}(!(A\&B), C) & \\ & = & \mathcal{G}(!A \otimes !B, C) & \\ & \cong & \mathcal{G}(!A, !B \multimap C) & (\mathcal{G} \text{ is a closed monoidal category}) \\ & = & \mathcal{C}(A, !B \multimap C) & \end{array}$$

Hence $\mathcal{C}$ is a cartesian closed category. Moreover $\mathcal{C}_i$ and $\mathcal{C}_b$ are sub-cartesian closed caterogies of $\mathcal{C}$ and $\mathcal{C}_{ib}$ is as sub-cartesian closed category of each of $\mathcal{C}$, $\mathcal{C}_i$ and $\mathcal{C}_b$.

Order enrichment

Strategies can be ordered using the inclusion ordering. Under this ordering, the set of strategies on a given game A is a pointed directed complete partial order : the least upper bound is given by the set-theoretic union and the least element is the empty strategy $\{\epsilon\}$.

Moreover all the operators on strategies that we have defined so far (composition, tensor product, ...) are continuous. Hence the categories $\mathcal{C}$ and $\mathcal{G}$ are cpo-enriched.

This property will prove to be useful when it comes to modeling programming languages with recursion such as PCF.

Intrinsic preorder

We now define a pre-ordering on strategies. The following definition is valid in any of the categories $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}_b$, $\mathcal{C}_{ib}$.

Let Σ be the game with a single question q and single answer a. There are only two strategies on Σ: $\bot = \{\epsilon\}$ and $\top = \{\epsilon, qa\}$ which are both innocent and well-bracketed. These strategies are used to test strategies. For any strategy $\sigma : \mathbf{1} \to A$ and for any test strategy $\alpha : A \to \Sigma$ we say that σ passes the test α if $\sigma \fatsemi \alpha = \top$.

The intrinsic preorder, written $\lesssim$, is defined as follows: for any strategy σ, τ on the game A, $\sigma \lesssim \tau$ if τ passes all the test passed by σ. Formally:

$$\sigma \lesssim \tau \quad \iff \quad \forall \alpha : A \to \Sigma . \sigma \fatsemi \tau = \top \Rightarrow \tau \fatsemi \alpha = \top$$

One can check that the relation $\lesssim$ is indeed a preorder on the set of strategies of the considered category. This preorder defines classes of equivalence. Two strategies are in the same equivalence class if no test can distinguish them. The quotiented category is written $\mathbf{C}/\lesssim$ where $\mathbf{C}$ ranges over $\{\mathcal{C}_i, \mathcal{C}_i, \mathcal{C}_b, \mathcal{C}_{ib}\}$.

Later on we will state the full abstraction of the game semantics model of PCF. This result will be proved in the quotiented category.

1.2.7 Pointers are superfluous for games on arenas of order 2

For any legal justified sequence of moves s, we write $?(s)$ for the subsequence of s obtained by keeping only the unanswered questions in s. It is easy to check that if s satisfies alternation then $?(s)$ also satisfies alternation.

Lemma 1.2.17. *If $s \cdot q$ is a legal position (i.e. justified sequence satisfying visibility and alternation) satisfying well-bracketing where q is a non-initial question then q points in $?(s)$.*

Proof. By induction on the length of $s \cdot q$. The base case $s = \epsilon$ is trivial. Let $s = s \cdot q$, where q is not initial.

Suppose q is a P-move. We prove that q cannot point to an O-question that has been answered. Suppose that an O-move q' occurs before q and is answered by the move a also occurring before q. Then we have $s = s_1 \cdot q'^O \cdot s_2 \cdot a^P \cdot s_3 \cdot q^P$ where a is justified by q'. a is not in the P-view $\ulcorner s_{<q} \urcorner$. Indeed this would imply that some O-move occurring in s_3 points to a, but this is impossible since answer moves are not enablers. Hence the move a must be situated underneath an O-to-P link. Let us note m the origin of this link, the P-view of s has the following form: $\ulcorner s \urcorner = \ulcorner s_1 \cdot q'^O \cdot s_2 \cdot a^P \dots m^O \urcorner \dots q^P$ where m is an O-move pointing before a.

If m is an answer move then it must point to the last unanswered move – that is to say the last move in $?(s_{<m})$. If m is a question move then it is not initial since there is a link going from m. Therefore by the induction hypothesis, m must point to a move in $?(s_{<m})$.

Since s is well bracketed, all the questions in the segment $q' \dots a$ are answered. Therefore since m points to an unanswered question occurring before a, m must point to a move occurring strictly before q'. Consequently q' does not occur in the P-view $\ulcorner s \urcorner$. By visibility, q must point in the P-view $\ulcorner s \urcorner$ therefore q does not point to q'.

A similar argument holds if q is an O-move. □

This means that in a well-bracketed legal position $s \cdot m$, if the move m is not initial then m must point to a question in $?(s)$ whether m is a question or an answer. Of course if m is an answer then it points precisely to the *last* question in $?(s)$. Moreover if m is a P-move then by visibility it should point to an unanswered question in $\ulcorner m \urcorner$ therefore it should also point in $?(\ulcorner m \urcorner)$. Similarly, if m is a non initial O-move then it points in $?(\llcorner m \lrcorner)$.

Lemma 1.2.18. *Let s be a legal well-bracketed position.*

1. *If $s = \epsilon$ or if the last move in s is not a P-answer then $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$;*
2. *If $s = \epsilon$ or if the last move in s is not an O-answer then $?(\llcorner s \lrcorner) = \llcorner ?(s) \lrcorner$.*

Proof. (i) By induction on the length of s. The base case is trivial. Step case: suppose that $s \cdot m$ is a legal well-bracketed position.

If m is an initial O-question then $?(\ulcorner s \cdot m \urcorner) = ?(m) = m = \ulcorner ?(s) \cdot m \urcorner = \ulcorner ?(s \cdot m) \urcorner$.

If m is a non initial O-question then $s \cdot m^O = s' \cdot q^P \cdot s'' \cdot m^O$ where m is justified by q. We have $?(\ulcorner s \urcorner) = ?(\ulcorner s' \urcorner \cdot q \cdot m) = ?(\ulcorner s' \urcorner) \cdot q \cdot m$. If s' is not empty then its last move must be an O-move (by alternation), therefore by the induction hypothesis $?(\ulcorner s' \urcorner) = ?(\ulcorner ?(s') \urcorner)$. By the previous lemma, the move m must point in $?(s)$ therefore we have $?(s \cdot m) = ?(s') \cdot q^P \cdot u \cdot m^O$ for some sequence u. And therefore $\ulcorner ?(s \cdot m) \urcorner = \ulcorner ?(s') \urcorner \cdot q^P \cdot m^O$.

If m is an O-answer then $s \cdot m = s' \cdot q^P \cdot s'' \cdot m^O$ where m is justified by q. Then $?(\ulcorner s \cdot m \urcorner) = ?(\ulcorner s' \urcorner q a) = ?(\ulcorner s' \urcorner)$. Moreover since s is well-bracketed, we have $?(s) = ?(s')$. Again the induction hypothesis permits to conclude.

If m is a P-question then $\ulcorner s \cdot m \urcorner = \ulcorner s \urcorner \cdot m$ and $?(\ulcorner s \cdot m \urcorner) = ?(\ulcorner s \urcorner) \cdot m$. Moreover $\ulcorner ?(s \cdot m) \urcorner = \ulcorner ?(s) \cdot m \urcorner = \ulcorner ?(s) \urcorner \cdot m$. By alternation if s is not empty it must end with an O-move and the induction hypothesis permits to conclude.

(ii) The argument is similar to (i). □

Note that for (i), and similarly for (ii), it is important that s does not end with a P-answer. For instance consider the legal position

$$s = q_0^O \; q_1^P \; q_2^O \; q_3^P \; q_4^O \; a^P$$

ending with a P-answer. We have $\ulcorner ?(s) \urcorner = \ulcorner q_0 \cdot q_1 \cdot q_2 \cdot q_3 \urcorner = q_0 \cdot q_1 \cdot q_2 \cdot q_3$ but $?(\ulcorner s \urcorner) = ?(q_0 \cdot q_1 \cdot q_4 \cdot a) = q_0 \cdot q_1 \cdot q_4$.

By the previous remark and lemma we obtain the following corollary:

Corollary 1.2.19. *Let $s \cdot m$ be a legal well-bracketed position.*

1. *If m is a P-move then it points in $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$;*
2. *if m is a non initial O-move then it points in $?(\llcorner s \lrcorner) = \llcorner ?(s) \lrcorner$.*

The height of a move m is the length of the longest sequence of moves $m \dots m_h$ in M such that $m \vdash m_2 \vdash \dots \vdash m_h$. The order of a move m written $\mathsf{ord}(M)$ is defined to be its height minus two. The order of an arena $\langle M, \lambda, \vdash \rangle$ is $\max_{m \in M} \mathsf{ord}(m)$.

We make the assumption that each question move in the arena enables at least one answer move. Consequently, moves of order 0 can only enable answer moves.

Lemma 1.2.20 (Pointers are superfluous up to order 2)**.** *Let A be an arena of order at most 2. Let s be a justified sequence of moves in the arena A satisfying alternation, visibility, well-openedness and well-bracketing then the pointers of the sequence s can be reconstructed uniquely from the underlying sequence of moves.*

Proof. Let A be an arena of order 2 at most. The case where A is a DAG with multiple roots can be reduced to the single root case as follows: since the justified sequence that we consider are well-opened, the first move in s denoted by m_0 is the only initial move in the sequence. m_0 must be the root of some sub-arena A' of A. Hence we just need to consider the arena A' instead of A and treat s as a play of A' instead of A. We now assume that A has a single root denoted by q_0.

Let s be a legal well-bracketed position in L_A. Note that since A is of order 2 at most, all the moves in s except q_0 are of order 1 at most.

We prove by induction on the length of s that $?(s)$ corresponds to one of the case 0, A, B, C or D shown on the table below, and that the pointers in s can be recovered uniquely.

Let L denote the language $L = \{\, pq \mid q_0 \vdash p \vdash q \wedge \mathsf{ord}(p) = 1 \wedge \mathsf{ord}(q) = 0 \}$.

Case	$\lambda_{OP}(m)$	$?(s) \in$	where...
0	O	$\{\epsilon\}$	
A	P	q_0	
B	O	$q_0 \cdot L^* \cdot p$	$q_0 \vdash p \quad \wedge \quad \mathsf{ord}(p) = 1$
C	P	$q_0 \cdot L^* \cdot pq$	$q_0 \vdash p \vdash q \quad \wedge \quad \mathsf{ord}(p) = 1 \quad \wedge \quad \mathsf{ord}(q) = 0$
D	O	$q_0 \cdot L^* \cdot q$	$q_0 \vdash q \quad \wedge \quad \mathsf{ord}(q) = 0$

Base cases: If s is the empty sequence ϵ then there is no pointer to recover and s corresponds to case 0. If s is a singleton then it must be the initial question q_0, therefore there is no pointer to recover. This corresponds to case A.

Step case: If $s = u \cdot m$ for some non empty legal well-bracketed position u and move $m \in M_A$ then by the induction hypothesis the pointers in u can all be recovered and u corresponds to one of the cases 0, A, B, C or D. We proceed by case analysis:

case 0 $?(u) = \epsilon$. By corollary 1.2.19, m points in $\ulcorner ?(u) \urcorner = \epsilon$. Hence this case is impossible.

case A $?(u) = q_0$ and the last move m is played by P. By corollary 1.2.19, m points to q_0.

If m is an answer to the initial question q_0 then s is a complete play and $?(s) = \epsilon$, which corresponds to case 0.

If m is a first order question then $?(s) = q_0 p$ and it is O's turn to play after s therefore s falls into category B.

If m is an order 0 question then s falls into category D.

case B $?(u) \in q_0 \cdot L^* \cdot p$ where $\mathsf{ord}(p) = 1$ and m is an O-move.

By corollary 1.2.19, m points in $\ulcorner ?(u) \urcorner = q_0 p$. Since m is an O-move it can only point to p.

If m is an answer to p then $?(s) = ?(u \cdot m) \in q_0 \cdot L^*$ which is covered by case A and C. If m is an order 0 question pointing to p then we have $?(s) = ?(u) \cdot m \in q_0 \cdot L^* \cdot pm$ and s falls into category C.

case C $?(u) \in q_0 \cdot L^* \cdot pq$ where $\mathsf{ord}(p) = 1$, $\mathsf{ord}(q) = 0$, q_0 justifies p, p justifies q and m is played by P.

Suppose that m is an answer, then the well-bracketing condition imposes q to be answered first. The move m therefore points to q and we have $?(s) = ?(u \cdot m) \in q_0 \cdot L^* \cdot p$. This corresponds to case B.

Suppose that m is a question. m is a P-move therefore is cannot be justified by p. It cannot be justified by q either because q is an order 0 question and therefore enables answer moves only. Similarly m is not justified by any move in L^*. Hence m must point to the initial question q_0. There are two sub-cases, either m is an order 0 move and then s falls into category D or m is an order 1 move and s falls into category B.

case D $?(u) \in q_0 \cdot L^* \cdot q$ where $\mathsf{ord}(q) = 0$ and m is played by O.

Again by corollary 1.2.19, m points in $\llcorner ?(u) \lrcorner = q_0 q$. Since m is a P-move it can only point to q. Since q is of order 0, it only enables answer moves therefore m is an answer to q. Hence $?(s) = ?(u \cdot m) \in q_0 \cdot L^*$ and s falls either into category A or C.

This completes the induction. □

1.2.8 ... but in general pointers are necessary

Up to order 2, the semantics of PCF terms is entirely defined by pointer-less strategies. In other words, the pointers of a justified sequence satisfying visibility and well-bracketing can be uniquely reconstructed from the underlying sequence of moves.

At level 3 however, pointers cannot be omitted in general. To illustrate this, consider the two Kierstead terms of type $((\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}^4$:

$$M_1 = \lambda f.f(\lambda x.f(\lambda y.y))$$

$$M_2 = \lambda f.f(\lambda x.f(\lambda y.x))$$

We assign a tag to each move so that q^i represents the question move of the component $\mathbb{N}^i$ in the type $((\mathbb{N}^1 \Rightarrow \mathbb{N}^2) \Rightarrow \mathbb{N}^3) \Rightarrow \mathbb{N}^4$. Now consider the play $s = q^4 q^3 q^2 q^3 q^2 q^1$ where pointers have been removed. It is possible to retrieve the pointers of the first five moves but there is an ambiguity for the last move: we do not know whether it points to the first or second occurrence of q^2 in the sequence s.

Note that the visibility condition does not eliminate the ambiguity because both occurrences of q^2 appear in the P-view at that point. Indeed, after recovering the pointers of s up to the second last move we get:

$$s = \overbrace{q^4\ q^3\ q^2\ q^3\ q^2}\ q^1,$$

and the P-view of s is s itself.

In fact these two different possibilities correspond to two different strategies. Suppose that the link goes to the first occurrence of q^2 then it means that the proponent is requesting the value of the variable x bound in the subterm $\lambda x.f(\lambda y....)$. If P needs to know the value of x, this is because P is in fact following the strategy of the subterm $\lambda y.x$. And the entire play is part of the strategy $[\![M_2]\!]$. If the link points to the second occurrence of q^2 then the play belongs to the strategy $[\![M_1]\!]$.

1.3 The fully abstract game model for PCF

In this section we introduce the functional languages PCF. We then describe the game model introduced in Abramsky et al. [1994] and finally we will state the full abstraction result.

1.3.1 The syntax of PCF

PCF is a simply-typed λ-calculus with the following additions: integer constants (of ground type), first-order arithmetic operators, if-then-else branching, and the recursion combinator $Y_A : (A \to A) \to A$ for any type A.

The types of PCF are given by the following grammar:

$$T ::= \mathtt{exp} \mid T \to T$$

and the structure of terms is given by:

$$\begin{aligned} M ::= {} & x \mid \lambda x : A.M \mid MM \mid \\ & \mid n \mid \mathtt{succ}\ M \mid \mathtt{pred}\ M \\ & \mid \mathtt{cond}\ MMM \mid \mathtt{Y}_A\ M \end{aligned}$$

where x ranges over a set of countably many variables and n ranges over the set of natural numbers.

Terms are generated according to the formation rules given in table 1.1 where the judgement is of the form $\Gamma \vdash M : A$.

$$(var)\frac{}{x_1 : A_1, x_2 : A_2, \ldots x_n : A_n \vdash x_i : A_i}\ i \in 1..n$$

$$(app)\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad (abs)\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

$$(const)\frac{}{\Gamma \vdash n : \mathtt{exp}} \qquad (succ)\frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}} \qquad (pred)\frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{pred}\ M : \mathtt{exp}}$$

$$(cond)\frac{\Gamma \vdash M : \mathtt{exp} \qquad \Gamma \vdash N_1 : \mathtt{exp} \qquad \Gamma \vdash N_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2} \qquad (rec)\frac{\Gamma \vdash M : A \to A}{\Gamma \vdash Y_A M : A}$$

Tab. 1.1: Formation rules for PCF terms

1.3.2 Operational semantics of PCF

We give the big-step operational semantics of PCF. The notation $M \Downarrow V$ means that the closed term M evaluates to the canonical form V. The canonical forms are given by the following grammar:

$$V ::= n \mid \lambda x.M$$

In other word, a canonical form is either a number or a function.

The full operational semantics is given in table 1.3.2. The evaluation rules are defined for closed terms only therefore the context Γ is not present in the rules. We write $M \Downarrow$ if the judgment $M \Downarrow V$ is valid for some value V.

$$\frac{}{V \Downarrow V} \quad \text{provided that } V \text{ is in canonical form.}$$

$$\frac{M \Downarrow \lambda x.M' \quad M'[x/N] \Downarrow V}{MN \Downarrow V}$$

$$\frac{M \Downarrow n}{\texttt{succ}\ M \Downarrow n+1} \qquad \frac{M \Downarrow n+1}{\texttt{pred}\ M \Downarrow n} \qquad \frac{M \Downarrow 0}{\texttt{pred}\ M \Downarrow 0}$$

$$\frac{M \Downarrow 0 \quad N_1 \Downarrow V}{\texttt{cond}\ MN_1N_2 \Downarrow V} \qquad \frac{M \Downarrow n+1 \quad N_2 \Downarrow V}{\texttt{cond}\ MN_1N_2 \Downarrow V}$$

$$\frac{M(\mathrm{Y}M) \Downarrow V}{\mathrm{Y}M \Downarrow V}$$

Tab. 1.2: Big-step operational semantics of PCF

1.3.3 Game model of PCF

As we have seen in section 1.2, games and strategies form a cartesian closed category, therefore games can model the simply-typed λ-calculus. We are now about to make this connection explicit by giving the strategy corresponding to a given λ-term. We will then extend the game model to PCF and IA.

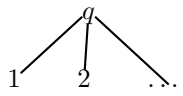
Simply-typed λ-calculus fragment

In the games that we are considering, the Opponent represents the environment and the Proponent represents the lambda term. Opponent opens the game by asking a question such as "What is the output of the function?". Then the proponent may ask further information such that "What is the input of the function?". O can provide P with an answer – the value of the input – or can pursue with another question. The dialog goes on until O gets the answer to his initial question.

O represents the environment, he is responsible for proving input values while P plays from the term's point of view: he is responsible for performing the computation and returning the output to O. P plays according to the strategy that is associated to the λ-term being modeled.

We recall that in the cartesian closed category $\mathcal{C}$, the objects are the games and the morphisms are the strategies. Given a simple type A, we will model it as a game $[\![A]\!]$. A context $\Gamma = x_1 : A_1, \ldots x_n : A_n$ will be mapped to the game $[\![\Gamma]\!] = [\![A_1]\!] \times \ldots \times [\![A_n]\!]$ and a term $\Gamma \vdash M : A$ will be modeled by a strategy on the game $[\![\Gamma]\!] \to [\![A]\!]$. Since $\mathcal{C}$ is cartesian closed, there is a terminal object $\mathbf{1}$ (the empty arena) that models the empty context ($[\![\Gamma]\!] = \mathbf{1}$).

Let ω denote the set of natural numbers. Consider the following flat arena over ω:



Then the base type `exp` is interpreted by the flat game $\mathbb{N}$ over the previous arena where the set of valid position is:

$$P_N = \{\epsilon, q\} \cup \{qn \mid n \in \omega\}$$

In this game, there is only one question: the initial O-question. P can then answer by playing a natural number $i \in \omega$. There are only two kinds of strategies on this arena:

- the empty strategy where P never answer the initial question. This corresponds to a non terminating computation;
- the strategies where P answers by playing a number n. This models a numerical constant of the language.

Given the interpretation of base types, we define the interpretation of $A \to B$ by induction:

$$[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!]$$

where the operator $\Rightarrow$ denotes the game construction $!A \multimap B$ i.e. the exponential object of the cartesian closed category $\mathcal{C}$.

Variables are interpreted by projection:

$$[\![x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i]\!] = \pi_i : [\![A_i]\!] \times \ldots \times [\![A_i]\!] \times \ldots \times [\![A_n]\!] \to [\![A_i]\!]$$

The abstraction $\Gamma \vdash \lambda x : A.M : A \to B$ is modeled by a strategy on the arena $[\![\Gamma]\!] \to ([\![A]\!] \Rightarrow [\![B]\!])$. This strategy is obtained by using the currying operator of the cartesian closed category:

$$[\![\Gamma \vdash \lambda x : A.M : A \to B]\!] = \Lambda([\![\Gamma, x : A \vdash M : B]\!])$$

The application $\Gamma \vdash MN$ is modeled using the evaluation map $ev_{A,B} : (A \Rightarrow B) \times A \to B$:

$$[\![\Gamma \vdash MN]\!] = \langle [\![\Gamma \vdash M, \Gamma \vdash N]\!] \rangle \fatsemi ev_{A,B}$$

PCF fragment

We now show how to model PCF constructs in the game semantics setting. In the following, each sub-arena of a game is tagged so that it is possible to distinguish identical arenas occurring in different components of the game. Moves are also tagged (in the exponent) so that it is possible to identify the arena component in which the move belongs. We will omit the pointers in the play when there is no ambiguity.

The successor arithmetic operator is modeled by the following strategy on the arena $\mathbb{N}^1 \Rightarrow \mathbb{N}^0$:

$$[\![\mathtt{succ}]\!] = \{q^0 \cdot q^1 \cdot n^1 \cdot (n+1)^0 \mid n \in \mathbb{N}\}$$

The predecessor arithmetic operator is denoted by the strategy

$$[\![\mathtt{pred}]\!] = \{q^0 \cdot q^1 \cdot n^1 \cdot (n-1)^0 \mid n > 0\} \cup \{q^0 \cdot q^1 \cdot 0^1 \cdot 0^0\}$$

Then given a term $\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}$ we define:

$$[\![\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}]\!] = [\![\Gamma \vdash M]\!] \fatsemi [\![\mathtt{succ}]\!]$$

$$[\![\Gamma \vdash \mathtt{pred}\ M : \mathtt{exp}]\!] = [\![\Gamma \vdash M]\!] \fatsemi [\![\mathtt{pred}]\!]$$

The conditional operator is denoted by the following strategy on the arena $\mathbb{N}^3 \times \mathbb{N}^2 \times \mathbb{N}^1 \Rightarrow \mathbb{N}^0$:

$$[\![\mathtt{cond}]\!] = \{q^0 \cdot q^3 \cdot 0 \cdot q^2 \cdot n^2 \cdot n^0 \mid n \in \mathbb{N}\} \cup \{q^0 \cdot q^3 \cdot m \cdot q^2 \cdot n^2 \cdot n^0 \mid m > 0, n \in \mathbb{N}\}$$

Given a term $\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2$ we define:

$$[\![\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2]\!] = \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_1]\!], [\![\Gamma \vdash N_2]\!] \rangle \fatsemi [\![\mathtt{cond}]\!]$$

The interpretation of the Y combinator is a bit more complicated.

Consider the term $\Gamma \vdash M : A \to A$, its semantics f is a strategy on $[\![\Gamma]\!] \times [\![A]\!] \to [\![A]\!]$. We define the chain g_n of strategies on the arena $[\![\Gamma]\!] \to [\![A]\!]$ as follows:

$$\begin{aligned} g_0 &= \bot \\ g_{n+1} &= F(g_n) = \langle id_{[\![\Gamma]\!]}, g_n \rangle \fatsemi f \end{aligned}$$

where $\bot$ denotes the empty strategy $\{\epsilon\}$.

It is easy to see that the g_n forms a chain. We define $[\![\mathtt{Y}M]\!]$ to be the least upper bound of the chain g_n i.e. the least fixed point of F. Its existence is guaranteed by the fact that the category of games is cpo-enriched.

Since all the strategies that we have given are innocent and well-bracketed, the game model of PCF can be interpreted in any of the four categories $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}_b$, $\mathcal{C}_{ib}$.

1.3.4 Full-abstraction of PCF

In this section we state the full abstraction result proved in Abramsky et al. [1994] and Hyland and Ong [2000].

Observational preorder

A context denoted $C[-]$ is a term containing a hole denoted by $-$. If $C[-]$ is a context then $C[M]$ denotes the term obtained after replacing the hole by the term M. $C[M]$ is well-formed provided that M has the appropriate type. Remark: this capture-permitting substitution must be distinguished from the capture-free substitution which is denoted by $M[N/x]$ for any two terms M and N.

We say that two programs are observationally equivalent if they can be safely interchanged in any program context.

Definition 1.3.1 (Observational preorder)**.** We define the relation on terms $\sqsubseteq$ as follows: let M and N be two closed terms of the same type then:

$$M \sqsubseteq N \quad \Longleftrightarrow \quad \begin{array}{l}\text{for all context } C[-] \text{ such that } C[M] \text{ and } C[N] \text{ are well-formed} \\ \text{closed term of type } \texttt{exp},\ C[M] \Downarrow \text{ implies } C[N] \Downarrow\end{array}$$

The observational equivalence relation, denoted by $\approx$, is defined to be the reflexive closure of $\sqsubseteq$.

Soundness and adequacy

A model of a programming language is said to be *sound* or *inequationally sound* if whenever the denotation of two programs are equal then the two programs are observationally equivalent, or more formally if for any closed terms M and N of the same type:

$$[\![M]\!] \subseteq [\![N]\!] \Rightarrow M \sqsubseteq N.$$

In a way, soundness is the minimum one can require for a model of programming language: it guarantees that we can reason about the program by manipulating the object of the denotational model.

It can be shown that the game model of PCF is sound for evaluation and computationally adequate. These two properties imply the soundness of the game model:

We said that the evaluation relation $\Downarrow$ is sound if the denotation is preserved by evaluation:

Lemma 1.3.2 (Soundness of evaluation)**.** *Let M be a PCF term then*

$$M \Downarrow V \quad \Rightarrow \quad [\![M]\!] = [\![V]\!].$$

Definition 1.3.3 (Computable terms)**.**

- A closed term $\vdash M$ of base type is computable if $[\![M]\!] \neq \bot$ implies $M \Downarrow$.
- A higher-order closed term $\vdash M : A \to B$ is computable if MN is computable for any computable closed term $\vdash N : A$.
- An open term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A \to B$ is computable if $\vdash M[N_1/x_1, \ldots N_n/x_n]$ is computable for all computable closed terms $N_1 : A_1, \ldots, N_n : A_n$.

A model is *computationally adequate* if all terms are computable.

Lemma 1.3.4 (Computational adequacy)**.** *The game model of PCF is computationally adequate.*

We refer the reader to Abramsky and McCusker [1998b] for the proofs.

Inequational soundness follows from the last two lemmas:

Proposition 1.3.5 (Inequational soundness)**.** *Let M and N be two closed terms then*

$$[\![M]\!] \subseteq [\![N]\!] \implies M \sqsubseteq N$$

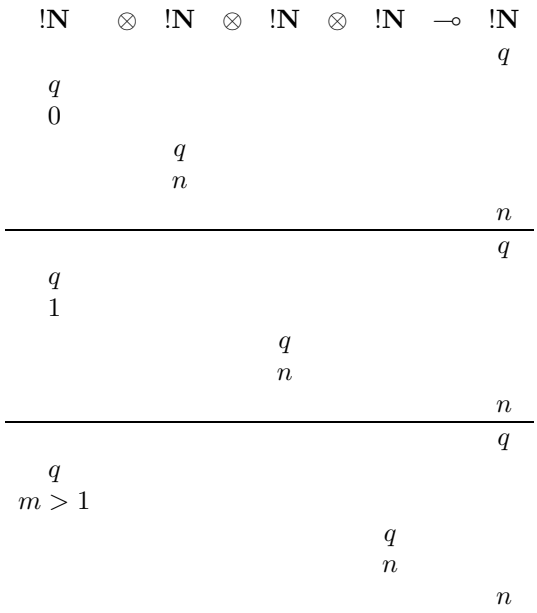
Proof. Suppose that $[\![M]\!] \subseteq [\![N]\!]$ and $C[M] \Downarrow$ for some context $C[-]$. Then by compositionality of game semantics we also have $C[[\![M]\!]] \subseteq C[[\![N]\!]]$. Lemma 1.3.2 gives $[\![C[M]]\!] \neq \bot$, therefore $[\![C[N]]\!] \neq \bot$. Lemma 1.3.4 then implies that $C[N] \Downarrow$. Hence $M \sqsubseteq N$. □

Definability

We will now consider only strategies that are innocent and well-bracketed which means that we work in the category $\mathcal{C}_{ib}$.

The compact morphisms of the category $\mathcal{C}_{ib}$ are those with finite view-function. The definability result says that every compact element of the model is the denotation of some term.

The economical syntax of PCF prevents us from stating this result directly: we need to consider an extension of PCF with some additional constants. Indeed, there are strategies that are not the denotation of any term in PCF, for instance the ternary conditional strategy : this strategy denotes the computation that tests the value of its first parameter, if it is equal to zero or one then it returns the value of the second or third parameter respectively, otherwise it returns the value of the fourth parameter. This strategy is illustrated by the following diagram:

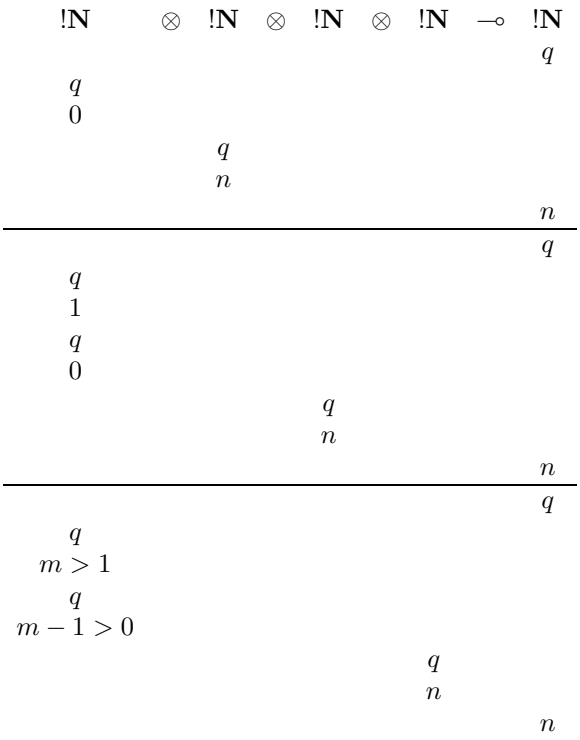


It is possible to simulate this computation in PCF using the conditional operator, for instance the following term is a potential candidate:

$$T_3 = \texttt{cond}\ M\ N_1(\texttt{cond}\ (\texttt{pred}\ M)\ N_2\ N_3)$$

Unfortunately the game semantics of T_3 is not given by the strategy that we have just defined,

it is instead the following one:



To make up for this deficiency we add a family of terms to PCF: the k-ary conditionals:

$$\mathtt{case}_k\ N\ N_1\ N_2\ \ldots\ N_k$$

with the desired operational semantics:

$$\frac{M \Downarrow i \quad N_{i+1} \Downarrow V}{\mathtt{case}_k\ N\ N_1\ N_2\ \ldots\ N_k\ \Downarrow V}\ i \in \{0, \ldots, k-1\}.$$

The denotation of this term is given by the first strategy illustrated above. The extended language is called PCF'.

We can now prove the definability result:

Proposition 1.3.6 (Definability)**.** *Let A be a PCF type and σ be a compact innocent and well-bracketed strategy on A. There exists a PCF' term M such that $[\![M]\!] = \sigma$.*

Note that definability is proved for PCF' and not for PCF. Nevertheless, PCF' is a conservative extension of PCF: if M and N are terms such that for any PCF-context $C[-]$, $C[M] \Downarrow\Rightarrow C[N] \Downarrow$ then the same is true for any PCF'-context. This is because $\mathtt{case}_k$ constructs can be "simulated" in PCF, for instance $\mathtt{case}_3$ can be replaced by the PCF term T_3 which shares the same operational semantics.

This observation will allow us to use definability in PCF' to prove the full-abstraction of PCF.

Full abstraction

Full abstraction of PCF cannot be stated directly in the category $\mathcal{C}_{ib}$. Instead we need to consider the quotiented category $\mathcal{C}_{ib}/\lesssim_{ib}$.

First we need to show that $\mathcal{C}_{ib}/\lesssim_{ib}$ is a model of PCF. $\mathcal{C}_{ib}/\lesssim_{ib}$ is a posset-enriched cartesian closed category. The game semantics of the basic types and constants of PCF can be transposed from $\mathcal{C}_{ib}$ to $\mathcal{C}_{ib}/\lesssim_{ib}$. Unfortunately it is not known whether $\mathcal{C}_{ib}/\lesssim_{ib}$ is enriched over the category

of CPOs. However it can be proved that it is a rational category [Abramsky et al., 1994] and this suffices to ensure that $\mathcal{C}_{ib}/\precsim_{ib}$ is indeed a model of PCF. The full abstraction of the game model then follows from proposition 1.3.5 and 1.3.6:

Theorem 1.3.7 (Full abstraction)**.** *Let M and N be two closed IA-terms.*

$$[\![M]\!] \precsim_{ib} [\![N]\!] \iff M \sqsubseteq N$$

where $\precsim_{ib}$ denotes the intrinsic preorder of the category $\mathcal{C}_{ib}$.

1.4 The fully abstract game model for Idealized Algol (IA)

We now extend the work of the previous section to the language IA, an imperative extension of PCF. We start by giving the syntax and operational semantics of the language, we then describe the game model which was introduced in Abramsky and McCusker [1999]. Finally we will state the full abstraction result for the game model.

1.4.1 The syntax of IA

IA is an extension of PCF introduced by J.C. Reynold in Reynolds. It adds imperative features such as local variables and sequential composition. On top of `exp`, PCF has two new types: `com` for commands and `var` for variables. There is a constant `skip` of type `com` which corresponds to the command that does nothing.

Commands can be composed using the sequential composition operator $\mathtt{seq}_A$: suppose that M and N are of type `com` and A respectively then they can be composed to form the term $S = \mathtt{seq}_A MN : \mathtt{com}$. S denotes program that executes M until it terminates and then behaves like $N : A$. If $A = \mathtt{exp}$ then the expression is allowed to have a side-effect and S returns the expression computed by N, if $A = \mathtt{com}$ then the command N is executed after M. We say that the language has *active expressions* to indicate the presence of the sequential operator $\mathtt{seq}_{\mathtt{exp}}$ in the language.

Local variables are declared using the `new` operator, variable content is altered using `assign` and retrieved using `deref`.

In addition IA has the constant `mkvar` that can be used to create a particular kind of variables. The `mkvar` operator works in an object oriented fashion: it takes two arguments, the first one is a function (called the acceptor) that affects a value to the variable and the second argument is an expression that returns a value from the variable. This mechanism is similar to the "set/get" object programming paradigm used by C++ programmers.

Variables created with `mkvar` are less constrained than the variables created with `new`. Indeed, variables created with `new` act like memory cells, they obey the following rule: the value read from the variable is always the last value that has been assigned to it. This rule does not apply to variables created with `mkvar`. For instance the variable:

$$\mathtt{mkvar}\ (\lambda v.\mathtt{skip})\ 0$$

will always return 0 even if another number has been assigned it.

One may think that this addition to the language is artificial, however the full abstraction result of the game model of IA relies upon this addition. At present, it is still an open problem to find a fully abstract model of IA deprived of `mkvar`.

The set of additional formations rules completing those of PCF are given in table 1.3. Judgements are of the form $\Gamma \vdash M : A$. If $\Gamma = \emptyset$ then we say that M is a closed term.

1.4.2 Operational semantics of IA

The operational semantics of IA is given in a slightly different form compared to PCF. Instead of giving the semantics for closed terms we consider terms whose free variables are all of type `var`.

$$\frac{\Gamma \vdash M : \mathtt{com} \quad \Gamma \vdash N : A}{\Gamma \vdash \mathtt{seq}_A\ M\ N\ : A} \quad A \in \{\mathtt{com}, \mathtt{exp}\}$$

$$\frac{\Gamma \vdash M : \mathtt{var} \quad \Gamma \vdash N : \mathtt{exp}}{\Gamma \vdash \mathtt{assign}\ M\ N\ : \mathtt{com}} \qquad \frac{\Gamma \vdash M : \mathtt{var}}{\Gamma \vdash \mathtt{deref}\ M\ : \mathtt{exp}}$$

$$\frac{\Gamma, x : \mathtt{var} \vdash M : A}{\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M} \quad A \in \{\mathtt{com}, \mathtt{exp}\}$$

$$\frac{\Gamma \vdash M_1 : \mathtt{exp} \to \mathtt{com} \quad \Gamma \vdash M_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{mkvar}\ M_1\ M_2\ : \mathtt{var}}$$

Tab. 1.3: Formation rules for IA terms

Terms are "closed" by mean of stores. A store is a function mapping free variables of type `var` to natural numbers. Suppose Γ is a context containing only variables of type `var`, then we say that Γ is a `var`-context. A store with domain Γ is called a Γ-store. The notation $s \mid x \mapsto n$ refers to the store that maps x to n and acts according to the store s otherwise.

The canonical forms for IA are given by the grammar:

$$V ::= \mathtt{skip} \mid n \mid \lambda x.M \mid x \mid \mathtt{mkvar}\ M\ N$$

where $n \in \mathbb{N}$ and $x : \mathtt{var}$.

In IA, a program is a term together with a Γ-store such that $\Gamma \vdash M : A$. The evaluation semantics is expressed by the judgment form:

$$s, M \Downarrow s', V$$

where s and s' are Γ-stores, V is a canonical form and $\Gamma \vdash V : A$.

The operational semantics for IA is given by the rule of PCF (table 1.3.2) together with the rules of table 1.4.2 where the following abbreviation is used:

$$\frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2}{M \Downarrow V} \quad \text{for} \quad \frac{s, M_1 \Downarrow s', V_1 \quad s', M_2 \Downarrow s'', V_2}{s, M \Downarrow s'', V}$$

$$\textbf{Sequencing}\ \frac{M \Downarrow \mathtt{skip} \quad N \Downarrow V}{\mathtt{seq}\ M\ N \Downarrow V}$$

$$\textbf{Variables}\ \frac{s, N \Downarrow s', n \quad s', M \Downarrow s'', x}{s, \mathtt{assign}\ M\ N \Downarrow (s'' \mid x \mapsto n), \mathtt{skip}} \qquad \frac{s, M \Downarrow s', x}{s, \mathtt{deref}\ M \Downarrow s', s'(x)}$$

$$\mathtt{mkvar}\ \frac{N \Downarrow n \quad M \Downarrow \mathtt{mkvar}\ M_1\ M_2 \quad M_1\ n \Downarrow \mathtt{skip}}{\mathtt{assign}\ M\ N \Downarrow \mathtt{skip}} \qquad \frac{N \Downarrow \mathtt{mkvar}\ M_1\ M_2 \quad M_2\ \Downarrow n}{\mathtt{deref}\ M \Downarrow n}$$

$$\textbf{Block}\ \frac{(s \mid x \mapsto 0), M \Downarrow (s' \mid x \mapsto n), V}{s, \mathtt{new}\ x\ \mathtt{in}\ M \Downarrow s', V}$$

Tab. 1.4: Big-step operational semantics of IA

1.4.3 Game model of IA

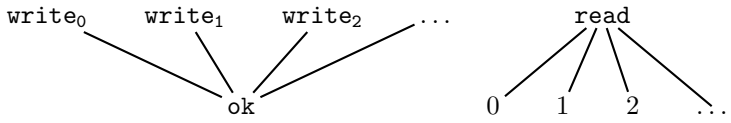
All the strategies used to model PCF are well-bracketed and innocent. On the other hand, to obtain a model of IA we need to introduce strategies that are not innocent. This is necessary to model memory cell variable created with the `new` operator. The intuition is that a cell needs to

remember the last value which was written in it in order to be able to return it when it is read, and this can only be done by looking at the whole history of moves, not only those present in the P-view. Hence we now consider the categories $\mathcal{C}$ and $\mathcal{C}_b$.

Base types

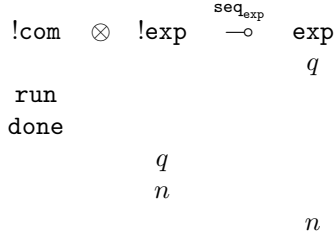
The type `com` is modeled by the flat game with a single initial question `run` and a single answer `done`. The idea is that O can request the execution of a command by playing `run`, P then executes the command and if it terminates, acknowledges it by playing `done`.

The variable type `var` is modeled by the game $\mathtt{com}^{\mathbf{N}} \times \mathtt{exp}$ illustrated below:

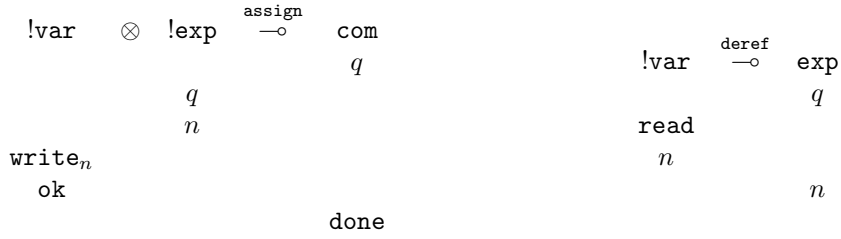


Constants

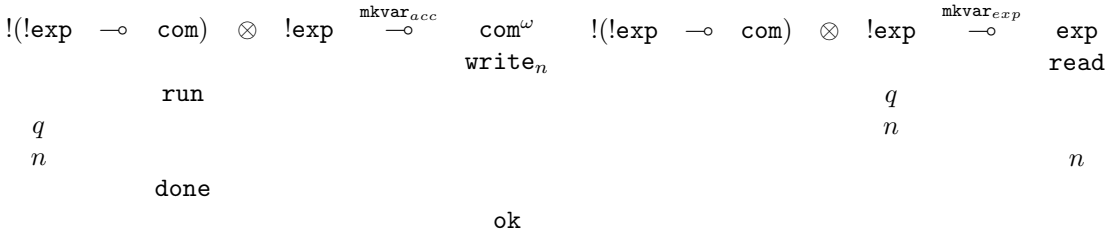
`skip` is interpreted by the strategy $\{\epsilon, \mathtt{run}\cdot\mathtt{done}\}$. The sequential composition $\mathtt{seq}_{\mathtt{exp}}$ is interpreted by the following strategy:



Assignment `assign` and dereferencing `deref` are denoted by the following strategies (left and right respectively):



`mkvar` is modeled by the paired strategy $\langle \mathtt{mkvar}_{acc}, \mathtt{mkvar}_{exp} \rangle$ where $\mathtt{mkvar}_{acc}$ and $\mathtt{mkvar}_{exp}$ are the following strategies:



The strategies used until now are all innocent. In order to model the `new` operator, we need to introduce non-innocent strategies, sometimes called *knowing strategies*. We define the knowing well-bracketed strategy $cell : I \multimap !\mathtt{var}$ that models a storage cell: it responds to `write` with `ok` and responds to `read` with the last value written or 0 if no value has yet been written.

Consider the term $\Gamma, x : \mathtt{var} \vdash M : A$ modeled by $[\![M]\!]$ then the term $\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M : A$ will be modeled by the strategy $(id_{[\![\Gamma]\!]} \otimes cell) \fatsemi [\![M]\!]$ on the game $!\Gamma \multimap \mathtt{com}$.

1.4.4 Full abstraction of IA

We now state the full abstraction result. All the details are omitted, the reader is refered to Abramsky and McCusker [1998b, 1997] for the proofs.

Inequational soundness

The inequational soundness result can be also proved for IA. Proving soundness of the evaluation requires a bit more work than in the PCF case because the store needs to be made explicit. Also, we need to define an appropriate notion of *computable term* that takes into account the presence of stores in the evaluation semantics. It is possible to prove that the model is computational adequate. The inequational soundness then follows from evaluation soundness and computational adequacy.

Proposition 1.4.1 (Inequational soundness)**.** *Let M and N be two IA closed terms then*

$$[\![M]\!] \subseteq [\![N]\!] \implies M \sqsubseteq N$$

Definability

The proof of definability is based on a factoring argument: strategies in $\mathcal{G}_b$ can all be obtained by composing the non-innocent strategy *cell* with an innocent strategy. The strategy *cell* can therefore be viewed as a generic non-innocent strategy. Using this factorization argument, it is possible to prove the definability result:

Proposition 1.4.2 (Definability)**.** *Let σ be a compact well-bracketed strategy on a game A denoting a IA type. There is an IA-term M such that $[\![M]\!] = \sigma$.*

Full abstraction

Full abstraction for the model $\mathcal{C}_b$ is a consequence of proposition 1.4.1 and 1.4.2:

Theorem 1.4.3 (Full abstraction)**.** *Let M and N be two closed IA-terms.*

$$[\![M]\!] \precsim_b [\![N]\!] \iff M \sqsubseteq N$$

where $\precsim_b$ denotes the intrinsic preorder of the category $\mathcal{C}_b$.

1.5 Algorithmic game semantics

After the resolution of the "Full Abstraction of PCF" problem, game semantics has become a very successful paradigm in fundamental computer science. It has permitted to give full abstract semantics for a variety of programming languages. More recently, game semantics has emerged as a new approach to program verification and program analysis. In particular in the paper Ghica and McCusker [2000], the authors considered a fragment of Idealized Algol for which the game semantics of programs can be expressed simply using regular expressions. In this setting, observational equivalence of programs becomes decidable. Consequently, numbers of interesting verification problems become solvable. This development opened up a new direction of research called *Algorithmic game semantics*.

1.5.1 Characterisation of observational equivalence

In [Abramsky and McCusker, 1997] it is shown that observational equivalence of IA is characterised by the equality of the set of complete plays.

A play of a game is *complete* if it is maximal and all question have been answered. A game is *simple* if the complete plays are exactly those in which the initial question has been answered. It can be shown that for any IA type T, $[\![T]\!]$ is a simple game. The following characterisation theorem holds for simple games:

Theorem 1.5.1 (Characterisation Theorem for Simple Game (Abramsky, McCusker 1997)). *Let σ and τ be strategies on a simple game A then:*

$$\sigma \leq \tau \iff \mathit{comp}(\sigma) = \mathit{comp}(\tau)$$

Therefore the game semantics of an IA term is fully determined by the set of complete plays of the strategy denoting it.

1.5.2 Finitary fragments of Idealized algol

We introduce some fragments of the language IA. Firstly, *Finitary Idealized Algol* denotes the recursion-free sub-fragment of IA over finite ground types (i.e. $\mathbb{N}$ is replaced by the set $0..max$ for some fixed value max).

Definition 1.5.2 (ith order IA term). A term $\Gamma \vdash M : T$ of finitary Idealized algol is an ith-order term if any sequent $\Gamma' \vdash N : A$ appearing in the typing derivation of M is such that $\mathsf{ord}(A) \leq i$ and all the variables in Γ' are of order strictly less than i.

IA_i denotes the fragment of finitary Idealized Algol consisting of the collection of ith-order terms.

$\mathsf{IA}_i + \mathsf{while}$ denotes the fragment IA_i augmented with primitive recursion : the formation rules of $\mathsf{IA}_i + \mathsf{while}$ are those of IA_i together with the following rule:

$$\frac{\Gamma \vdash M : \mathtt{bool} \qquad \Gamma \vdash N : \mathtt{com}}{\Gamma \vdash \mathtt{while}\ M \mathtt{do}\ N : \mathtt{com}} \quad \text{where } \forall x \in \Gamma : \mathsf{ord}(x) < i$$

Finally $\mathsf{IA}_i + \mathsf{Y}_j$ where $j < i$ denotes the fragment IA_i augmented with a set of fixed-point iterators $\mathsf{Y}_A : (A \to A) \to A$ for any type A of order j at most. The formation rules of $\mathsf{IA}_i + \mathsf{Y}_j$ are those of IA_i together with the following rule:

$$\frac{\Gamma \vdash \lambda x.M : A \to A}{\Gamma \vdash Y_A M : A} \quad \text{where } \forall x \in \Gamma : \mathsf{ord}(x) < i \text{ and } \mathsf{ord}(A) \leq j$$

We recall the observational equivalence decision problem: given two β-normal forms M and N in a given fragment of IA, does $M \approx N$ hold?

This problem has been investigated and decidability results have been obtained for a complete class of fragments of Idealized Algol. These results help us to understand the limits of Algorithmic Game Semantics. We now present briefly those results.

IA_2 fragment

In Ghica and McCusker [2000], the authors show that in the IA_2 fragment, the set of complete plays are representable by extended regular languages.

Lemma 1.5.3 (Ghica and McCusker 2000). *For any IA_2-term $\Gamma \vdash M : T$, the set of complete plays of $[\![\Gamma \vdash M : T]\!]$ is regular.*

Since equivalence of regular expression is decidable, this shows decidability of observational equivalence of IA_2-terms. In the same paper they show that the same result holds for the $\mathsf{IA}_2 + \mathsf{while}$ fragment.

In Ong [2002], it is shown that observational equivalence is undecidable for $\mathsf{IA}_2 + \mathsf{Y}_1$.

Other fragments of IA

Observational equivalence is decidable for IA_3. This is proved in Ong [2002] by reduction to the *Deterministic Push-down Automata Equivalence* problem. Unfortunately, this result cannot be extended beyond order 3: Murawski showed in Murawski [2003] that the problem is undecidable for IA_i with $i \geq 4$.

However in $\mathsf{IA}_3 + \mathsf{while}$ the problem becomes decidable. More precisely, it is shown in Murawski and Walukiewicz [2005] that for $\mathsf{IA}_2 + \mathsf{while}$ and $\mathsf{IA}_3 + \mathsf{while}$ the problem is EXPTIME.

Moreover in Murawski et al. [2005] it is shown that $\mathsf{IA}_i + Y_0$, for $i = 1, 2, 3$ is as difficult as the DPDA equivalence problem. This problem is decidable [Sénizergues, 2001] but no complexity result is known about it. We only know that it is primitive recursive [Stirling, 2002].

The complete classification

Fragment	pure	+while	+Y0	+Y1
IA_0	PTIME	$\times^{(i)}$	$\times$	$\times$
IA_1	coNP	PSPACE	DPDA EQUIV	$\times$
IA_2	PSPACE	PSPACE	DPDA EQUIV	undecidable
IA_3	EXPTIME	EXPTIME	DPDA EQUIV	undecidable
$\mathsf{IA}_i, i \geq 4$	undecidable	undecidable	undecidable	undecidable

Notes: The $\times$ symbol denotes undefined IA fragments. (i) Adding iteration to IA_0 does not increase the power of the language since variables are forbidden in the language.

The coNP and PSPACE results are due to Murawski [Murawski, 2005].

Chapter 2

SAFE λ-CALCULUS

In Knapik et al. [2002], the authors introduced a restriction on higher-order grammars called *safety* in order to study the infinite hierarchy of trees recognized by a higher-order pushdown automaton. They proved that trees recognized by pushdown automata of level n coincide with trees generated by safe higher-order grammars of level n. This characterisation permited them to prove the decidability of the monadic second-order theory of infinite trees recognized by a higher-order pushdown automaton of any level.

Safety has also appeared in a different form in Damm [1982] under the name *restriction of derived types*. The forthcoming thesis of Jolie de Miranda [de Miranda, 2006] contains a comparison of safety and the restriction of derived types.

More recently, Ong proved in Ong [2006b] that the safety assumption of Knapik et al. [2002] is in fact not necessary. More precisely, the paper shows that the MSO theory of trees generated by order-n recursion schemes is n-EXPTIME complete.

For this particular problem, *safety* happens to be an artificial restriction. However when the *safety* condition is transposed to the simply-typed λ-calculus, it gives some interesting properties. In particular, for safe terms, it becomes unnecessary to rename variables when performing substitution.

This chapter starts with a presentation of the original version of the Safe λ-Calculus where types are required to satisfy a condition called homogeneity. We then give a more general definition which does not require type homogeneity.

2.1 Homogeneous Safe λ-Calculus

2.1.1 Type homogeneity

Let $Types$ be the set of simple types generated by the grammar $A ::= o \mid A \to A$. Any type different from the base type o can be written $(A_1, \cdots, A_n, o)$ for some $n \geq 1$, which is a shorthand for $A_1 \to \cdots \to A_n \to o$ (by convention, $\to$ associates to the right). If $T = (A_1, \cdots, A_n, o)$ then the arity of T, written $arity(T)$, is defined to be n.

Suppose that a ranking function $\mathsf{rank} : Types \longrightarrow (L, \leq)$ is given where $(L, \leq)$ is any linearly ordered set. Possible candidates for the ranking function are:

- $\mathsf{ord} : Types \longrightarrow (\mathbb{N}, \leq)$ with $\mathsf{ord}(o) = 0$ and $\mathsf{ord}(A \to B) = \max(\mathsf{ord}(A) + 1, \mathsf{ord}(B))$;
- $\mathsf{height} : Types \longrightarrow (\mathbb{N}, \leq)$ with $\mathsf{height}(A \to B) = 1 + \max(\mathsf{height}(A), \mathsf{height}(B))$ and $\mathsf{height}(o) = 0$;
- $\mathsf{nparam} : Types \longrightarrow (\mathbb{N}, \leq)$ with $\mathsf{nparam}(o) = 0$ and $\mathsf{nparam}(A_1, \cdots, A_n) = n$;
- $\mathsf{ordernp} : Types \longrightarrow (\mathbb{N} \times \mathbb{N}, \leq)$ with $\mathsf{ordernp}(t) = \langle \mathsf{order}(t), \mathsf{nparam}(t) \rangle$ for $t \in Types$.

Following Knapik et al. [2002], we say that a type is rank-homogeneous if it is o or if it is $(A_1, \cdots, A_n, o)$ with the condition that $\mathsf{rank}(A_1) \geq \mathsf{rank}(A_2) \geq \cdots \geq \mathsf{rank}(A_n)$ and each A_1, $\ldots$, A_n is rank-homogeneous.

Suppose that $\overline{A_1}$, $\overline{A_2}$, $\ldots$, $\overline{A_n}$ are n lists of types, where A_{ij} denotes the jth type of list $\overline{A_i}$ and l_i the size of $\overline{A_i}$, then the notation $A = (\overline{A_1} \mid \cdots \mid \overline{A_r} \mid o)$ means that

- A is the type $(A_{11}, A_{12}, \cdots, A_{1l_1}, A_{21}, \cdots, A_{2l_2}, \cdots A_{n1}, \cdots, A_{nl_n}, o)$
- $\forall i : \forall u, v \in A_i : \mathsf{rank}(u) = \mathsf{rank}(v)$
- $\forall i, j. \forall u \in A_i. \forall v \in A_j. i < j \implies \mathsf{rank}(u) > \mathsf{rank}(v)$

and therefore A is rank-homogenous. This notation organises the A_{ij}s into partitions according to their ranks. Suppose $B = (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o)$, we write $(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid B)$ to mean

$$(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid \overline{B_1} \mid \cdots \mid \overline{B_m} \mid o).$$

From now on, we only consider the rank function ord. We will use the term "homogeneous" to refer to ord-homogeneity.

2.1.2 Safe Higher-order recursion scheme

We now present the original notion of safety introduced in Knapik et al. [2002] as a restriction on higher-order recursion schemes. We introduce briefly the notion of higher-order recursion scheme. The reader is refered to Knapik et al. [2002], de Miranda [2006], Aehlig et al. [2004] for more details.

Suppose that Γ is a set of typed symbol then the set of *applicative terms* written $\mathcal{T}(\Gamma)$ is the closure of Γ under the application rule i.e. if $s : A \to B$ and $t : A$ are in $\mathcal{T}(\Gamma)$ then so is $st : B$.

A higher-order recursion scheme is a deterministic grammar that can be used to define potentially infinite term trees:

Definition 2.1.1 (Higher-order recursion scheme)**.** A *deterministic higher-order grammar* or *higher-order recursion scheme* is a tuple $\mathcal{G} = \langle \Sigma, \mathcal{N}, V, \mathcal{R}, S \rangle$, where

- Σ is a ranked alphabet of terminals of order at most 1,
- V is a finite set of typed variables,
- $\mathcal{N}$ is a finite set of homogeneously-typed non-terminals,

- S a distinguished symbol of $\mathcal{N}$ of ground type, called the start symbol,
- $\mathcal{R}$ is a finite set of production rules, one for each $F : (A_1, \ldots, A_n, o) \in N$, of the form

$$Fz_1 \ldots z_m \to e$$

where z_i is a variable of type A_i and e is an applicative term of type o in $\mathcal{N}(\Sigma \cup \mathcal{N} \cup \{z_1 \ldots z_m\})$. The z_is are called the *parameters* of the rule.

The order of a rewrite rule is the order of the non-terminal symbol appearing on the left hand side of the rule. The order of a grammar is the highest order of its non-terminals.

Safety is a syntactic restriction on higher-order grammars. It can be formulated as follows:

Definition 2.1.2 (Safe Higher-order grammar)**.** Let G be a higher-order grammar G of order n whose non-terminals are of homogeneous type. G is *unsafe* if and only if there is a rewrite rule $Fz_1 \ldots z_m \to e$ where e contains a subterm t such that:

1. t occurs in an operand position in e,
2. t is of order $k > 0$,
3. t contains a parameter of order strictly less than k.

G is *safe* if it is not unsafe.

Let us give an example taken from Knapik et al. [2002]:

Example 2.1.3. Let $f : (o, o, o)$, $g, h : (o, o)$ and $a, b : o$ be Σ constants. The grammar of level 3 with non-terminals $S : o$ and $F : ((o, o), o, o, o)$ and production rules:

$$\begin{aligned} S &\to Fgab \\ F\varphi xy &\to f(\mathcal{F}(F\varphi x)y(hy))(f(\varphi x)y) \end{aligned}$$

is not safe because the term $\mathcal{F}\varphi x : (o, o)$ containing a variable of order 0 occurs at an operand position in the right-hand side expression of the second rule.

On the other hand, the grammar with the following production rules is safe:

$$\begin{aligned} S &\to Fgab \\ F\varphi xy &\to f(\mathcal{F}(F\varphi x)y(hy))(f(\varphi x)y) \end{aligned}$$

Moreover it can be shown that these two grammars are equivalent in the sense that they generate the same infinite tree.

2.1.3 Rules of the Safe λ-Calculus

There is a correspondence between higher-order recursion schemes and the simply-typed λ-calculus. The non-terminals of a recursion scheme can be interpreted as λ-abstractions in the simply-typed λ-calculus. The Σ-constants are interpreted as "constructors" constants (in the sense of constructor used in functional programming languages to represent abstract data-types such as trees). The notions of variable and application are directly transposed to the equivalent notions in the simply-typed λ-calculus. Using this analogy it is possible to derive a version of the safety restriction for the λ-calculus.

The Safe λ-Calculus has been first proposed in Aehlig et al. [2005], a corrected definition appeared in Ong [2005]. The definition that we give here is slightly more general in the sense that we allow the use of Σ-constants of any higher-order type whereas the original definition only allows first order constants.

The **Safe λ-Calculus** is a sub-system of the simply-typed λ-calculus. Typing judgements (or terms-in-context) are of the form:

$$\overline{x_1} : \overline{A_1} \mid \cdots \mid \overline{x_n} : \overline{A_n} \vdash M : B$$

which is shorthand for $x_{11} : A_{11}, \cdots, x_{1r} : A_{1r}, A_{21}, \ldots \vdash M : B$ such that the context variables are listed in decreasing type order and with the condition that $\mathsf{ord}(x_{ik}) < \mathsf{ord}(x_{jl})$ for any k, l and $i < j$.

Valid typing judgements of the system are defined by induction over the following rules, where Δ is a given homogeneously-typed alphabet and Σ is a set of homogeneously-typed constants:

$$(\mathbf{wk})\frac{\Gamma \vdash M : B \qquad \Gamma \subset \Delta}{\Delta \vdash M : B} \qquad (\mathbf{perm})\frac{\Gamma \vdash M : B \qquad \sigma(\Gamma) \text{ homogeneous}}{\sigma(\Gamma) \vdash M : B}$$

$$(\mathbf{\Sigma\text{-const}})\frac{}{\vdash b : A}\ b : A \in \Sigma \qquad (\mathbf{var})\frac{}{x_{ij} : A_{ij} \vdash x_{ij} : A_{ij}}$$

$$(\lambda\mathbf{\text{-abs}})\frac{\overline{x_1} : \overline{A_1} \mid \cdots \mid \overline{x_{n+1}} : \overline{A_{n+1}} \vdash M : B \qquad \mathsf{ord}(\overline{A_{n+1}}) \geq \mathsf{ord}(B) - 1}{\overline{x_1} : \overline{A_1} \mid \cdots \mid \overline{x_n} : \overline{A_n} \vdash \boldsymbol{\lambda}\overline{x_{n+1}} : \overline{A_{n+1}}.M : (\overline{A_{n+1}} \mid B)}$$

$$(\mathbf{app})\frac{\Gamma \vdash M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o) \qquad \Gamma \vdash N_1 : B_{11} \quad \cdots \quad \Gamma \vdash N_l : B_{1l} \qquad l = |\overline{B_1}|}{\Gamma \vdash MN_1 \cdots N_l : (\overline{B_2} \mid \cdots \mid \overline{B_m} \mid o)}$$

$$(\mathbf{app}^+)\frac{\Gamma \vdash M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o) \qquad \Gamma \vdash N_1 : B_{11} \quad \cdots \quad \Gamma \vdash N_l : B_{1l} \qquad l < |\overline{B_1}|}{\Gamma \vdash MN_1 \cdots N_l : (\overline{B} \mid \overline{B_2} \mid \ \cdots \mid \overline{B_m} \mid o)}$$

where $\overline{B_1} = B_{11}, \ldots, B_{1l}, \overline{B}$ with the condition that every variable in Γ has an order strictly greater than $\mathsf{ord}(\overline{B_1})$.

Property 2.1.4 (Basic properties). Suppose $\Gamma \vdash M : B$ is a valid judgment then

(i) B is homogeneous;

(ii) every free variable of M has order at least $\mathsf{ord}(M)$;

(iii) $fv(M) \vdash M : B$,

where $fv(M) \subseteq \Gamma$ denotes the context constituted of the variables in Γ occurring free in M.

Proof. (i) and (ii) are proved by an easy structural induction. (iii) is due to the fact that the weakening rule is the only rule which can introduce a variable not occurring freely in M in the context of a typing judgement. □

We now define a special kind of substitution that performs simultaneous substitution and permits variable capture i.e. that does not rename variables when the substitution is performed on an abstraction.

Definition 2.1.5 (Capture-permitting simultaneous substitution (for homogeneous safe terms))**.** We use the notation $\left[\overline{N}/\overline{x}\right]$ for $[N_1 \ldots N_n/x_1 \ldots x_n]$ and $\overline{y} : \overline{A}$ for $y_1 : A_1, \ldots, y_p : A_p$. A safe term has necessarily one of the forms occurring on the left-hand side of the following equations, where $M,\ N_1, \ldots N_l$ are safe terms. The capture-permitting simultaneous substitution is then defined by:

$$\begin{array}{rcl}
c\left[\overline{N}/\overline{x}\right] & = & c \quad \text{where } c \text{ is a } \Sigma\text{-constant} \\
x_i\left[\overline{N}/\overline{x}\right] & = & N_i \\
y\left[\overline{N}/\overline{x}\right] & = & y \quad \text{if } y \neq x_i \text{ for all } i, \\
(MN_1 \ldots N_l)\left[\overline{N}/\overline{x}\right] & = & (M\left[\overline{N}/\overline{x}\right])(N_1\left[\overline{N}/\overline{x}\right]) \ldots (N_l\left[\overline{N}/\overline{x}\right]) \\
(\lambda\overline{y} : \overline{A}.M)\left[\overline{N}/\overline{x}\right] & = & \lambda\overline{y}.M\left[\overline{N} \restriction I/\overline{x} \restriction I\right] \\
 & & \text{where } I = \{i \in 1..n \mid x_i \notin \overline{y}\}
\end{array}$$

where $\restriction$ is the index filtering operator: if s is a sequence and I a set of indices then $s \restriction I$ is the subsequence of s obtained by keeping only the element in s at positions in I.

This substitution is well-defined for safe terms in the sense that safety is preserved by substitution:

Lemma 2.1.6 (Capture-permitting simultaneous substitution preserves safety)**.** *Let $\Gamma \cup \overline{x} \vdash M$ be a safe term where $\overline{x}$ denotes a list of variables (which do not necessarily belong to the same partition).*

For any safe terms $\Gamma \vdash N_1, \cdots, \Gamma \vdash N_n$, the capture-permitting simultaneous substitution $M[N_1/x_1, \cdots, N_n/x_n]$ is safe. In other words, the following judgment is valid:

$$\Gamma \vdash M[N_1/x_1, \cdots, N_n/x_n]$$

Proof. An easy proof by an induction on the structure of the safe term. □

With the traditional substitution, it is necessary to rename variables when performing substitution on an abstraction in order to avoid possible variable capture. As a consequence, in order to implement substitution one needs to have access to an unbound number of variable names. An interesting property of the homogeneous Safe λ-Calculus is that variable capture never occurs when performing substitution. In other words, the traditional substitution can be safely replaced by the capture-permitting substitution:

Lemma 2.1.7 (No variable capture lemma)**.** *In the Safe λ-Calculus, there is no variable capture when performing the following capture-permitting simultaneous substitution:*

$$M[N_1/x_1, \cdots, N_n/x_n]$$

provided that $\Gamma \cup \overline{x} \vdash M$, $\Gamma \vdash N_1, \cdots, \Gamma \vdash N_n$ are valid judgments.

Proof. We prove the result by induction. The variable, constant and application cases are trivial. For the abstraction case, suppose $M = \lambda \overline{y} : \overline{A}.P$ where $\overline{y} = y_1 \dots y_p$. The capture-permitting simultaneous substitution gives:

$$M\left[\overline{N}/\overline{x}\right] = \lambda \overline{y}.P\left[\overline{N} \restriction I/\overline{x} \restriction I\right] \text{ where } I = \{i \in 1..n \mid x_i \notin \overline{y}\}.$$

By the induction hypothesis there is no variable capture in $P\left[\overline{N} \restriction I/\overline{x} \restriction I\right]$. Hence variable capture can only happen when the variable y_j occurs freely in N_i and x_i occurs freely in P for some $i \in I$ and $j \in 1..p$. In that case, property 2.1.4 (ii) gives:

$$\mathsf{ord}(y_j) \geq \mathsf{ord}(N_i) = \mathsf{ord}(x_i)$$

Moreover $i \in I$ therefore $x_i \notin \overline{y}$ and since x_i occurs freely in P, x_i must also occur freely in the safe term $\lambda \overline{y}.P$. Thus, property 2.1.4 (ii) gives:

$$\mathsf{ord}(x_i) \geq \mathsf{ord}(\lambda y_1 \dots y_p.T) \geq 1 + \mathsf{ord}(y_j) > \mathsf{ord}(y_j)$$

which, together with the previous equation, gives a contradiction. □

2.1.4 Safe β-reduction

We now introduce the notion of safe β-redex and show how to reduce them using the capture-permitting simultaneous substitution. We will then show that a safe β-reduction reduces to a safe term.

In the simply-typed lambda calculus a redex is a term of the form $(\lambda x.M)N$. We generalize this notion to the safe lambda calculus. We call multi-redex a term of the form $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ (it is not required to have $n = l$).

We say that a multi-redex is safe if it respects the formation rules of the Safe λ-Calculus: the multi-redex $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ is a safe redex if the variable $x_1, \dots, x_n$ are abstracted altogether at once using the abstraction rule and if the terms $N_1 \dots N_l$ are applied to the term $\lambda x_1 \dots x_n.M$ at once using either the rule (**app**$^+$) or (**app**). The formal definition follows:

Definition 2.1.8 (Safe redex)**.** A safe redex is a term of the form:

$$(\lambda\overline{x}.M)N_1 \dots N_l$$

such that

- variables $\overline{x} = x_1 \dots x_n$ are abstracted altogether by one occurrence of the rule (**abs**) in the proof tree (possibly followed by the weakening rule). This implies that:

$$\mathsf{ord}(M) - 1 \leq \mathsf{ord}(\overline{x}) = \mathsf{ord}(x_1) = \ldots = \mathsf{ord}(x_n);$$

- the terms $(\lambda\overline{x}.M)$, N_1, N_l are applied together at once using either:
  - the rule (**app**):

$$\frac{\Sigma \vdash \lambda\overline{x}.M : (\overline{B_1}| \dots |\overline{B_m}|o) \quad \Sigma \vdash N_1 \quad \dots \quad \Sigma \vdash N_l \quad l = |\overline{B_1}|}{\Sigma \vdash (\lambda\overline{x}.M)N_1 \dots N_l}(\mathbf{app}),$$

    in which case $n \leq |\overline{B_1}| = l$;
  - or the rule (**app**$^+$):

$$\frac{\Sigma \vdash \lambda\overline{x}.M : (\overline{B_1}| \dots |\overline{B_m}|o) \quad \Sigma \vdash N_1 \quad \dots \quad \Sigma \vdash N_l \quad l < |\overline{B_1}|}{\Sigma \vdash (\lambda\overline{x}.L)N_1 \dots N_l}(\mathbf{app}^+),$$

    in which case $n \leq |\overline{B_1}|$ and no relation holds between n and l.

It is not required to have $n = |\overline{B_1}|$.

Note that there are safe terms of the form $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ with $l > n$. For instance the term $(\lambda f.((\lambda gh.h)a))aa$ of type $o \to o$ for some constant $a : o \to o$ and variables $x : o$ and $f, g, h : o \to o$, can be formed using the (**app**) rule as follows:

$$\frac{\emptyset \vdash (\lambda f.((\lambda gh.h)a)) : (o,o),(o,o),o,o \quad \emptyset \vdash a : o,o \quad \emptyset \vdash a : o,o}{\emptyset \vdash (\lambda f.((\lambda gh.h)a))aa : o,o}(\mathbf{app})$$

Definition 2.1.9 (Safe reduction β_s)**.** For the sake of concision, the following abbreviations are used $\overline{x} = x_1 \dots x_n$, $\overline{N} = N_1 \dots N_l$, and when $n \geq l$, $\overline{x_L} = x_1 \dots x_l$, $\overline{x_R} = x_{l+1} \dots x_n$.

- The relation β_s is defined on the set of safe redex as follows:

$$\begin{array}{rcl} \beta_s & = & \{\ (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l \mapsto \lambda\overline{x_R}.T\left[\overline{N}/\overline{x_L}\right] \\ & & \quad \text{where } (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l \text{ is a safe redex such that } n > l\} \\ & \cup & \{\ (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l \mapsto T\left[\overline{N}/\overline{x}\right] N_{n+1} \dots N_l \\ & & \quad \text{where } (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l \text{ is a safe redex such that } n \leq l\} \end{array}$$

  where the notation $\left[\overline{N}/\overline{x}\right]$ denotes the capture-permitting simultaneous substitution.

- The safe β-reduction, written $\to_{\beta_s}$, is the closure of the relation β_s by compatibility with the formation rules of the Safe λ-Calculus.

We observe that safe β-reduction is a certain kind of multi-steps β-reduction.

Property 2.1.10. $\to_{\beta_s} \subset \twoheadrightarrow_\beta$, i.e. the safe β-reduction relation is included in the transitive closure of the β-reduction relation.

Proof. Suppose that $(M \mapsto N) \in \beta_s$. We show that $M \to_\beta^* N$.

- Suppose that the safe-redex is $M \equiv (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l$ such that $n \leq l$ then:

$$\begin{array}{rcl} M & =_\alpha & (\lambda z_1 \dots z_n.T[z_1, \dots z_n/x_1, \dots x_n])\ N_1 N_2 \dots N_l \\ & & \text{where the } z_i \text{ are fresh variables} \\ & \to_\beta & (\lambda z_2 \dots z_n.T[z_1, \dots z_n/x_1, \dots x_n]\,[N_1/z_1])\ N_2 \dots N_l \\ & & \text{(because the } z_i\text{s do not occur freely in } N_1) \\ & \to_\beta & \dots \\ & \to_\beta & (T[z_1, \dots z_n/x_1, \dots x_n]\,[N_1/z_1] \dots [N_n/z_n])\ N_{n+1} \dots N_l \\ & \to_\beta & (T[N_1 \dots N_l/x_1, \dots x_l])\ N_{n+1} \dots N_l, \end{array}$$

  and since T is safe, the substitution $T[N_1 \dots N_l/x_1, \dots x_l]$ in the last equation can be performed using the capture-permitting substitution. Hence $M \to_\beta^* N$.

- Suppose that $M \equiv (\lambda\overline{x} : \overline{A}.T)N_1 \dots N_l$ such that $n > l$, then necessarily the redex must be formed using the $(\mathbf{app}^+)$ rule. The side-condition of this rules says that the free variables of the terms $N_1, \dots N_l$ have all order strictly greater than $\mathsf{ord}(\overline{x})$, hence the x_is do not occur freely in $N_1, \dots N_l$. Therefore:

$$\begin{array}{rcl} M & = & (\lambda x_1 \dots x_n.T)\ N_1 N_2 \dots N_l \\ & \to_\beta & (\lambda x_2 \dots x_n.T\,[N_1/x_1])\ N_2 \dots N_l \\ & & \text{(for } i \in 2..n,\ x_i \text{ does not occur freely in } N_1) \\ & \to_\beta & \dots \\ & \to_\beta & \lambda x_{l+1} \dots x_n.T\,[N_1/x_1] \dots [N_l/x_l] \\ & & \text{(for } i \in (l+1)..n,\ x_i \text{ does not occur freely in } N_l) \\ & \to_\beta & \lambda x_{l+1} \dots x_n.T[N_1 \dots, N_l/\ x_1, \dots, x_l] \\ & & \text{(the } x_i \text{ do not occur freely in } N_1, \dots N_l), \end{array}$$

  and since T is safe, the substitution $T[N_1 \dots N_l/x_1, \dots x_l]$ in the last equation can be performed using the capture-permitting substitution. Hence $M \to_\beta^* N$.

□

Property 2.1.11. In the simply-typed λ-calculus:

1. $\to_{\beta_s}$ is strongly normalizing.
2. β_s has the unique normal form property.
3. β_s has the Church-Rosser property.

Proof. 1. This is because $\to_{\beta_s} \subset \twoheadrightarrow_\beta$ and, $\to_\beta$ is strongly normalizing in the simply-typed λ-calculus. 2. A term has a safe redex iff it has a β-redex therefore the set of β_s normal forms is equal to the set of β_s normal forms. Hence, the unicity of β-normal form implies the unicity of β_s-normal form. 3. is a consequence of 1 and 2. □

Capture-permitting simultaneous substitution preserves safety (lemma 2.1.6), consequently any safe redex reduces to a safe term:

Lemma 2.1.12 (The safe reduction β_s preserves safety)**.** *If M is safe and $M \to_{\beta_s} N$ then N is safe.*

Proof. It suffices to show that the relation β_s preserves safety. Consider the safe-redex $(s \mapsto t) \in \beta_s$ where $s \equiv (\lambda x_1 \dots x_n.M)N_1 \dots N_l$. We proceed by case analysis on the last rule used to form the redex.

- Suppose the last rule used is (**app**), then necessarily $n \leq l$ and the reduction is

$$(\lambda x_1 \dots x_n.M)N_1 \dots N_l \qquad \mapsto \qquad t \equiv M[N_1/x_1, \cdots, N_n/x_n]\ N_{n+1} \dots N_l.$$

  The first premise of the rule (**app**) tells us that M is safe therefore using lemma 2.1.6 and the application rule we obtain that t is safe.

- Suppose the last rule used is ($\mathbf{app}^+$) and $n > l$ then the reduction is

$$(\lambda \overline{x_L} : \overline{A_L}\ \overline{x_R} : \overline{A_R}.T)\overline{N_L} \qquad \mapsto \qquad t \equiv \lambda \overline{x_R} : \overline{A_R}.T\left[\overline{x_L}/\overline{N_L}\right].$$

  By lemma 2.1.6, $T\left[\overline{x_L}/\overline{N_L}\right]$ is a safe term. Using the rule (**abs**) we derive that t is safe.

- Suppose the last rule used is ($\mathbf{app}^+$) and $n \leq l$ then the reduction is

$$(\lambda x_1 \dots x_n.M)N_1 \dots N_l \qquad \mapsto \qquad t \equiv M[N_1/x_1, \cdots, N_n/x_n]\ N_{n+1} \dots N_l$$

  We conclude that t is safe similarly to case (**app**).

- Rule (**wk**) (**seq**): these cases reduce to one of the previous cases.

□

Remark 2.1.13. $\rightarrow_{\beta_s}$ *does not* preserves un-safety: given two terms S safe and U unsafe of the same type, the term $(\lambda xy.y)US$ is also unsafe but it β_s-reduces to S which is safe.

2.1.5 An alternative system of rules

In this section, we will refine the formation rules given in the previous section. We say that $\Gamma \vdash M : A$ verifies P_i for $i \in \mathbb{Z}$ if the variables in Γ all have orders at least $\mathsf{ord}(A) + i$. We introduce the notation $\Gamma \vdash^i M : A$ for $i \in \mathbb{Z}$ to mean that $\Gamma \vdash M : A$ is a valid judgment satisfying P_i.

We remark that if $\Gamma \vdash M : A$ then the variables in Γ with order strictly smaller than M cannot occur freely in M and therefore it is possible to restrict the context to a smaller number of variables:

Lemma 2.1.14 (Context reduction)**.** *If* $\Gamma \vdash^i M : A$ *then* $\Gamma' \vdash^0 M : A$ *where*

$$\Gamma' = \{z \in \Gamma \mid \mathsf{ord}(z) \geq \mathsf{ord}(M)\} = \Gamma \setminus \{z \in \Gamma \mid \mathsf{ord}(M) + i \leq \mathsf{ord}(z) < \mathsf{ord}(M)\}$$

Proof. If $i \geq 0$ then the result is trivial. Suppose $i < 0$. We proceed by structural induction and case analysis. We only give the details for the application cases (**app**) and ($\mathbf{app}^+$):

- Case of the rule (**app**):

$$(\mathbf{app})\ \frac{\Gamma \vdash M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o) \qquad \Gamma \vdash N_1 : B_{11} \quad \cdots \quad \Gamma \vdash N_l : B_{1l} \qquad l = |\overline{B_1}|}{\Gamma \vdash MN_1 \cdots N_l : (\overline{B_2} \mid \cdots \mid \overline{B_m} \mid o)}$$

  If the conclusion verifies P_i then, for all $z \in \Gamma$:

$$\begin{aligned} \mathsf{ord}(z) \geq 1 + \mathsf{ord}(\overline{B_2}) + i &= 1 + \mathsf{ord}(\overline{B_1}) + \mathsf{ord}(\overline{B_2}) - \mathsf{ord}(\overline{B_1}) + i \\ &= \mathsf{ord}(M) + (\mathsf{ord}(\overline{B_2}) - \mathsf{ord}(\overline{B_1}) + i) \end{aligned}$$

  Therefore the first premise satisfies P_j where $j = \mathsf{ord}(\overline{B_2}) - \mathsf{ord}(\overline{B_1}) + i$. Hence by the induction hypothesis,

$$\Gamma' \vdash^0 M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o)$$

  where $\Gamma' = \Gamma \setminus \{z \in \Gamma \mid \mathsf{ord}(M) + j \leq \mathsf{ord}(z) < \mathsf{ord}(M)\}$.

Similarly, for all $z \in \Sigma$:

$$\begin{aligned} \mathsf{ord}(z) \geq 1 + \mathsf{ord}(\overline{B_2}) + i &= \mathsf{ord}(\overline{B_1}) + (1 + \mathsf{ord}(\overline{B_2}) - \mathsf{ord}(\overline{B_1}) + i) \\ &= \mathsf{ord}(\overline{B_1}) + j + 1 \end{aligned}$$

Hence by the induction hypothesis:

$$\Gamma'' \vdash^0 N_k : B_{1k} \text{ for } k \in 1..l$$

where $\Gamma'' = \Gamma \setminus \{z \in \Gamma \mid \mathsf{ord}(M) + j + 1 \leq \mathsf{ord}(z) < \mathsf{ord}(M)\}$.
Furthermore, $\Gamma'' = \Gamma' \cup \{z \in \Gamma \mid \mathsf{ord}(M) + j = \mathsf{ord}(z)\}$ therefore the weakening rule gives:

$$\Gamma'' \vdash^{-1} M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o)$$

Finally the (**app**) rule gives:

$$\frac{\Gamma'' \vdash^{-1} M : (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o) \quad \Gamma'' \vdash^0 N_1 : B_{11} \quad \ldots \quad \Gamma'' \vdash^0 N_1 : B_{1l}}{\Gamma'' \vdash MN_1 \ldots N_l : (\overline{B_2} \mid \cdots \mid \overline{B_m} \mid o)}$$

such that for all $z \in \Gamma''$:

$$\mathsf{ord}(z) \geq \mathsf{ord}(\overline{B_1}) \quad \geq \quad 1 + \mathsf{ord}(\overline{B_2}) = \mathsf{ord}(MN_1 \ldots N_l)$$

Therefore:

$$\Gamma'' \vdash^0 MN_1 \ldots N_l : (\overline{B_2} \mid \cdots \mid \overline{B_m} \mid o)$$

- (**app**$^+$) The side-condition of the rule (**app**$^+$) ensures that the first premise verifies P_0. The conclusion of the rule has the same order as the first premise therefore the conclusion also verifies P_0.

□

Lemma 2.1.15. *If $\Gamma \vdash^0 M : T$ or $\Gamma \vdash^{-1} M : T$ then there is a valid proof tree showing $\Gamma \vdash M : T$ such that all the judgments appearing in the proof tree verify either P_0 or P_{-1}.*

Proof. Since P_{-1} implies P_0, w.l.o.g. we can assume that the judgment $\Gamma \vdash M : T$ satisfies P_{-1}. We show that there is a proof tree for $\Gamma \vdash M : T$ where all the nodes of the tree verify P_0 or P_{-1}. We proceed by structural induction and case analysis on the last rule used to show $\Gamma \vdash M : T$:

- Axiom (**Σ-const**): the context is empty therefore the sequent verifies P_{-1}.
- Axiom (**var**): the context contains only the variable itself therefore the sequent verifies P_0.
- Rule (**wk**): The premise is $\Delta \vdash M : T$ with $\Delta \subset \Gamma$. Since $\Gamma \vdash M : T$ verifies P_{-1} and $\Delta \subset \Gamma$ the premise must also verify P_{-1}. We can conclude using the induction hypothesis.
- Rule (**perm**): By the induction hypothesis.
- Rule (**abs**): the second premise of the rule guarantees that the first premise verifies P_{-1}.
- Rule (**app**$^+$): The first premise has the same order as the conclusion of the rule therefore the first premise verifies P_0. The side-condition of the rule (**app**$^+$) ensures that all the other premises verify P_0.

- Rule (**app**):

$$(\mathbf{app})\ \frac{\Gamma \vdash M : (\overline{A}\,|B) \qquad \Gamma \vdash N_1 : A_1 \quad \cdots \quad \Gamma \vdash N_l : A_l \qquad l = |\overline{A}|}{\Gamma \vdash^0 MN_1 \cdots N_l : B}$$

Applying lemma 2.1.14 to the first premise we obtain:

$$\Sigma \vdash^0 M : (\overline{A}\,|B) \tag{2.1}$$

where $\Sigma = \{z \in \Gamma \mid \mathsf{ord}(z) \geq \mathsf{ord}((\overline{A}\,|B))\} = \{z \in \Gamma \mid \mathsf{ord}(z) \geq 1 + \mathsf{ord}(\overline{A})\}$.

Applying lemma 2.1.14 to each of the remaining premises gives :

$$\Sigma' \vdash^0 N_i : A_i \quad \text{for all } i \in 1..p$$

where $\Sigma' = \{z \in \Gamma \mid \mathsf{ord}(z) \geq \mathsf{ord}(A_i) = \mathsf{ord}(\overline{A})\} \supseteq \Sigma$.

If the inclusion $\Sigma \subseteq \Sigma'$ is strict then we apply the weakening rule to sequent (2.1):

$$\frac{\Sigma \vdash^0 M : (\overline{A}\,|B)}{\Sigma' \vdash^{-1} M : (\overline{A}\,|B)}(\mathbf{wk})$$

Finally, we obtain the following proof tree:

$$\frac{\dfrac{\Sigma' \vdash^{-1} M : (\overline{A}\,|B) \qquad \Sigma' \vdash^0 N_1 : A_1 \quad \cdots \quad \Sigma' \vdash^0 N_l : A_l \qquad l = |\overline{A}|}{\Sigma' \vdash^0 MN_1 \cdots N_l : B}(\mathbf{app})}{\Gamma \vdash^0 MN_1 \cdots N_l : B}(\mathbf{wk})$$

where the last weakening rules is applied only if the inclusion $\Sigma' \subseteq \Gamma$ is strict.

We can now conclude by applying the induction hypothesis on the sequents $\Sigma' \vdash^{-1} M$, $\Sigma' \vdash^0 N_1$, $\ldots$, $\Sigma' \vdash^0 N_l$.

□

An alternative definition of the homogeneous Safe λ-Calculus

Using the observations that we have just made, we will now derive new rules for the Safe λ-Calculus with homogeneous type. We want a system of rules generating sequents that verify P_0. Also, it must be able to generate intermediate sequents that do not necessarily satisfy P_0 provided that they can be used to produce *in fine* terms satisfying P_0.

Because of the lemma 2.1.15, we know that the only necessary intermediate sequents are those that either satisfy P_0 or P_{-1}. Hence, we can assume by default that premises of the rules all satisfy P_{-1} at least.

First we define an additional rule expressing the fact that P_0 implies P_{-1}:

$$(\mathbf{seq})\ \frac{\Gamma \vdash^0 M : A}{\Gamma \vdash^{-1} M : A}$$

The weakening rule can be rewritten as follows:

$$(\mathbf{wk^0})\ \frac{\Gamma \vdash^0 M : A}{\Gamma, x : B \vdash^0 M : A} \quad \mathsf{ord}(B) \geq \mathsf{ord}(A)$$

$$(\mathbf{wk^{-1}})\ \frac{\Gamma \vdash^{-1} M : A}{\Gamma, x : B \vdash^{-1} M : A} \quad \mathsf{ord}(B) \geq \mathsf{ord}(A) - 1$$

Because of the context reduction lemma, any sequent verifying P_{-1} can be obtained by applying the weakening rule ($\mathbf{wk^{-1}}$) or the rule (**seq**) to another sequent verifying P_0. Therefore, with the exception of these two rules, we only need to use rules whose conclusion sequents verify P_0:

- For the rules (**perm**), (**const**) and (**var**), only the tagging of the sequents changes:

$$(\mathbf{var})\ \frac{}{x : A \vdash^0 x : A} \qquad (\mathbf{\Sigma\text{-const}})\ \frac{}{\vdash^0 b : A}\ b : A \in \Sigma$$

$$(\mathbf{perm})\ \frac{\Gamma \vdash^0 M : B \qquad \sigma(\Gamma) \text{ homogeneous}}{\sigma(\Gamma) \vdash^0 M : B}$$

- (**abs**) The abstraction rule has a side condition expressing the fact that the premise verifies P_0 or P_{-1}. Since this is always true for sequents generated by our new system of rules, we can drop the side condition:

$$(\mathbf{abs})\ \frac{\Gamma | \overline{x} : \overline{A} \vdash^{-1} M : B}{\Gamma \vdash^0 \lambda \overline{x} : \overline{A}.M : (\overline{A}, B)}$$

- (**app**) The application rule has the following form:

$$(\mathbf{app})\ \frac{\Gamma \vdash^{-1} M : (\overline{A}\,|B) \qquad \Gamma \vdash^{-1} N_1 : A_1 \quad \cdots \quad \Gamma \vdash^{-1} N_l : A_l \qquad l = |\overline{A}|}{\Gamma \vdash^0 M N_1 \cdots N_l : B}$$

  Since the first premise verifies P_{-1}, by property 2.1.4(ii) we have:

$$\forall z \in \Gamma : \mathsf{ord}(z) \geq 1 + \mathsf{ord}(\overline{A}) - 1 = \mathsf{ord}(\overline{A}) = \mathsf{ord}(\overline{N})$$

  Hence, all the sequents of the premises but the first must verify P_0. The rule (app) is therefore given by:

$$(\mathbf{app})\ \frac{\Gamma \vdash^{-1} M : (\overline{A}\,|B) \qquad \Gamma \vdash^{0} N_1 : A_1 \quad \cdots \quad \Gamma \vdash^{0} N_l : A_l \qquad l = |\overline{A}|}{\Gamma \vdash^0 M N_1 \cdots N_l : B}$$

- For the application rule ($\mathbf{app}^+$), the type of the sequent in the first premise has the same order as the type of the conclusion premise, and since the conclusion verifies P_0, the first premise must also verify P_0. The side-condition implies that all the other sequents in the premise verify P_0. Moreover since the first premise verifies P_0, the side-condition must hold. Hence the rule becomes:

$$(\mathbf{app}^+)\ \frac{\Gamma \vdash^0 M : (\overline{B_1}\,|\,\cdots\,|\,\overline{B_m}\,|\,o) \qquad \Gamma \vdash^0 N_1 : B_{11} \quad \cdots \quad \Gamma \vdash^0 N_l : B_{1l} \qquad l < |\overline{B_1}|}{\Gamma \vdash^0 M N_1 \cdots N_l : (\overline{B}\,|\,\cdots\,|\,\overline{B_m}\,|\,o)}$$

  where $\overline{B_1} = B_{11}, \ldots, B_{1l}, \overline{B}$. Clearly, this rule can be equivalently stated as:

$$\frac{\Gamma \vdash^0 M : A \to B \qquad \Gamma \vdash^0 N : A}{\Gamma \vdash^0 M N : B}$$

The full set of rules is given in table 2.1.

$$(\mathbf{perm})\ \frac{\Gamma \vdash^0 M : B \qquad \sigma(\Gamma)\ \text{homogeneous}}{\sigma(\Gamma) \vdash^0 M : B} \qquad (\mathbf{seq})\ \frac{\Gamma \vdash^0 M : A}{\Gamma \vdash^{-1} M : A}$$

$$(\mathbf{\Sigma\text{-}const})\ \frac{}{\vdash^0 b : A}\ b : A \in \Sigma \qquad (\mathbf{var})\ \frac{}{x : A \vdash^0 x : A}$$

$$(\mathbf{wk^0})\ \frac{\Gamma \vdash^0 M : A}{\Gamma, x : B \vdash^0 M : A}\quad \mathsf{ord}(B) \geq \mathsf{ord}(A)$$

$$(\mathbf{wk^{-1}})\ \frac{\Gamma \vdash^{-1} M : A}{\Gamma, x : B \vdash^{-1} M : A}\quad \mathsf{ord}(B) \geq \mathsf{ord}(A) - 1$$

$$(\mathbf{app})\ \frac{\Gamma \vdash^{-1} M : (\overline{A}\,|B) \qquad \Gamma \vdash^0 N_1 : A_1 \quad \cdots \quad \Gamma \vdash^0 N_l : A_l \qquad l = |\overline{A}|}{\Gamma \vdash^0 M N_1 \cdots N_l : B}$$

$$(\mathbf{app^+})\ \frac{\Gamma \vdash^0 M : A \to B \qquad \Gamma \vdash^0 N : A}{\Gamma \vdash^0 MN : B}$$

$$(\mathbf{abs})\ \frac{\Gamma|\overline{x} : \overline{A} \vdash^{-1} M : B}{\Gamma \vdash^0 \lambda\overline{x} : \overline{A}.M : (\overline{A}|B)}$$

where $\Gamma|\overline{x} : \overline{A}$ means that the lowest type-partition of the context is $\overline{x} : \overline{A}$.

Tab. 2.1: Alternative rules for the homogeneous safe lambda calculus

2.2 Safe λ-Calculus without the homogeneity constraint

In section 2.1, we have presented a version of the safe lambda calculus where types are required to be homogeneous. We now give a more general version of the safe simply-typed λ-calculus where type homogeneity is not required.

2.2.1 Rules

We use a set of sequents of the form $\Gamma \vdash M : A$ where Γ is the context of the term and A is its type. Let Σ be a set of higher-order constants. We call safe terms any simply-typed lambda term that is typable within the following system of formation rules:

$$(\mathbf{var})\ \frac{}{x : A \vdash x : A} \qquad (\mathbf{const})\ \frac{}{\vdash f : A}\ f \in \Sigma \qquad (\mathbf{wk})\ \frac{\Gamma \vdash M : A}{\Delta \vdash M : A}\ \Gamma \subset \Delta$$

$$(\mathbf{app})\ \frac{\Gamma \vdash M : (A, \ldots, A_l, B) \qquad \Gamma \vdash N_1 : A_1 \quad \ldots \quad \Gamma \vdash N_l : A_l}{\Gamma \vdash M N_1 \ldots N_l : B}\ \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(B)$$

$$(\mathbf{abs})\ \frac{\Gamma \cup \overline{x} : \overline{A} \vdash M : B}{\Gamma \vdash \lambda \overline{x} : \overline{A}.M : (\overline{A}, B)} \qquad \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(\overline{A}, B)$$

Remark:

- $(\overline{A}, B)$ denotes the type $(A_1, A_2, \ldots, A_n, B)$;
- all the types appearing in the rule are not required to be homogeneous (for instance it is possible to have $\mathsf{ord}(A_l) < \mathsf{ord}(B)$ in rule (**app**)) ;
- the environment $\Gamma \cup \overline{x} : \overline{A}$ is not stratified, in particular, variables in $\overline{x}$ do not necessarily have the same order;
- in the abstraction rule, the side-condition imposes that at least all variables of the lowest order in the context are abstracted. Variables of greater order can also be abstracted together with the lowest order variables and, in contrast to the homogeneous safe lambda calculus, there is no constraint on the order in which these variables are abstracted;

Example 2.2.1. Suppose $x : o$, $f : o \to o$ and $\varphi : (o \to o) \to o$ then the term

$$\vdash \lambda x f \varphi.\varphi : o \to (o \to o) \to ((o \to o) \to o) \to (o \to o) \to o$$

is a valid non homogeneous safe term.

Side-remark: safety is preserved by full η-expansion. Indeed, consider the safe term $\Gamma \vdash M : (A_1, \ldots, A_l, o)$ where $(A_1, \ldots, A_l, o)$ is not homogeneous. Its full η-expansion is $\lambda x_1..x_l.M x_1 \ldots x_l$ for some variables $x_1 : A_1, \ldots, x_l : A_l$ fresh in M. For all $i \in 1..l$ we have $\Gamma, \Sigma \vdash x_i : A_i$ where $\Sigma = \{x_1 : A_1, \cdots x_l : A_l\}$. Applying (**app**) we obtain $\Gamma, \Sigma \vdash M x_1 \ldots x_l$ and by the (abs) rule we get

$$\Gamma \vdash \lambda x_1 : A_1 \ldots x_l : A_l.M x_1 \ldots x_l.$$

Lemma 2.2.2 (Context reduction)**.** *If $\Gamma \vdash M : B$ is a valid judgment then*

1. *$fv(M) \vdash M : B$*
2. *every variable in* Γ occurring free in M *has order at least $ord(M)$.*

where $fv(M)$ denotes the context constituted of the free variables occurring in M.

Proof. (i) Suppose that some variable x in Γ does not occur free in M, then necessarily x has been introduced in the context using the weakening rule. Hence $\Gamma \setminus \{x\} \vdash M$ must also be typable. (ii) An easy structural induction. □

The converse of this lemma is not true: consider the simply-typed term $\lambda yz.(\lambda x.y)z$ with $x, y, z : o$. This term is closed therefore it satisfies property (i) and (ii) of lemma 2.2.2. However it is not typable by the rules of the safe lambda-calculus since the subterm $\lambda x.y$ is not safe.

2.2.2 Substitution in the safe lambda calculus

The traditional notion of substitution, on which the λ-calculus is based, is defined as follows:

Definition 2.2.3 (Substitution)**.**

$$\begin{array}{rcl} c\,[t/x] & = & c \quad \text{where } c \text{ is a } \Sigma\text{-constant,} \\ x\,[t/x] & = & t \\ y\,[t/x] & = & y \quad \text{for } x \neq y, \\ (M_1 M_2)\,[t/x] & = & (M_1\,[t/x])(M_2\,[t/x]) \\ (\lambda x.M)\,[t/x] & = & \lambda x.M \\ (\lambda y.M)\,[t/x] & = & \lambda z.M\,[z/y]\,[t/x] \text{ where } z \text{ is a fresh variable and } x \neq y. \end{array}$$

In the setting of the safe lambda calculus, the notion of substitution can be simplified. Indeed, similarly to what we observe in the homogeneous Safe λ-Calculus, we remark that for safe λ-terms there is no need to rename variables when performing substitution:

Lemma 2.2.4 (No variable capture lemma)**.** *There is no variable capture when performing substitution on a safe term.*

This is the counterpart of lemma 2.1.7. The proof (which does not rely on homogeneity) is the same. Consequently, in the safe lambda calculus setting, we can omit to rename variable when performing substitution. The equation

$$(\lambda x.M)\,[t/y] = \lambda z.M\,[z/x]\,[t/y] \text{ where } z \text{ is a fresh variable}$$

becomes

$$(\lambda x.M)\,[t/y] = \lambda x.M\,[t/y]\,.$$

Unfortunately, this notion of substitution is still not adequate for the purpose of the safe simply-typed lambda calculus. The problem is that performing a single β-reduction on a safe term will not necessarily produce another safe term.

The solution consists in reducing several consecutive β-redex at the same time until we obtain a safe term. To achieve this, we introduce the *simultaneous substitution*, a generalization of the standard substitution given in definition 2.2.3.

Definition 2.2.5 (Simultaneous substitution)**.** The expression $\left[\overline{N}/\overline{x}\right]$ is an abbreviation for $[N_1 \dots N_n/x_1 \dots x_n]$:

$$\begin{array}{rcl} c\,\left[\overline{N}/\overline{x}\right] & = & c \quad \text{where } c \text{ is a } \Sigma\text{-constant,} \\ x_i\,\left[\overline{N}/\overline{x}\right] & = & N_i \\ y\,\left[\overline{N}/\overline{x}\right] & = & y \quad \text{if } y \neq x_i \text{ for all } i, \\ (MN)\,\left[\overline{N}/\overline{x}\right] & = & (M\,\left[\overline{N}/\overline{x}\right])(N\,\left[\overline{N}/\overline{x}\right]) \\ (\lambda x_i.M)\,\left[\overline{N}/\overline{x}\right] & = & \lambda x_i.M\,[N_1 \dots N_{i-1} N_{i+1} \dots N_n/x_1 \dots x_{i-1} x_{i+1} \dots x_n] \\ (\lambda y.M)\,\left[\overline{N}/\overline{x}\right] & = & \lambda z.M\,[z/y]\,\left[\overline{N}/\overline{x}\right] \\ & & \text{where } z \text{ is a fresh variables and } y \neq x_i \text{ for all } i. \end{array}$$

In general, variable capture should be avoided, this explains why the definition of simultaneous substitution uses auxiliary fresh variables. However in the current setting, lemma 2.2.4 can clearly be transposed to the simultaneous substitution, therefore there is no need to rename variables.

The notion of substitution that we need is therefore the *capture-permitting simultaneous substitution* defined as follows:

Definition 2.2.6 (Capture-permitting simultaneous substitution)**.** We use the notation $\left[\overline{N}/\overline{x}\right]$ for $[N_1 \dots N_n/x_1 \dots x_n]$:

$$\begin{array}{rcl} c\left[\overline{N}/\overline{x}\right] & = & c \quad \text{where } c \text{ is a } \Sigma\text{-constant,} \\ x_i\left[\overline{N}/\overline{x}\right] & = & N_i \\ y\left[\overline{N}/\overline{x}\right] & = & y \quad \text{where } x \neq y_i \text{ for all } i, \\ (M_1M_2)\left[\overline{N}/\overline{x}\right] & = & (M_1\left[\overline{N}/\overline{x}\right])(M_2\left[\overline{N}/\overline{x}\right]) \\ (\lambda x_i.M)\left[\overline{N}/\overline{x}\right] & = & \lambda x_i.M\left[N_1 \dots N_{i-1}N_{i+1} \dots N_n/x_1 \dots x_{i-1}x_{i+1} \dots x_n\right] \\ (\lambda y.M)\left[\overline{N}/\overline{x}\right] & = & \lambda y.M\left[\overline{N}/\overline{x}\right] \text{ where } y \neq x_i \text{ for all } i. \qquad (\star) \end{array}$$

The symbol $(\star)$ identifies the equation which has changed compared to the previous definition.

Lemma 2.2.7 (Substitution preserves safety)**.**

$$\Gamma \cup \overline{x} : \overline{A} \vdash M : T \quad \textit{and} \quad \Gamma \vdash N_k : B_k,\ k \in 1..n \qquad \textit{implies} \qquad \Gamma \vdash M[\overline{N}/\overline{x}] : T$$

Proof. Suppose that $\Gamma \cup \overline{x} : \overline{A} \vdash M : T$ and $\Gamma \vdash N_k : B_k$ for $k \in 1..n$.

We prove $\Gamma \vdash M[\overline{N}/\overline{x}]$ by induction on the size of the proof tree of $\Gamma \cup \overline{x} : \overline{A} \vdash M : T$ and by case analysis on the last rule used. We only give the proof for the abstraction case. If $\Gamma \cup \overline{x} : \overline{A} \vdash \lambda\overline{y} : \overline{C}.P : (\overline{C}|D)$ where $\Gamma \cup \overline{x} : \overline{A} \cup \overline{y} : \overline{C} \vdash P : D$, then by the induction hypothesis $\Gamma \cup \overline{y} : \overline{C} \vdash P\left[\overline{N}/\overline{x}\right] : D$. Applying the rule (**abs**) gives $\Gamma \vdash \lambda\overline{y} : \overline{C}.P\left[\overline{N}/\overline{x}\right]$. □

2.2.3 Safe-redex

In the simply-typed lambda calculus a redex is a term of the form $(\lambda x.M)N$. We generalize this definition to the safe lambda calculus:

Definition 2.2.8 (Safe redex)**.** We call safe redex a term of the form $(\lambda\overline{x}.M)N_1 \dots N_l$ such that:

- $\Gamma \vdash (\lambda\overline{x}.M)N_1 \dots N_l$;
- the variable $\overline{x} = x_1 \dots x_n$ are abstracted altogether by one occurrence of the rule (**abs**) in the proof tree;
- the terms $(\lambda\overline{x}.M)$, N_1, N_l are applied together at once using the (**app**) rule :

$$\frac{\Sigma \vdash \lambda\overline{x}.M \quad \Sigma \vdash N_1 \quad \dots \quad \Sigma \vdash N_l}{\Sigma \vdash (\lambda\overline{x}.L)N_1 \dots N_l}(\mathbf{app})$$

  (and consequently each N_i is safe);
- $l \leq n$.

Note that the condition $l \leq n$ is not too restrictive. Indeed if $l > n$ then the application rule is "wider" than the abstraction and therefore it can be replaced by an application of exactly n terms followed by another application for the remaining terms $N_{n+1}, \dots, N_l$.

The relation β_s is defined on safe-redex: $(s \mapsto t) \in \beta_s$ iff $s \equiv (\lambda\overline{x}.M)N_1 \dots N_l$ is a safe redex and $t \equiv \lambda x_l \dots x_n.M\left[N_1 \dots N_l/x_1 \dots x_l\right]$. The safe β-reduction denoted by $\rightarrow_{\beta_s}$ is the closure of the relation β_s by compatibility with the formation rules of the Safe λ-Calculus. It is straightforward to show, as we did for the homogeneous Safe λ-Calculus, that $\rightarrow_{\beta_s} \subseteq \rightarrow_\beta^*$.

Lemma 2.2.9. *A safe redex reduces to a safe term.*

This lemma, which is a consequence of lemma 2.2.7, is the counterpart of lemma 2.1.12 in the homogeneous safe lambda calculus. Their proofs are identical.

2.2.4 Particular case of homogeneously-safe lambda terms

In this section, we derive a new set of rules by adding the type-homogeneity restriction to the non-homogenous safe lambda calculus.

We recall the definition of type-homogeneity from section 2.1: a type $(A_1, A_2, \dots A_n, o)$ is said to be homogeneous whenever $\mathsf{ord}(A_1) \geq \mathsf{ord}(A_2) \geq \dots \geq \mathsf{ord}(A_n)$ and each of the A_i is homogeneous. A term is said to be homogeneous if its type is homogeneous.

We now impose type-homogeneity to all the sequents present in the rules of the Safe λ-Calculus: we say that a term is *homogeneously-safe* if there is a proof tree showing its safety in which all sequents are of homogenous type. Consequently a homogeneously-safe term is safe and has an homogenous type.

We say that $\Gamma \vdash M : A$ verifies P_i for $i \in \mathbb{Z}$ if all the variables in Γ have order at least $\mathsf{ord}(A)+i$. Lemma 2.2.2 can then be restated as follows:

Lemma 2.2.10 (Context reduction)**.** *If $\Gamma \vdash M : A$ then the sequent $fv(M) \vdash M : A$ is valid and satisfies P_0.*

We now prove that if we impose the homogeneity of types, the set of rules of the non-homogenous Safe λ-Calculus and the rules of table 2.1 are equivalent. We recall that in the system of rules of table 2.1, if the sequent $\Gamma \vdash^i M : A$ is valid for some $i \in \mathbb{Z}$ then all the variables in Γ have orders at least $\mathsf{ord}(A) + i$.

Proposition 2.2.11 (Homogeneity restriction)**.** *Let $k \in \{0, -1\}$. The sequent $\Gamma \vdash M : A$ is valid, homogeneously-safe and satisfies P_k if and only if the sequent $\Gamma \vdash^k M : A$ is valid in the system of rules of table 2.1.*

Proof. If: An easy induction by case analysis on the last rule used to derive $\Gamma \vdash^0 M : A$.

Only if: Consider an homogeneously-safe term $\Gamma \vdash S : T$ satisfying P_0. We proceed by induction and case analysis on the last rule used to derive $\Gamma \vdash S : T$. We only give the details for the application and abstraction case:

- **Abstraction.** We recall the abstraction rule:

$$(\mathbf{abs}) \quad \frac{\Gamma \cup \overline{x} : \overline{A} \vdash M : B}{\Gamma \vdash \lambda \overline{x} : \overline{A}.M : (\overline{A}, B)} \qquad \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(\overline{A}, B)$$

  Type homogeneity requires that for all i: $\mathsf{ord}(x_i) = \mathsf{ord}(A_i) \geq \mathsf{ord}(B) - 1$. Therefore the premise of the rule verifies P_{-1}. Using the induction hypothesis we have:

$$\Gamma \cup \overline{x} : \overline{A} \vdash^{-1} M : B. \tag{2.2}$$

  We now partition the context Γ according to the order of the variables. The partitions are written in decreasing order of type order. The notation $\Gamma | \overline{x} : \overline{A}$ means that $\overline{x} : \overline{A}$ is the lowest partition of the context. We also use the notation $(\overline{A}|B)$ to denote the homogeneous type $(A_1, A_2, \dots A_n, B)$ where $\mathsf{ord}(A_1) = \mathsf{ord}(A_2) = \dots \mathsf{ord}(A_n) \geq \mathsf{ord}(B) - 1$.

  Suppose that we abstract a single variable x, then in order to respect the side condition, we need to abstract all variables of order less or equal to $\mathsf{ord}(x)$. In particular we need to abstract the partition of the order of x. Moreover to respect type homogeneity, we need to abstract variables of the lowest order first.

  Hence $\overline{x}$ must contain at least the lowest variable partition (all the variables of the lowest order). If $\overline{x}$ contains variables of different order, then the instance of the abstraction rule can be replaced by consecutive instances of the abstraction rule, one for each of the different variable order in $\overline{x}$. Therefore, without loss of generality, we can assume that $\overline{x}$ only contains the lowest partition, that is to say, $\overline{x}$ *is* the lowest partition.

  The sequent 2.2 therefore becomes:

$$\Gamma | \overline{x} : \overline{A} \vdash^{-1} M : B.$$

We conclude by applying the abstraction rule of table 2.1:

$$(\mathbf{abs}) \quad \frac{\Gamma|\overline{x} : \overline{A} \vdash^{-1} M : B}{\Gamma \vdash^{0} \lambda\overline{x} : \overline{A}.M : (\overline{A}|B)}$$

- **Application.** We recall the application rule:

$$(\mathbf{app}) \ \frac{\Gamma \vdash M : (A, \ldots, A_l, B) \qquad \Gamma \vdash N_1 : A_1 \quad \ldots \quad \Gamma \vdash N_l : A_l}{\Gamma \vdash MN_1 \ldots N_l : B} \quad \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(B)$$

The term in the conclusion is homogeneously safe therefore the term in the first premise must be of homogeneous type. This implies that $\mathsf{ord}(A_1) \geq \ldots \geq \mathsf{ord}(A_l) \geq \mathsf{ord}(B) - 1$. Furthermore, we can make the assumption that $\mathsf{ord}(A_1) = \ldots = \mathsf{ord}(A_l) = \mathsf{ord}(\overline{A})$ (it is always possible to replace an instance of the application rule by several consecutive instances of this kind).

By lemma 2.2.10, we have for all $i \in 1..l$:

$$fv(N_i) \vdash N_i : A_i \text{ is valid and satisfies } P_0.$$

Let $\Sigma = \bigcup_{i=1..p} fv(N_i)$. Since $\mathsf{ord}(A_1) = \ldots = \mathsf{ord}(A_l)$, by applying the weakening rule we get for all $i \in 1..p$:

$$\Sigma \vdash N_i : A_i \text{ is valid and satisfies } P_0.$$

Applying lemma 2.2.10 to the term M we have:

$$fv(M) \vdash M : (A_1, \ldots, A_l, B) \text{ is valid and satisfies } P_0.$$

The weakening rule (**wk**) then gives: $fv(M) \cup \Sigma \vdash M : (A_1, \ldots, A_l, B)$. Since $\Sigma \vdash N_i : A_i$ satisfies P_0, for any $z \in \Sigma$ we have $\mathsf{ord}(z) \geq \mathsf{ord}(A_i) = \mathsf{ord}((A_1, \ldots, A_l, B)) - 1$. Hence:

$$fv(M) \cup \Sigma \vdash M : (A_1, \ldots, A_l, B) \text{ is valid and satisfies } P_{-1}. \tag{2.3}$$

Similarly, for all $i \in 1..p$, the weakening rule gives $fv(M) \cup \Sigma \vdash N_i : A_i$. Since $fv(M) \vdash M : (A_1, \ldots, A_l, B)$ satisfies P_0, for any $z \in fv(M)$ we have $\mathsf{ord}(z) \geq \mathsf{ord}(M) \geq \mathsf{ord}(A_i)$. Hence:

$$fv(M) \cup \Sigma \vdash N_i : A_i \text{ is valid and satisfies } P_0. \tag{2.4}$$

Let us define the context $\Sigma' = fv(M) \cup \Sigma$. Using the induction hypothesis on equation 2.3 and 2.4 we have:

$$\Sigma' \vdash^{-1} M : (A_1, \ldots, A_l, B) \qquad \text{and} \qquad \Sigma' \vdash^{0} N_i : A_i \text{ for all } i \in 1..l.$$

We consider the following two sub-cases:

- If $A_1, \ldots, A_l$ forms a type partition then we can apply rule (**app**) of table 2.1:

$$\frac{\Sigma' \vdash^{-1} M : \overline{A}|B \qquad \Sigma' \vdash^{0} N_1 : A_1 \quad \ldots \quad \Sigma' \vdash^{0} N_l : A_l \quad l = |\overline{A}|}{\Sigma' \vdash^{0} MN_1 \ldots N_l : B} \quad (\mathbf{app})$$

  where $\overline{A} = A_1, \ldots, A_l$.
- Suppose that $A_1, \ldots, A_l$ does not form a type partition, then we have

$$\mathsf{ord}(A_1) = \ldots = \mathsf{ord}(A_l) = \mathsf{ord}(B) - 1.$$

The side condition in the original instance of the application rule says that for any variable y in Γ we have

$$\mathsf{ord}(y) \geq \mathsf{ord}(B) = 1 + \mathsf{ord}(A_l) = \mathsf{ord}((A_1, \ldots, A_l, B)) = \mathsf{ord}(M).$$

In particular the variables in $\Sigma' \subseteq \Gamma$ are of order greater than $\mathsf{ord}(M)$ and consequently the sequent $\Sigma' \vdash M : (A, \ldots, A_l, B)$ verifies P_0. The induction hypothesis then gives:

$$\Sigma' \vdash^0 M : (A, \ldots, A_l, B)$$

By using l consecutive instances of the rules $(\mathbf{app}^+)$ from table 2.1 we get:

$$\dfrac{\dfrac{\dfrac{\Sigma' \vdash^0 M : (A_1, \ldots, A_l, B) \qquad \Sigma' \vdash^0 N_1 : A_1}{\Sigma' \vdash^0 MN_1 : (A_2, \ldots, A_l, B)}\,(\mathbf{app}^+)}{\vdots}\,(\mathbf{app}^+)}{\Sigma' \vdash^0 MN_1 \ldots N_l : B}\,(\mathbf{app}^+)$$

In both cases we have proved that $\Sigma' \vdash^0 MN_1 \ldots N_l : B$ is a valid sequent.

Clearly $\Sigma' \subseteq \Gamma$ since $fv(M) \subseteq \Gamma$ and $\Sigma' = \bigcup_{i \in 1..l} fv(N_i) \subseteq \Gamma$. Suppose that $\Sigma' = \Gamma$ then the proof is done. Suppose that $\Sigma' \subset \Gamma$, then the side condition in the original instance of the application rule says that all the variables in Γ have order greater or equal to $\mathsf{ord}(B)$, we can therefore apply the weakening rule $(\mathbf{wk^0})$ of table 2.1 exactly $|\Gamma \setminus \Sigma'|$ times and get:

$$\dfrac{\Sigma' \vdash^0 MN_1 \ldots N_l : B}{\Gamma \vdash^0 MN_1 \ldots N_l : B}\,(\mathbf{wk^0}).$$

□

2.2.5 Examples

Example 1

Let $f, g : o \to o$, $x, y : o \to o$, $\Gamma = g : o \to o$ and $\Gamma' = g : o \to o, y : o$. The term $(\lambda fx.x)gy$ is safe. One possible proof tree is:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \lambda fx.x} \qquad \dfrac{}{\Gamma \vdash g}(\mathbf{app})}{\Gamma \vdash (\lambda fx.x)g}(\mathbf{wk}) \qquad \dfrac{}{\Gamma' \vdash y}(\mathbf{app})}{\Gamma' \vdash (\lambda fx.x)gy}$$

Here is another proof for the same judgment:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \lambda fx.x}}{\Gamma' \vdash \lambda fx.x}(\mathbf{wk}) \qquad \dfrac{}{\Gamma' \vdash g} \qquad \dfrac{}{\Gamma' \vdash y}(\mathbf{app})}{\Gamma' \vdash (\lambda fx.x)gy}$$

We see on this particular example that there may exist different proof trees deriving the same judgment.

Example 2 - Damien Sereni's SCT counter-example

In Sereni [2005], the following counter-example is given to show that not all simply-typed terms are size-change terminating (see Lee et al. [2001] for a definition of size-change termination):

$$E = (\lambda a.a(\lambda b.a(\lambda cd.d)))(\lambda e.e(\lambda f.f))$$

where:

$$\begin{array}{rcl} a & : & \sigma \to \mu \to \mu \\ b & : & \tau \to \tau \\ c & : & \tau \to \tau \\ d & : & \mu \\ e & : & \sigma = (\tau \to \tau) \to \mu \to \mu \\ f & : & \tau \end{array}$$

and τ, μ and σ are type variables.

This example shows that the rules of the Safe λ-Calculus without the homogeneity restriction generates a class of terms that strictly contains the class of terms generated by the rules of the homogeneous Safe λ-Calculus of section 2.1.

Indeed, for E to be an homogeneous safe lambda term, in the sense of the rules of section 2.1, τ and μ must be homogeneous types and the variables a, b, c, d, e, f must be homogeneously typed. This implies that $\mathsf{ord}(\tau) \geq \mathsf{ord}(\mu) - 1$. Conversely, if this condition is met then $\vdash E : \mu \to \mu$ is a valid judgement of the *homogeneous* Safe λ-Calculus.

In the Safe λ-Calculus *without* the homogeneity constraint, however, the judgement $\vdash E : \mu \to \mu$ is always valid whatever the types μ and τ are.

Chapter 3

COMPUTATION TREES, TRAVERSALS AND GAME SEMANTICS

The aim of this chapter is to develop tools that will be used in the next chapter to give a characterisation of the game semantics of the Safe λ-Calculus. Establishing such a characterisation is complicated by the fact that Safety is a syntactic restriction whereas Game Semantics is by nature a syntax-independent semantics. We therefore need to make an explicit correspondence between the game denotation of a term and its syntax.

Our approach follows ideas recently introduced in Ong [2006b], mainly the notion of computation tree of a simply-typed λ-term and traversals over the computation tree. A computation tree can be regarded as an abstract syntax tree (AST) of the η-long normal form of a term. A traversal is a justified sequence of nodes of the computation tree respecting some formation rules. Traversals are used to describe computations. An interesting property is that the *P-view* of a traversal (computed in the same way as P-view of plays in Game Semantics) is a path in the computation tree.

The main result that we will prove in this chapter is called the *Correspondence Theorem* (theorem 3.2.23). It states that traversals over the computation tree are just representations of the uncovering of plays in the strategy-denotation of the term. Hence there is an isomorphism between the strategy denotation of a term and its revealed game denotation (i.e. its strategy denotation where internal moves are not hidden after composition). This theorem permits us to explore the effect that a given syntactic restriction has on the strategy denotating a term.

To really make use of the Correspondence Theorem, it will be necessary to restate it in the standard game-semantic framework in which internal moves are hidden. For that purpose, we will define a *reduction* operation on traversals responsible of eliminating the "internal nodes" of the computation. This leads to a correspondence between the standard game denotation of a term and the set of reductions of traversals over its computation tree. Fortunately, the reduction process preserves the good properties of traversals. This is guaranteed by the facts that the P-view of the reduction of a traversal is equal to the reduction of the P-view of the traversal, and the O-view of a traversal is the same as the O-view of its reduction (lemma 3.1.20).

Related works: Traversals of a computation tree provide a way to perform *local computation* of β-reductions as opposed to a global approach where the β-reduction is implemented by performing substitutions. A notion of local computation of β-reduction has been investigated in Danos and Regnier [1993] through the use of special graphs called "virtual nets" that embed the lambda-calculus.

In Asperti et al. [1994], a notion of graph based on Lamping's graphs [Lamping, 1990] is introduced to represent λ-terms. The authors unify different notions of paths (regular, legal, consistent and persistent paths) that have appeared in the literature as ways to implement graph-based reduction of lambda-expressions. We can regard a traversal as an alternative notion of path adapted to the graph representation of λ-expressions given by computation trees.

3.1 Computation tree

We work in the general setting of the simply-typed λ-calculus extended with a fixed set Σ of higher-order constants.

3.1.1 η-long normal form and computation tree

The η-long normal form appeared in [Jensen and Pietrzykowski, 1976] and [Huet, 1975] under the names *long reduced form* and *η-normal form* respectively. It was then investigated in [Huet, 1976] under the name *extensional form*.

The η-expansion of $M : A \to B$ is defined to be the term $\lambda x.Mx : A \to B$ where $x : A$ is a fresh variable. A term $M : (A_1, \ldots, A_n, o)$ can be expanded in several steps into $\lambda\varphi_1 \ldots \varphi_l.M\varphi_1 \ldots \varphi_l$ where the $\varphi_i : A_i$ are fresh variables. The η-normal form of a term is obtained by hereditarily η-expanding every subterm occurring at an operand position.

Definition 3.1.1 (η-long normal form)**.** A simply-typed term is either an abstraction or it can be written uniquely as $s_0 s_1 \ldots s_m$ where $m \geq 0$ and s_0 is a variable, a Σ-constant or an abstraction. The η-long normal form of a term M, written $\lceil M \rceil$ or sometimes $\eta_{\mathsf{nf}}(M)$, is defined as follows:

$$\begin{aligned}
\lceil \alpha s_1 \ldots s_m : (A_1, \ldots, A_n, o) \rceil &= \lambda\overline{\varphi}.\alpha \lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil && \text{with } m, n \geq 0 \\
\lceil \lambda x.s \rceil &= \lambda x. \lceil s \rceil \\
\lceil (\lambda x.s_0) s_1 \ldots s_m : (A_1, \ldots, A_n, o) \rceil &= \lambda\overline{\varphi}.(\lambda x. \lceil s_0 \rceil) \lceil s_1 \rceil \ldots \lceil s_m \rceil \lceil \varphi_1 \rceil \ldots \lceil \varphi_n \rceil && \text{with } m \geq 1, n \geq 0
\end{aligned}$$

where x and each $\varphi_i : A_i$ are variables and α is either a variable or a constant.

For $n = 0$, the first clause in the definition becomes:

$$\lceil x s_1 \ldots s_m : o \rceil = \lambda.x \lceil s_1 \rceil \lceil s_2 \rceil \ldots \lceil s_m \rceil,$$

and we deliberately keep the *dummy* lambda in the right-hand side of the equation because it will play an important role in the correspondence with game semantics.

Note that our version of the η-long normal form is defined not only for β-normal terms but also for any simply-typed term. Moreover it is defined in such a way that β-normality is preserved:

Lemma 3.1.2. *The η-long normal form of a term in β-normal form is also in β-normal form.*

Proof. By induction on the structure of the term and the order of its type. *Base case*: If $M = x : 0$ then $\lceil x \rceil = \lambda.x$ is also in β-nf. *Step case*: The case $M = \lceil (\lambda x.s_0) s_1 \ldots s_m : (A_1, \ldots, A_n, o) \rceil$ with $m > 0$ is not possible since M is in β-normal form. Suppose $M = \lambda x.s$ then s is in β-nf. By the induction hypothesis $\lceil s \rceil$ is also in β-nf and therefore so is $\lceil M \rceil = \lambda x. \lceil s \rceil$.

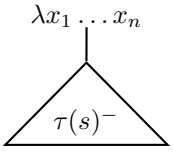
Suppose $M = \alpha s_1 \ldots s_m : (A_1, \ldots, A_n, o)$. Let i, j range over $1..n$ and $1..m$ respectively. The s_j are in β-nf and the φ_i are variables of order smaller than M, therefore by the induction hypothesis the $\lceil \varphi_i \rceil$ and the $\lceil s_j \rceil$ are in β-nf. Hence $\lceil M \rceil$ is also in β-nf. □

The computation tree of term is a certain tree representation of its η-long normal form. It is defined as follows:

Definition 3.1.3 (Computation tree)**.** For any term M in η-normal form we define the tree $\tau(M)$ by induction on the structure of the term. Since M is in η-normal form, there are only two cases: M is either an abstraction or it is of ground type and can be written uniquely as $s_0 s_1 \ldots s_m : 0$ where $m \geq 0$, s_0 is a variable, a constant or an abstraction and each of the s_j for $j \in 1..m$ is in η-normal form:

- the tree for $\lambda x_1 \ldots x_n.s$ where $n \geq 0$ and s is not an abstraction is:

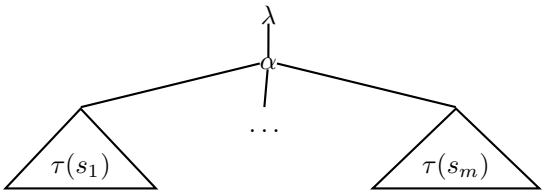
$\tau(\lambda x_1 \ldots x_n.s) =$



where $\tau(s)^-$ denotes the tree obtained after deleting the root of $\tau(s)$.

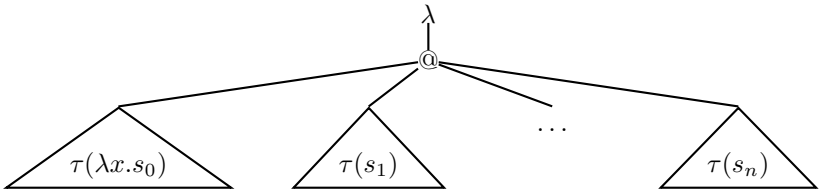
- the tree for $\alpha s_1 \dots s_m : o$ where $m \geq 0$ and α is a variable or constant is:

$\tau(\alpha s_1 \dots s_m) =$



- the tree for $(\lambda x.s_0)s_1 \dots s_n : o$ where $n \geq 1$ is:

$\tau((\lambda x.s_0)s_1 \dots s_n) =$



The *computation tree* of a simply-typed term M (whether or not in η-normal form) is written $\tau(M)$ and defined to be $\tau(M) = \tau(\eta_{\mathsf{nf}}(M))$.

The nodes (and leaves) of the tree are of three kinds:

- λ-nodes labelled $\lambda\overline{x}$ (note that a λ-node represents several consecutive variable abstractions),
- application nodes labelled @,
- variable or constant nodes labelled α for some constant or variable α.

We write N for the set of nodes of $\tau(M)$, N_Σ for the set of Σ-labelled nodes, $N_@$ for the set of @-labelled nodes, N_{var} for the set of variable nodes and N_{fv} for the subset of N_{var} constituted of free-variable nodes.

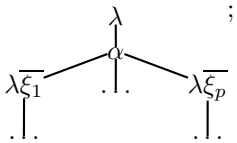
Let $\mathcal{T}$ denote the set of λ-terms. Each subtree of the computation tree $\tau(M)$ represents a subterm of $\lceil M \rceil$. We define the function $\kappa : N \to \mathcal{T}$ that maps a node $n \in N$ to the subterm of $\lceil M \rceil$ represented by the subtree of $\tau(M)$ rooted at n. In particular if r is the root of $\tau(M)$ then $\kappa(r) = \lceil M \rceil$.

Definition 3.1.4 (Node order)**.** The node-order function ord is defined on nodes as follows:

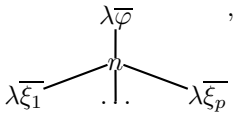
$$\mathsf{ord}(n) = \begin{cases} \mathsf{ord}(T), & \text{if } n \text{ is a variable or constant of type } T; \\ 1 + \max_{z \in \overline{\xi} \cup fv(M)} \mathsf{ord}(z), & \text{if } n \text{ is labelled } \lambda\overline{\xi} \text{ and is the root of } \tau(M); \\ 1 + \max_{z \in \overline{\xi}} \mathsf{ord}(z), & \text{if } n \text{ is labelled } \lambda\overline{\xi} \text{ and is not the root;} \\ 1, & \text{if } n \text{ is labelled } \lambda \text{ and is not the root;} \\ 0, & \text{if } n \text{ is labelled } @. \end{cases}$$

Some remarks:

- In a computation tree, nodes at even level are λ-nodes and nodes at odd level are either application nodes, variable or constant nodes;
- for any ground type variable or constant α, $\tau(\alpha) = \tau(\lambda.\alpha) =$ 

;
- for any higher-order variable or constant $\alpha : (A_1, \dots, A_p, o)$, the computation tree $\tau(\alpha)$ has the following form: 

;

- for any tree of the form 

, we have $\mathsf{ord}(\kappa(n)) = 0$.

3.1.2 Pointers and justified sequence of nodes

Definition 3.1.5 (Binder)**.** Let n be a variable node of the computation tree labelled x. We say that a node n is bound by the node m, and m is called the binder of n, if m is the closest node in the path from n to the root of the tree such that m is labelled $\lambda\overline{\xi}$ with $x \in \overline{\xi}$.

Definition 3.1.6 (Enabling)**.** The enabling relation $\vdash$ is defined on the set of nodes of the computation tree. We write $m \vdash n$ and we say that m enables n if and only if

- n is a bound variable node and m is the binder of n,
- or n is a free variable node and r is the root of the computation tree,
- or n is a λ-node and m is the parent node of n.

We call *input-variable* a variable that is hereditarily justified by the root of the computation tree. Free variables and variables bound by the root are example of input-variables.

Definition 3.1.7 (Justified sequence of nodes)**.** A *justified sequence of nodes* is a sequence of nodes of the computation tree $\tau(M)$ with pointers attached to the nodes. A node n in the sequence that is either a variable node or a lambda-node different from the root of the computation tree has a pointer to a node m occurring before n in the sequence such that $m \vdash n$. If n points to m then we say that m *justifies* n and we represent the pointer in the sequence as follows:

Note that justified sequences are also defined for open terms: occurrences of nodes in N_{fv} must point to an occurrence of the root of the computation tree.

A pointer is sometime labeled with an index i: if m is a λ-node then it indicates that n is labelled with the ith variable abstracted in m; otherwise it indicates that n is the ith child of m. A pointer in a justified sequence of nodes has therefore one of the following forms:

where r denotes the root of $\tau(M)$, $z \in N_{fv}$, $\xi_1, \ldots \xi_n$ are bound variables, $\alpha \in N_\Sigma \cup N_{var}$, $i \in 1..n$, j ranges from 0 to the number of children nodes of @ minus 1 and $k \in 1..arity(\alpha)$.

The following numbering conventions are used:

- the first child of a @-node is numbered 0,
- the first child of a variable or constant node is numbered 1,
- variables in $\overline{\xi}$ are numbered from 1 onward ($\overline{\xi} = \xi_1, \ldots \xi_n$).

We use the notation $n.i$ to denote the ith child of node n.

We write $s = t$ to denote that the justified sequences t and s have same nodes *and* pointers. Justified sequence of nodes can be ordered using the prefix ordering: $t \sqsubseteq t'$ if and only if $t = t'$ or the sequence of nodes t is a finite prefix of t' (and the pointers of t are the same as the pointers of the corresponding prefix of t'). Note that with this definition, infinite justified sequences can also be compared. This ordering gives rise to a complete partial order.

We say that a node n_0 of a justified sequence is hereditarily justified by n_p if there are nodes $n_1, n_2, \ldots n_{p-1}$ in the sequence such that for all $i \in 0..p-1$, n_i points to n_{i+1}.

If N is a set of nodes and s a justified sequence of nodes then we write $s \restriction N$ to denote the subsequence of s obtained by keeping only the nodes that are hereditarily justified by nodes in N. This subsequence is also a justified sequence of nodes. If n denotes a node of $\tau(M)$ we abbreviate $s \restriction \{n\}$ into $s \restriction n$.

Lemma 3.1.8. *For any set of node N, the filtering function $\_ \restriction N$ defined on the cpo of justified sequences ordered by the prefix ordering is continuous.*

Proof. Clearly $\_ \restriction N$ is monotonous. Suppose that $(t_i)_{i\in\omega}$ is a chain of justified sequence of nodes. Let u be a finite prefix of $(\bigvee t_i) \restriction r$. Then $u = s \restriction r$ for some finite prefix s of $\bigvee t_i$. Since s is finite we must have $s \sqsubseteq t_j$ for some $j \in \omega$. Therefore $u \sqsubseteq t_j \restriction r \sqsubseteq \bigvee(t_j \restriction r)$. This is valid for any finite prefix u therefore $(\bigvee t_i) \restriction r \sqsubseteq \bigvee(t_j \restriction r)$. □

Definition 3.1.9 (P-view of justified sequence of nodes)**.** The P-view of a justified sequence of nodes t of $\tau(M)$, written $\ulcorner t \urcorner$, is defined as follows:

$$\begin{array}{rcl} \ulcorner \epsilon \urcorner & = & \epsilon \\ \ulcorner s \cdot n \urcorner & = & \ulcorner s \urcorner \cdot n \\ \ulcorner s \cdot m \cdot \ldots \cdot \lambda\overline{\xi} \urcorner & = & \ulcorner s \urcorner \cdot m \cdot \lambda\overline{\xi} \\ \ulcorner s \cdot r \urcorner & = & r \end{array}$$

where r is the root of the tree $\tau(M)$ and n ranges over non-lambda nodes (i.e. $N_\Sigma \cup N_@ \cup N_{var}$).

In the second clause, the pointer associated to n is preserved from the left-hand side to the right-hand side i.e. 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 this occurrence of the node in $\ulcorner s \urcorner$.

Similarly, in the third clause the pointer associated to m is preserved.

We also define O-view, the dual notion of P-view:

Definition 3.1.10 (O-view of justified sequence of nodes)**.** The O-view of a justified sequence of nodes t of $\tau(M)$, written $\llcorner t \lrcorner$, is defined as follows:

$$\begin{array}{rcl} \llcorner \epsilon \lrcorner & = & \epsilon \\ \llcorner s \cdot \lambda\overline{\xi} \lrcorner & = & \llcorner s \lrcorner \cdot \lambda\overline{\xi} \\ \llcorner s \cdot m \cdot \ldots \cdot x \lrcorner & = & \llcorner s \lrcorner \cdot m \cdot x \\ \llcorner s \cdot n \lrcorner & = & n \end{array}$$

where x ranges over variable nodes and n ranges over non-lambda nodes without pointer (i.e. $N_@ \cup N_\Sigma$).

The pointer associated to $\lambda\overline{\xi}$ in the second equality and the pointer associated to m in the third equality are preserved from the left-hand side to the right-hand side of the equalities.

Definition 3.1.11 (Alternation and Visibility)**.**
A justified sequence of nodes s satisfies:

- *Alternation* if for any two consecutive nodes in s, one is a λ-node and the other is not;
- *P-visibility* if every variable node in s points to a node occurring in the P-view a that point;
- *O-visibility* if every lambda node in s points to a node occurring in the O-view a that point.

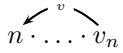
Property 3.1.12. The P-view (resp. O-view) of a justified sequence verifying P-visibility (resp. O-visibility) is a well-formed justified sequence verifying P-visibility (resp. P-visibility).

This is proved by an easy induction.

3.1.3 Adding value-leaves to the computation tree

We now add leaves to the computation tree that has been defined in the previous section. These leaves, called *value-leaves*, are attached to the nodes of the computation tree. Each value-leaf corresponds to a possible value of the base type o. We write $\mathcal{D}$ to denote the set of values of the base type o. The values leaves are added as follows: every node $n \in \tau(M)$ has a child leaf denoted by v_n for each possible value $v \in \mathcal{D}$.

Everything that we have defined for computation tree can be lifted to this new version of computation tree. The node order of a value-leaf is defined to be 0. The enabling relation $\vdash$ is extended so that every leaf is enabled by its parent node. The definition of justified sequence does not change. When representing a link in a justified sequence going from a value-leaf v_n to a node n, we label the link with v:



For the definition of P-view, O-view and visibility, value-leaves are treated as λ-nodes if they are at odd level in the computation tree and as variable nodes if there at a even level.

From now the term "computation tree" refers to this extended definition.

Let n be a node of a justified sequence of nodes. If there is an occurrence of a value-leaf v_n in the sequence that points to n we say that n is *matched* by v_n. If there is no value-leaf in the sequence that points to n we say that n is an *unmatched node*. The last unmatched node is called the *pending node*. A justified sequence of nodes is *well-bracketed* if each value-leaf in the traversal points to 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.

3.1.4 Traversal of the computation tree

We first define traversals for computation tree of simply-typed λ-terms with no interpreted constants. We will then we show how to extend the definition to the general setting of λ-calculus augmented with interpreted constants.

Traversals for simply-typed λ-terms

Intuitively, 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 3.1.13 (Traversals for pure simply-typed λ-terms)**.** In the simply-typed λ-calculus with no constants, a traversal over a computation tree $\tau(M)$ is a justified sequence of nodes defined by induction on the rules given below. A *maximal-traversal* is a traversal that cannot be extended by any rule. If T denotes a computation tree then we write $\mathcal{T}rav(T)$ to denote the set of traversals of T. We also use the abbreviation $\mathcal{T}rav(M)$ for $\mathcal{T}rav(\tau(M))$.

Initialization rules

- (ϵ) The empty sequence of node ϵ is a traversal.
- (Root) The length 1 sequence r, where r is denotes the root of $\tau(M)$, is a traversal.

Structural rules

- (Lam) Suppose that $t \cdot \lambda\overline{\xi}$ is a traversal and n is the only child node of $\lambda\overline{\xi}$ in the computation tree then
$$t \cdot \lambda\overline{\xi} \cdot n$$
is also a traversal where n points to the (only) occurrence of its enabler in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$. In particular, if n is a free variable node then n points to the first node of t.
- (App) If $t \cdot @$ is a traversal then so is


i.e. the next visited node is the 0th child node of @ : the node corresponding to the operator of the application.

Input-variable rules

- (InputVar^0) If $t = t_1 \cdot x \cdot t_2$ is a traversal where x is the pending node in t (i.e. $?(t_2) = \epsilon$) and x is a ground-type input-variable then for any $v \in \mathcal{D}$ the following is a traversal

$$t_1 \cdot x \overset{v}{\cdot} t_2 \cdot v_x$$

- (InputVar$^{\geq 1}$) If $t = t_1 \cdot x \cdot t_2$ is a traversal where x is the pending node in t (i.e. $?(t_2) = \epsilon$) and x is a higher-order input-variable then the following is a traversal:

$$t_1 \cdot x \overset{i}{\cdot} t_2 \cdot n \qquad \text{for } 1 \leq i \leq arity(x).$$

  Moreover for any $v \in \mathcal{D}$ the sequence $t_1 \cdot x \overset{v}{\cdot} t_2 \cdot v_x$ is also a traversal.

Copy-cat rules

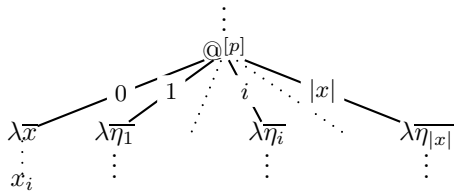
- (CCAnswer-@) If $t \cdot @ \overset{0}{\cdot} \lambda\overline{z} \overset{v}{\cdot} \ldots \cdot v_{\lambda\overline{z}}$ is a traversal then so is: $t \cdot @ \overset{0}{\cdot} \lambda\overline{z} \overset{v}{\cdot} \ldots \cdot v_{\lambda\overline{z}} \overset{v}{\cdot} v_{@}$.
- (CCAnswer-λ) If $t \cdot \lambda\overline{\xi} \overset{v}{\cdot} x \cdot \ldots \cdot v_x$ is a traversal then so is: $t \cdot \lambda\overline{\xi} \overset{v}{\cdot} x \cdot \ldots \cdot v_x \overset{v}{\cdot} v_{\lambda\overline{\xi}}$.
- (CCAnswer-var) If $t \cdot y \cdot \lambda\overline{\xi} \overset{v}{\cdot} \ldots \cdot v_{\lambda\overline{\xi}}$ is a traversal, where y is a non input-variable node, then the following is also a traversal:

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

- (Var) If $t \cdot x_i$ is a traversal where x_i is not an input-variable, then the rule (Var) permits to visit the node corresponding to the subterm that would be substituted for x_i if all the β-redexes occurring in M were reduced.

  The binding node $\lambda\overline{x}$ necessarily occur previously in the traversal. Since x is not hereditarily justified by the root, $\lambda\overline{x}$ is not the root of the tree and therefore its justifier p - which is also its parent node - occurs immediately before itself it in the traversal. We do a case analysis on p:

  – Suppose p is an @-node then $\lambda\overline{x}$ is necessarily the first child node of p and p has exactly $|\overline{x}| + 1$ children nodes:



  In that case, the next step of the traversal is a jump to $\lambda\overline{\eta_i}$ – the ith child of @ – which corresponds to the subterm that would be substituted for x_i if the β-reduction was performed:

$$t' \cdot @^{[p]} \cdot \lambda\overline{x} \overset{i}{\cdot} \ldots \cdot x_i \overset{i}{\cdot} \lambda\overline{\eta_i} \cdot \ldots \in \mathcal{T}rav(M)$$

  – Suppose p is variable node y, then necessarily the node $\lambda\overline{x}$ has been added to the traversal $t_{\leq y}$ using the (Var) rule (this is proved in proposition 3.1.17(i)). Therefore y is substituted by the term $\kappa(\lambda\overline{x})$ during the evaluation of the term and we have $\mathsf{ord}(y) = \mathsf{ord}(\lambda\overline{x})$.

  Consequently, during reduction, the variable x_i is substituted by the subterm represented by $\lambda\overline{\eta_i}$ – the ith child node of y. Hence the following justified sequence is also a traversal:

$$t' \cdot y^{[n]} \cdot \lambda\overline{x} \overset{i}{\cdot} \ldots \cdot x_i \overset{i}{\cdot} \lambda\overline{\eta_i} \cdot \ldots$$

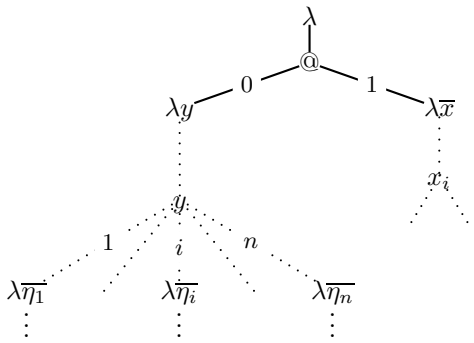
Note that a traversal always starts with the root of the tree.

Remark 3.1.14. Our notions of computation tree and traversal differ slightly from Ong [2006b].

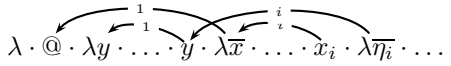
Firstly, our computation tree do not have nodes labelled with (uninterpreted) first-order constants. On the other hand, there are nodes which are labelled by free variables of any order. Since uninterpreted constants can be regarded as free variables, we do not lose any expressivity. The traversal rules (InputVar^0) and (InputVar$^{\geq 1}$) provide a more general version of the (Sig) rule of Ong [2006b].

Secondly we have introduced copy-cat rules that permit to visit the value-leaves of the computation tree. The presence of value-leaves is necessary to model free variables as well as the interpreted constants present in extensions of the λ-calculus such as PCF or IA.

Example 3.1.15. Consider the following computation tree:



An example of traversal of this tree is:



Traversals for interpreted constants

Definition 3.1.16 (Well-behaved traversal rule)**.** A traversal rule is *well-behaved* if it can be stated under the following form:

$$\frac{t = t_1 \cdot n \cdot t_2 \in \mathcal{T}rav \quad ?(t_2) = \epsilon \quad P(t)}{t' = t_1 \cdot n \cdot t_2 \cdot m \in \mathcal{T}rav}\ m \in S(t)$$

such that:

1. n is a variable or a constant node;
2. P expresses some condition on t;
3. $S(t)$ is some subset of $E(n)$, the set of children λ-nodes and value-leaves of n. If $S(t)$ has more than one element then the rule is non-deterministic.

Note that if t is well-bracketed then t' is also well-bracketed and if $?(t)$ satisfies alternation (resp. visibility) then so does $?(t')$.

The rules (InputVar^0) and (InputVar$^{\geq 1}$) are two examples of non-deterministic well-behaved traversal rules for which $S(t)$ is exactly the set of all children-nodes and value-leaves of n: $S(t) = \{n.i \mid i \in 1..arity(n)\} \cup \{v_n \mid v \in \mathcal{D}\}$.

In the presence of higher-order interpreted constants, additional rules must be specified to indicate how the constant nodes should be traversed in the computation tree. These rules are specific to the language that is being studied. In the last section of this chapter we will define such traversals for the interpreted constants of PCF and IA.

From now on, we consider a simply-typed λ-calculus language extended with higher-order interpreted constants for which some constant traversal rules have been defined and we take the following condition as a prerequisite:

Condition (WB) : the constant traversal rules are well-behaved.

Some properties of traversals

Proposition 3.1.17. *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 a path in the computation tree going from the root to the last node in $?(t)$.

This is the counterpart of proposition 6 from Ong [2006a] which is proved by induction on the traversal rules. This proof can be easily adapted to take into account the constant rules (using the assumption that constants rules are well-behaved) and the presence of value-leaves in the traversal.

Proof. We just give a partial proof of (i). We prove that in the second case of the (Var) rule, where p is a variable node y, the node $\lambda\overline{x}$ has necessarily been added to the traversal $t_{\leq y}$ using the (Var) rule.

Suppose that an (InputVar) rule is used to produce $t_{<y} \cdot y \cdot \lambda\overline{x}$, then y is an input-variable node and also the parent node of $\lambda\overline{x}$. But x is not an input-variable therefore it cannot be the child node of an input-variable. Hence (Var) is the only rule which can be used to produce $t_{<y} \cdot y \cdot \lambda\overline{x}$. □

Definition 3.1.18 (Traversal reduction)**.** Let r be the root of the computation tree. We say that the justified sequence of nodes s *is a reduction of the traversal* t just when $s = t \restriction r$.

Since @-nodes and Σ-constants do not have pointer, the reduction of traversal contains only nodes in $N_\lambda \cup N_{var}$.

Lemma 3.1.19. *Let M be a term in β-normal form and t be a traversal of $\tau(M)$. If $?(t) = u_1 \cdot m \overset{\frown}{\cdot\, u_2 \cdot} n$ where m is a not a λ-node then $u_2 = \epsilon$.*

Proof. By induction on the traversal rules. The only relevant rules are (Var), (CCAnswer-var), (InputVar^0), ($\text{InputVar}^{\geq 1}$) and the constant rules. Since the term is in β-normal form, there is no @-node in $\tau(M)$ and therefore (Var) cannot be used. For the rules (CCAnswer-var), (InputVar^0) and ($\text{InputVar}^{\geq 1}$) we just use the well-bracketedness of traversals. For the constant rules, the result is a consequence of condition (WB) stating that constant rules are the well-behaved. □

Lemma 3.1.20 (View of a traversal reduction)**.** *Let M be a term in β-normal form, r be the root of $\tau(M)$ and t be a traversal of $\tau(M)$. We have*

(i) $\ulcorner ?(t) \restriction r \urcorner = \ulcorner ?(t) \urcorner \restriction r$;

(ii) if the last node in t is hereditarily justified by r then $\llcorner ?(t) \restriction r \lrcorner = \llcorner ?(t) \lrcorner$.

Proof. (i) By induction. Base case: it is trivially true for the empty traversal $t = \epsilon$. Step case: consider a traversal t and suppose that the property (i) is verified for all traversal smaller than t. There are three cases:

- Suppose $?(t) = t' \cdot r$ then:

$$\begin{aligned}
\ulcorner ?(t) \urcorner \restriction r &= \ulcorner t' \cdot r \urcorner \restriction r && \text{(definition of } ?(t)) \\
&= r \restriction r && \text{(def. P-view)} \\
&= r && \text{(def. operator } \restriction) \\
&= \ulcorner (t' \restriction r) \cdot r \urcorner && \text{(def. P-view)} \\
&= \ulcorner (t' \cdot r) \restriction r \urcorner && \text{(def. operator } \restriction) \\
&= \ulcorner ?(t) \restriction r \urcorner && \text{(definition of } ?(t))
\end{aligned}$$

- Suppose $?(t) = t' \cdot n$ where n is a non-lambda move. We have:

$$\ulcorner ?(t) \urcorner = \ulcorner t' \cdot n \urcorner = \ulcorner t' \urcorner \cdot n \tag{3.1}$$

  - If n is not hereditarily justified by r then:

$$\begin{aligned}
\ulcorner ?(t) \urcorner \restriction r &= (\ulcorner t' \urcorner \cdot n) \restriction r && \text{(equation 3.1)} \\
&= \ulcorner t' \urcorner \restriction r && (n \text{ is not hereditarily justified by } r) \\
&= \ulcorner t' \restriction r \urcorner && \text{(induction hypothesis)} \\
&= \ulcorner (t' \cdot n) \restriction r \urcorner && (n \text{ is not hereditarily justified by } r) \\
&= \ulcorner ?(t) \restriction r \urcorner && \text{(definition of } ?(t))
\end{aligned}$$

  - If n is hereditarily justified by r then:

$$\begin{aligned}
\ulcorner ?(t) \urcorner \restriction r &= (\ulcorner t' \urcorner \cdot n) \restriction r && \text{(equation 3.1)} \\
&= (\ulcorner t' \urcorner \restriction r) \cdot n && (n \text{ is hereditarily justified by } r) \\
&= \ulcorner t' \restriction r \urcorner \cdot n && \text{(induction hypothesis)} \\
&= \ulcorner (t' \restriction r) \cdot n \urcorner && \text{(P-view computation)} \\
&= \ulcorner (t' \cdot n) \restriction r \urcorner && (n \text{ is hereditarily justified by } r) \\
&= \ulcorner ?(t) \restriction r \urcorner && \text{(definition of } ?(t))
\end{aligned}$$

- Suppose that $?(t) = t' \cdot m \overset{\curvearrowleft}{\cdot} u \cdot n$ where n is a λ-node then by lemma 3.1.19 we have $u = \epsilon$ and therefore:

$$\begin{aligned}
\ulcorner ?(t) \urcorner \restriction r &= \ulcorner t' \cdot m \overset{\curvearrowleft}{\cdot} n \urcorner \restriction r && (u = \epsilon) \\
&= (\ulcorner t' \urcorner \cdot m \overset{\curvearrowleft}{\cdot} n) \restriction r && \text{(P-view computation)} \\
&= \ulcorner t' \urcorner \restriction r && (m, n \text{ are not hereditarily justified by } r) \\
&= \ulcorner t' \restriction r \urcorner && \text{(induction hypothesis)} \\
&= \ulcorner (t' \cdot m \overset{\curvearrowleft}{\cdot} n) \restriction r \urcorner && \text{(def. operator } \restriction \text{ and } m, \lambda \text{ are not her. just. by } r) \\
&= \ulcorner ?(t) \restriction r \urcorner && \text{(def. of } ?(t))
\end{aligned}$$

(ii) By a straightforward induction similar to (i). □

Lemma 3.1.21 (Traversal of β-normal terms)**.** *Let M be a β-normal term, r be the root of the tree $\tau(M)$ and t be a traversal of $\tau(M)$. For any node n occurring in t:*

r does not hereditarily justify n $\iff$ n is hereditarily justified by some node in N_Σ.

Proof. In a computation tree, the only nodes that do not have justification pointer are: the root r, @-nodes and Σ-constant nodes. But since M is in β-normal form, there is no @-node in the computation tree. Hence nodes are either hereditarily justified by r or hereditarily justified by a node in N_Σ. Moreover r is not in N_Σ therefore the "or" is exclusive : a node cannot be hereditarily justified at the same time by r and by some node in N_Σ.

□

3.2 Game semantics of simply-typed λ-calculus with Σ-constants

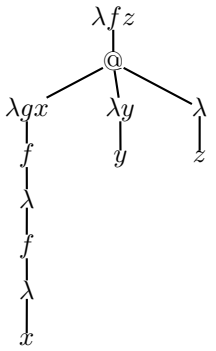
We are working in the general setting of an applied simply-typed λ-calculus with a given set of higher-order constants Σ. The operational semantics of these constants is given by certain reduction rules. We assume that a fully abstract model of the calculus is provided by mean of a category of well-bracketed games. For instance, if Σ is the set of PCF constants then we work in the category $\mathcal{C}_b$ of well-bracketed defined in section 1.3.3 of the first chapter.

We will use the alternative representation of strategy defined in remark 1.2.9: a strategy is given by a prefix-closed set instead of an "even length prefix"-closed set. In practice this means that we replace the set of plays σ by $\sigma \cup \mathsf{dom}(\sigma)$. This permits to avoid considerations on the parity of the length of traversals when we show the correspondence between traversals and game semantics. We write $[\![\Gamma \vdash M : A]\!]$ for the strategy denoting the simply-typed term $\Gamma \vdash M : A$ and $\mathsf{Pref}(S)$ to denote the prefix-closure of the set S.

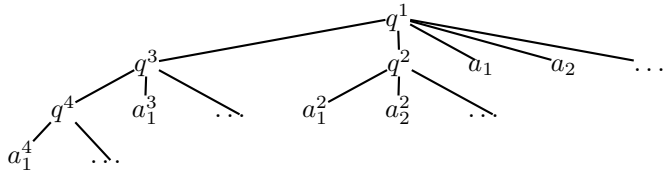
3.2.1 Relationship between computation trees and arenas

Example

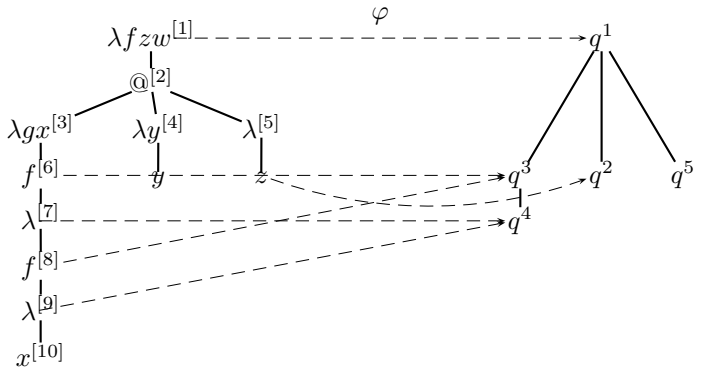
Consider the following term $M \equiv \lambda fz.(\lambda gx.f(fx))(\lambda y.y)z$ of type $(o \to o) \to o \to o$. Its η-long normal form is $\lambda fz.(\lambda gx.f(fx))(\lambda y.y)(\lambda.z)$. The computation tree is:



The arena for the type $(o \to o) \to o \to o$ is:



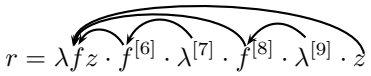
The figure below represents the computation tree (left) and the arena (right). The dashed line defines a partial function φ from the set of nodes in the computation tree to the set of moves. For simplicity, we now omit answers moves when representing arenas.



Consider the justified sequence of moves $s \in [\![M]\!]$:

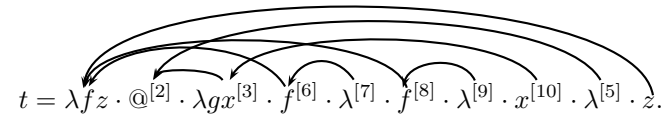
$$s = q^1\ q^3\ q^4\ q^3\ q^4\ q^2 \in [\![M]\!]$$

There is a corresponding justified sequence of nodes in the computation tree:



such that $s_i = \varphi(r_i)$ for all $i < |s|$.

The sequence r is in fact the reduction of the following traversal:



By representing side-by-side the computation tree and the type arena of a term in η-normal form we have observed that some nodes of the computation tree can be mapped to question moves of the arena. In the next section, we show how to define this mapping in a systematic manner.

Formal definition

Let us establish precisely the relationship between arenas of the game semantics and the computation trees. Let $\Gamma \vdash M : A$ be a term in η-long normal form. The computation tree $\tau(M)$ is represented by a pair (V, E) where V is the set of vertices of the trees and E is the edges relation. $V = N \cup L$ where N is the set of nodes and L is the set of value-leaves.

The relation $E \subseteq V \times V$ gives the parent-child relation on the vertices of the tree. $E = E_n \cup E_l$ where $E_n \subseteq N \times N$ gives the node-node parent relation and $E_l \subseteq N \times L$ gives the node-leaf parent relation. We write $L_\$$ for $E_l(N_\$)$ and $V_\$$ for $N_\$ \cup L_\$$ where $\$$ ranges over $\{@, var, \Sigma, fv\}$.

Let $\mathcal{D}$ be the set of values of the base type o. If n is a node in N then the value-leaves in $E_l(n)$ attached to the node n are written v_n where v ranges in $\mathcal{D}$. Similarly, if q is a question in $[\![A]\!]$ then the answer moves enabled by q are written v_q where v ranges in $\mathcal{D}$.

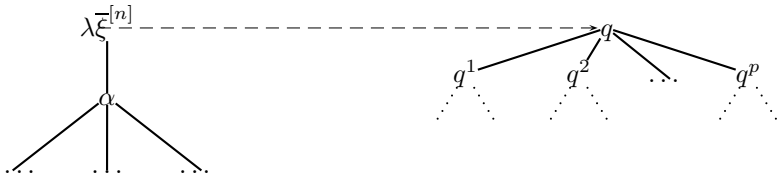
If A is an arena and q is a move in A then we write A_q to denote the subarena of A rooted at q.

Definition 3.2.1 (Relation between moves of the arena and nodes of the computation tree)**.** We consider the computation tree of a simply-typed-term. For any arena A, we define a function $f_A(n, q)$ taking two parameters: a node n of the computation tree and a question move q of the arena A such that q and n have the same type. $f_A(n, q)$ returns a partial function from V to A. It is defined as follows:

case 1 If n is an order 0 λ-node (i.e. labelled λ) or a ground type variable node then

$$f_A(n, q) = \{n \mapsto q\} \quad \cup \quad \{v_n \mapsto v_q \mid v \in \mathcal{D}\}$$

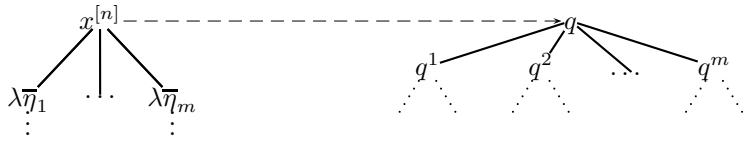
case 2 If n is a λ-node labelled $\lambda\overline{\xi} = \lambda\xi_1 \dots \xi_p$ with $p \geq 1$ then the computation tree and the arena A_q have the following form (value-leaves and answer moves are not represented for simplicity):



For each of the abstracted variable ξ_i there exists a corresponding question move q^i of the same order in the arena. $f_A(n, q)$ maps each free occurrence of a variable ξ_i to the corresponding move q^i:

$$f_A(n, q) = \{n \mapsto q\} \quad \cup \quad \{v_n \mapsto v_q \mid v \in \mathcal{D}\} \quad \cup \quad \bigcup_{\substack{m \in N | n \vdash m \\ m \text{ labelled } \xi_i}} f_A(m, q^i)$$

case 3 If n is a variable node labelled with a higher-order variable x of type $(A_1|\ldots|A_m|o)$ then the computation tree and the arena A_q have the following form:



$f_A(n,q)$ maps each child node of n to the corresponding question move q^i of the same type in the arena A_q:

$$f_A(n,q) = \{n \mapsto q\} \quad \cup \quad \{v_n \mapsto v_q \mid v \in \mathcal{D}\} \quad \cup \quad \bigcup_{i=1..m} f_A(\lambda\overline{\eta}_i, q^i)$$

Note that $f_A(n,q)$ is only a partial function from V to A since it is defined only on nodes that are hereditarily justified by the root *and* not hereditarily justified by a free variable node. In other words, $f_A(n,q)$ is undefined on nodes that are hereditarily justified by $N_{fv} \cup N_@ \cup N_\Sigma$.

Suppose $\Gamma \vdash M : T$ is a simply-typed term and N denotes the set of nodes of the computation tree. We write $\mathcal{M}_M$ to denote the following disjoint union of arenas:

$$\mathcal{M}_M = [\![\Gamma \to T]\!] \quad \uplus \quad \biguplus_{n \in E_n(\!| N_@ \cup N_\Sigma |\!)} [\![type(\kappa(n))]\!].$$

Moves in $\mathcal{M}_M$ are implicitly tagged so it is possible to recover the arena in which they belong.

Definition 3.2.2 (Total mapping from nodes to moves)**.** Let $\Gamma \vdash M : T$ be a simply-typed term with $\Gamma = x_1 : X_1 \ldots x_p : X_p$. We write $q^1_{[\![\Gamma]\!]}, \ldots, q^p_{[\![\Gamma]\!]}$ to denote the initial question moves of the component Γ of the arena $[\![\Gamma \to T]\!]$ and q^0_A to denote the single initial question of any arena A (arenas involved in the game semantics of pure simply-typed λ-calculus have only one root). r denotes the root of the computation tree.

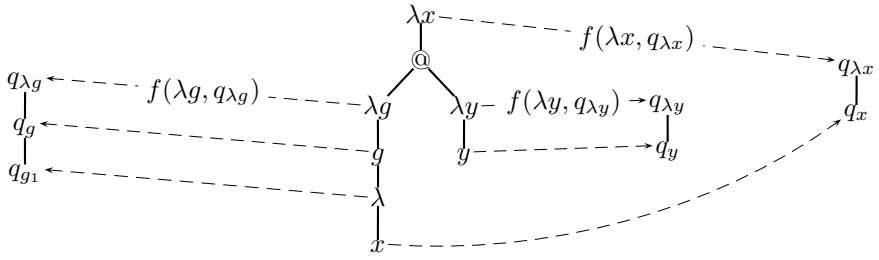
We define the total function $\varphi_M : V_\lambda \cup V_{var} \to \mathcal{M}_M$ as follows:

$$\begin{aligned}\varphi_M = f_{[\![\Gamma\to T]\!]}(r, q^0_{[\![\Gamma\to T]\!]}) \quad &\cup \bigcup_{n \in N_{fv} | n \text{ labelled } x_i} f_{[\![\Gamma\to T]\!]}(n, q^i_{[\![\Gamma]\!]}) \\ &\cup \bigcup_{n \in E_n(\!| N_@ \cup N_\Sigma |\!)} f_{[\![type(\kappa(n))]\!]}(n, q^0_{[\![type(\kappa(n))]\!]})\end{aligned}$$

When there is no ambiguity we just write φ instead of φ_M.

Nodes of $\tau(M)$ are either hereditarily justified by the root, by a @-node or by a Σ-node, therefore φ_M is totally defined on $V_\lambda \cup V_{var} = V \setminus (V_@ \cup V_\Sigma)$.

Example 3.2.3. Consider the term $\lambda x.(\lambda g.gx)(\lambda y.y)$ with $x, y : o$ and $g : (o,o)$. The diagram below represents the computation tree (middle), the arenas $[\![(o,o) \to o]\!]$ (left), $[\![o \to o]\!]$ (right), $[\![o \to o]\!]$ (rightmost) and the function $\varphi = f(\lambda x, q_{\lambda x}) \cup f(\lambda g, q_{\lambda g}) \cup f(\lambda y, q_{\lambda y})$ (dashed-lines).



The following properties are immediate consequences of the definition of the procedure f:

Property 3.2.4.

(i) 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;

(ii) φ maps nodes of a given order to moves of the same order.

Remark: we recall that in definition 3.1.4, the node-order is defined differently for the root λ-node and other λ-nodes. This convention was chosen to guarantee that property (ii) holds.

By extension, the function φ is also defined on justified sequences of nodes: if $t = t_0 t_1 \dots$ is a justified sequence of nodes in $V_\lambda \cup V_{var}$ then $\varphi(t)$ is defined to be the following sequence of moves:

$$\varphi(t) = \varphi(t_0)\ \varphi(t_1)\ \varphi(t_2) \dots$$

where the pointers of $\varphi(t)$ are defined to be exactly those of t. This definition implies that $\varphi : (V_\lambda \cup V_{var})^* \to \mathcal{M}^*$ regarded as a function from pointer-less sequences of nodes to pointer-less sequences of moves is a monoid homomorphism.

Property 3.2.5. Let t be a justified sequence of nodes. The following properties hold:

(i) $\varphi(t)$ and t have the same pointers;

(ii) the P-view of $\varphi(t)$ and the P-view of t are computed identically: the set of indices of elements that must be removed from both sequences in order to obtain their P-view is the same;

(iii) the O-view of $\varphi(t)$ and the O-view of t are computed identically;

(iv) if t is a justified sequence of nodes in $V_\lambda \cup V_{var}$ then $?(\varphi(t)) = \varphi(?(t))$,

where $?(\varphi(t))$ denotes the set of unanswered questions in the justified sequence of moves $\varphi(t)$ and $?(t)$ denotes the set of unmatched nodes in the justified sequence of nodes t (see the definition in section 3.1.3).

Proof. (i): By definition of φ, t and $\varphi(t)$ have the same pointers;

(ii) and (iii): φ maps lambda nodes to O-question, non-lambda nodes to P-question, value-leaves of lambda nodes to P-answers and value-leaves of non-lambda to O-answers. Therefore since t and $\varphi(t)$ have the same pointers, the computations of the P-view (resp. O-view) of the sequence of moves and the P-view (resp. O-view) of the sequence of nodes follow the same steps;

(iv) is a consequence of (i).

□

3.2.2 Category of interaction games

In game semantics, strategy composition is achieved by performing a CSP-like "composition + hiding". It is possible to define an alternative semantics where the internal moves are not hidden when performing composition. This semantics is named *interaction* semantics in Dimovski et al. [2005] and *revealed semantics* in Greenland [2004].

In addition to the moves of the standard semantics, the interaction semantics contains certain internal moves of the computation. Consequently, the interaction semantics depends on the syntactical structure of the term and therefore cannot lead to a full abstraction result. However this semantics will prove to be useful to identify a correspondence between the game semantics of a term and the traversals of its computation tree.

We will be interested in the interaction semantics computed from the η-normal form of a term. However we do not want to keep all the internal moves. We will only keep the internal moves that are produced when composing two subterms of the computation tree that are joined by an @-node. This means that when computing the strategy of $yN_1 \dots N_p$ where y is a variable, we keep the internal moves of $N_1, \dots, N_p$, but we omit the internal moves produced by the copy-cat projection strategy denoting y.

Definition 3.2.6 (Type-tree)**.** We call *type decomposition tree* or *type-tree*, a tree whose leaves are labelled with linear simple types and nodes are labelled with symbol in $\{;, \times, \otimes, \dagger, \Lambda\}$.

Nodes labelled ;, $\times$ or $\otimes$ are binary nodes and nodes labelled $\dagger$ or Λ are unary nodes.

Every node or leaf of the tree has a linear type, this type is determined by the structure of the tree as follows:

- a leaf has the type of its label;
- a $\dagger$-node with the child node of type $!A \multimap B$ has type $!A \multimap !B$;
- a Λ-node with the child node of type $A \otimes B \multimap C$ has type $A \multimap (B \multimap C)$;
- a $\times$-node with two children nodes of type A and B has type $A \times B$;
- a $\otimes$-node with two children nodes of type A and B has type $A \otimes B$;
- a ;-node with two children nodes of type $A \multimap B$ and $B \multimap C$ has type $A \multimap C$.

For a type-tree to be well-defined, the type of the children nodes must be compatible with the meaning of the node, for instance the two children nodes of a ;-node must be of type $A \multimap B$ and $B \multimap C$.

We write $type(T)$ to denote the type represented by the root of the tree T. An we say that T is a *valid tree decomposition* of $type(T)$.

If T_1 and T_2 are type-tree we write $T_1 \times T_2$ to denote the tree obtained by attaching T_1 and T_2 to a $\times$-node. Similarly we use the notations $T_1 \otimes T_2$, $T_1; T_2$, $\Lambda(T_1)$ and $T_1^\dagger$.

Let T be a type-tree. Each leaf or node of type A in T can be mapped to the (standard) arena $[\![A]\!]$. By taking the image of T across this mapping we obtain a tree whose leaves and nodes are labelled by arenas. This tree, written $\langle\!\langle T \rangle\!\rangle$, is called the *interaction arena* of type T. We write $root(\langle\!\langle T \rangle\!\rangle)$ to denote the arena located at the root of the interaction arena $\langle\!\langle T \rangle\!\rangle$.

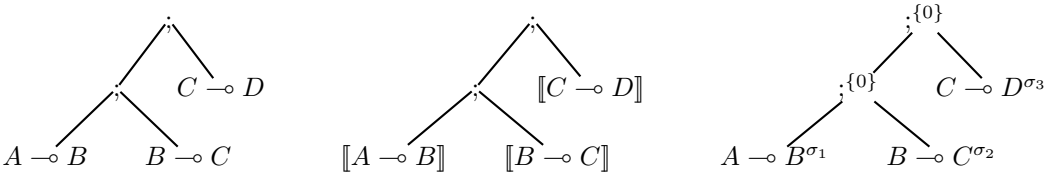
A *revealed strategy* Σ on the interaction arena $\langle\!\langle T \rangle\!\rangle$ is a composition of several standard strategies where certain internal moves are not hidden. Formally this can be defined as follows:

Definition 3.2.7 (Revealed strategy)**.** A revealed strategy Σ on a game $\langle\!\langle T \rangle\!\rangle$, written $\Sigma : \langle\!\langle T \rangle\!\rangle$, is a tree type T where

- each leaf $[\![A]\!]$ of $\langle\!\langle T \rangle\!\rangle$ is annotated with a (standard) strategy σ on the game $[\![A]\!]$;
- each ;-node is annotated with a set of indices $U \subseteq \mathbb{N}$.

A ;-node with children of type $A \multimap B$ and $B \multimap C$ is annotated with a set of indices U indicating which components of B should be uncovered when performing composition.

Example 3.2.8. The diagrams below represent a type-tree T (left) the corresponding interaction arena $\langle\!\langle T \rangle\!\rangle$ (middle) and an revealed strategy Σ (right):



A revealed strategy can also be written as an expression, for instance the strategy represented above is given by the expression $\Sigma = (\sigma_1;^{\{0\}} \sigma_2);^{\{0\}} \sigma_3$. We will use the abbreviation $\Sigma_1 \fatsemi^U \Sigma_2$ for $\Sigma_1^\dagger;^U \Sigma_2$.

Definition 3.2.9 (Composition of revealed strategies)**.** Suppose $\Sigma_1 : \langle\!\langle T_1 \rangle\!\rangle$ and $\Sigma_2 : \langle\!\langle T_2 \rangle\!\rangle$ are revealed strategies where $type(T_1) = A \multimap B$ and $type(T_2) = B \multimap C$ then the *interaction composition* of Σ_1 and Σ_2 written $\Sigma_1; \Sigma_2$ is the revealed strategy on $\langle\!\langle T_1; T_2 \rangle\!\rangle$ obtained by copying the annotation of the leaves and nodes from Σ_1 and Σ_2 to the corresponding leaves and nodes of the type-tree $T_1; T_2$ and by annotating the root node with $\emptyset$.

A play of the interaction semantics, called an *uncovered play*, is a play containing internal moves. The moves are implicitly tagged so that it is possible to retrieve in which component of which node or leaf-arenas the move belongs to. Note that a same move can belong to different node/leaf-arenas. The internal moves of an interaction play on the game $\langle\!\langle T \rangle\!\rangle$ are those which do not belong to the arena $root(\langle\!\langle T \rangle\!\rangle)$.

For any uncovered play s and any interaction arena $\langle\!\langle T \rangle\!\rangle$ we can define the filtering operator $s \upharpoonright \langle\!\langle T \rangle\!\rangle$ to be the sequence of moves obtained from s by keeping only the moves belonging to a node or leaf-arena of $\langle\!\langle T \rangle\!\rangle$.

Revealed strategies can alternatively be represented by mean of sets of uncovered plays instead of annotated type-trees. This set is defined inductively on the structure of the annotated type-tree Σ as follows:

- for a leaf $[\![A]\!]$ of Σ annotated by $\sigma : [\![A]\!]$, it is just the set of plays of the standard strategy σ;
- for a $\otimes$-node with two children strategies Σ_1 and Σ_2, it is the tensor product written $\Sigma_1 \otimes \Sigma_2$;
- for a $\times$-node, it is the pairing written $\langle \Sigma_1, \Sigma_2 \rangle$;
- for a $\dagger$-node with a child strategy Σ, it is the promotion written $\Sigma^\dagger$;
- for a Λ-node with a child strategy Σ, it is the same set of plays with the moves retagged appropriately;
- for a $;^U$-node, it is the "uncovered-composition" of $\Sigma_1 : \langle\!\langle T_1 \rangle\!\rangle$ and $\Sigma_2 : \langle\!\langle T_2 \rangle\!\rangle$ which is written $\Sigma_1;^U \Sigma_1$ and defined as follows: suppose that $type(T_1) = A \multimap B_0 \times \ldots \times B_l$ and $type(T_2) = B_0 \times \ldots \times B_l \multimap C$ then $\Sigma_1;^U \Sigma_1$ is the set of uncovered plays obtained by performing the usual composition while ignoring and copying the internal moves from arenas in $\langle\!\langle T_1 \rangle\!\rangle$ or $\langle\!\langle T_2 \rangle\!\rangle$ and preserving any internal move produced by the composition in some component B_k for $k \in U$. Formally:

$$\Sigma_1 \| \Sigma_2 = \{u \in int(\langle\!\langle T \rangle\!\rangle) \mid u \upharpoonright \langle\!\langle T_1 \rangle\!\rangle \in \Sigma_1 \text{ and } u \upharpoonright \langle\!\langle T_2 \rangle\!\rangle \in \Sigma_2\}$$

$$\Sigma_1;^{\{i_0,\ldots i_l\}} \Sigma_2 = \{u \upharpoonright A, B_{i_0}, \ldots, B_{i_l}, C \mid u \in \Sigma_1 \| \Sigma_2\}$$

  where $int(\langle\!\langle T \rangle\!\rangle)$ denotes the set of sequences of moves in (some arena of) $\langle\!\langle T \rangle\!\rangle$;

where the tensor product, pairing and promotion are defined similarly as in the standard game semantics.

We can now define the category $\mathcal{I}$ of interaction games:

Definition 3.2.10 (Category of interaction games)**.** The category of interaction games is denoted by $\mathcal{I}$. The objects of $\mathcal{I}$ are those of $\mathcal{C}$ i.e. the arenas $[\![A]\!]$ for some linear type A. The morphisms of the category are the revealed strategies: a morphism from A to B is an revealed strategy Σ on some interaction arena $\langle\!\langle T \rangle\!\rangle$ such that $root(\langle\!\langle T \rangle\!\rangle) = [\![!A \multimap B]\!]$.

The composition of two morphisms Σ_1 and Σ_2 is given by $\Sigma_1 \fatsemi \Sigma_2 = \Sigma_1^\dagger; \Sigma_2$ where ; denotes the revealed strategy composition. The identity on A is the revealed strategy given by the single annotated leaf $[\![!A \multimap A]\!]^{der_A}$.

It can be checked that this indeed defines a category. The constructions of the category $\mathcal{C}$ can be transposed to $\mathcal{I}$ making $\mathcal{I}$ a cartesian closed category.

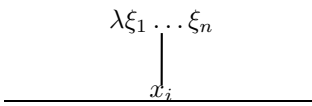
Definition 3.2.11 (Valid strategy)**.** Consider a term $\Gamma \vdash M : A$ and an revealed strategy $\Sigma : \langle\!\langle T \rangle\!\rangle$. We say that Σ is a valid revealed strategy for M if $root(\langle\!\langle T \rangle\!\rangle) = [\![\Gamma \to A]\!]$ or equivalently if $type(T) = \Gamma \to A$.

Modeling the λ-calculus in $\mathcal{I}$

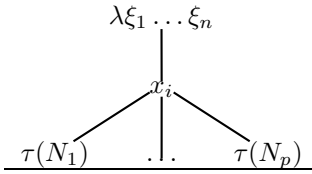
We would like to use the category $\mathcal{I}$ to model terms of the simply-typed lambda calculus. However there may be several valid type decomposition tree for a given term M and therefore several strategies denoting M. To fix this problem, we will base our definition on the computation tree of M. Since the computation tree is unique, the denotation of a term will be uniquely defined.

Definition 3.2.12 (Revealed denotation of a term)**.** Let $\Gamma \vdash M : A$ be a term with $\Gamma = x_1 : X_1, \ldots, x_k : X_k$. Let $\pi_i : [\![\Gamma \to X_i]\!]$ denote the ith projection copycat strategy and ev^p denote the evaluation strategy with p parameters.

The *revealed game denotation of* M or *revealed strategy of* M written $\langle\!\langle \Gamma \vdash M : A \rangle\!\rangle$ is the revealed strategy defined by structural induction on the computation tree $\tau(M)$ as follows:

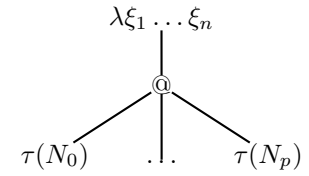


$\langle\!\langle M \rangle\!\rangle = \Lambda^n(\pi_i)$



where $\Gamma \vdash x_i : (A_1, \ldots, A_p, B)$ and $\Gamma \vdash N_j : A_j$ for $j \in 1..p$

$\langle\!\langle M \rangle\!\rangle = \Lambda^n(\langle \pi_i, \langle\!\langle \Gamma \vdash N_1 : A_1 \rangle\!\rangle, \ldots, \langle\!\langle \Gamma \vdash N_p : A_p \rangle\!\rangle \rangle \fatsemi^{1..p} ev^p)$

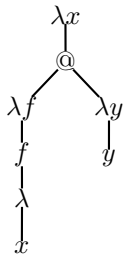


where $\Gamma \vdash N_0 : (A_1, \ldots, A_p, B)$ and $\Gamma \vdash N_j : A_j$ for $j \in 1..p$

$\langle\!\langle M \rangle\!\rangle = \Lambda^n(\langle \langle\!\langle \Gamma \vdash N_0 : A_0 \rangle\!\rangle, \ldots, \langle\!\langle \Gamma \vdash N_p : A_p \rangle\!\rangle \rangle \fatsemi^{0..p} ev^p)$

We write $\langle\!\langle \Gamma \to A \rangle\!\rangle_M$ to denote the interaction arena of the revealed strategy $\langle\!\langle \Gamma \vdash M : A \rangle\!\rangle$.

Example 3.2.13. Consider the term $\lambda x.(\lambda f.fx)(\lambda y.y)$. Its computation tree is:



and its revealed strategy is $\langle [\![x : X \vdash \lambda f.fx]\!], [\![x : X \vdash \lambda y.y]\!] \rangle \fatsemi^{\{0,1\}} ev_2$.

From interaction semantics to standard semantics and vice-versa

In the standard semantics, given two strategies $\sigma : A \to B$, $\tau : B \to C$ and a sequence $s \in \sigma \fatsemi \tau$, it is possible to (uniquely) recover the internal moves. The uncovered sequence is written $\mathbf{u}(s, \sigma, \tau)$. The algorithm to obtain this unique uncovering is given in part II of Hyland and Ong [2000].

Given a term M, we can completely uncover the internal moves of a sequence $s \in [\![M]\!]$ by performing the uncovering recursively at every @-node of the computation tree. This operation is called *full-uncovering with respect to* M.

Conversely, the standard semantics can be recovered from the interaction semantics by filtering the moves, keeping only those played in the root arena:

$$[\![\Gamma \vdash M : A]\!] = \langle\!\langle \Gamma \vdash M : A \rangle\!\rangle \upharpoonright [\![\Gamma \to T]\!] \tag{3.2}$$

Full abstraction

Let $\mathcal{I}'$ denote lluf sub-category of $\mathcal{I}$ consisting only of strategies Σ with a single annotated leaf and no nodes. We have the following lemma:

Lemma 3.2.14 ($\mathcal{I}'$ is isomorphic to $\mathcal{C}$)**.** $\mathcal{I}' \cong \mathcal{C}$

Proof. We define the functor $F : \mathcal{I}' \to \mathcal{C}$ by $F(A) = A$ for any object $A \in \mathcal{I}'$ and for $\Sigma \in \mathcal{I}'(A, B)$, $F(\Sigma)$ is defined to be the annotation σ of the only leaf in Σ. The functor $G : \mathcal{C} \to \mathcal{I}'$ is defined by $G(A) = A$ for any object $A \in \mathcal{C}$ and for $\sigma \in \mathcal{C}(A, B)$, $G(\sigma)$ is the tree formed with the single annotated leaf $[\![A]\!]^\sigma$. Then $F; G = id_{\mathcal{I}'}$ and $G; F = id_{\mathcal{C}}$. □

Consequently the lluf sub-category $\mathcal{I}'$ is fully abstract for the simply-typed lambda calculus. Note that this is a major difference with $\mathcal{I}$ which is not fully-abstract since there may be several maps denoting a given term.

3.2.3 The correspondence theorem for the pure simply-typed λ-calculus

In this section, we establish a connection between the interaction semantics of a simply-typed term without constants ($\Sigma = \emptyset$) and the traversals of its computation tree.

Removing @-nodes from traversals

When defining computation trees, it was necessary to introduce application nodes (labelled @) in order to connect the operator and the operand of an application. The presence of @-nodes has also another advantage: it ensures that the lambda-nodes are all at even level in the computation tree. Consequently a traversal respects Alternation.

Application nodes are however redundant in the sense that they do not play any role in the computation of the term. In other words, the @-nodes occurring in traversals are superfluous. In fact it is necessary to filter them out if we want to establish the correspondence with the interaction game semantics.

Definition 3.2.15 (Filtering @-nodes in traversals)**.** Let t be a traversal of $\tau(M)$. We write $t - @$ for the sequence of nodes with pointers obtained by

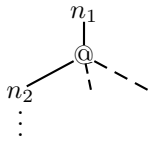
- removing from t all @-nodes and value-leaves of a @-node;
- replacing any link pointing to an @-node by a link pointing to the immediate predecessor of @ in t.

Suppose $u = t - @$ is a sequence of nodes obtained by applying the previously defined transformation on the traversal t, then t can be partially recovered from u by reinserting the @-nodes as follows. For each @-node @ in the computation tree with parent node denoted by p, we perform the following operations:

1. replace every occurrence of the pattern $p \cdot n$, where n is a λ-nodes, by $p \cdot @ \cdot n$;
2. replace any link in u starting from a λ-node and pointing to p by a link pointing to the inserted @-node;
3. if there is an occurrence in u of a value-leaf v_p pointing to p then insert a value-leaf $v_@$ immediately before v_p and make it points to the node immediately following p (which is also the @-node that we inserted in 1).

We write $u + @$ for this second transformation.

These transformations are well-defined because in a traversal, an @-node always occurs in-between two nodes n_1 and n_2 such that n_1 is the parent node of @ and n_2 is the first child node of @ in the computation tree:



Remark: $t - @$ is not a proper justified sequence since after removing a @-node, any λ-node justified by @ will become justified by the parent of @ which is also a λ-node.

The following lemma follows directly from the definition:

Lemma 3.2.16. *For any traversal t we have $(t-@)+@ \sqsubseteq t$ and if t does not end with an @-node then $(t - @) + @ = t$.*

Let M be a term and r be the root of $\tau(M)$. We introduce the following notations:

$$\begin{aligned} \mathcal{T}rav(M)^{-@} &= \{t - @ \mid t \in \mathcal{T}rav(M)\} \\ \mathcal{T}rav(M)^{\restriction r} &= \{t \restriction r \mid t \in \mathcal{T}rav(M)\}. \end{aligned}$$

Lemma 3.2.17. *Let M be a pure simply-typed term and r be the root of $\tau(M)$. If M is in β-normal form then $t = t \restriction r = t - @$ for any $t \in \mathcal{T}rav(M)$. Consequently,*

$$\mathcal{T}rav(M)^{-@} \cong \mathcal{T}rav(M) \cong \mathcal{T}rav(M)^{\restriction r}.$$

Proof. This is because the computation tree of a term in β-normal does not contain any @-node and therefore all the nodes are hereditarily justified by the root. □

Lemma 3.2.18 (Filtering lemma)**.** *Let $\Gamma \vdash M : T$ be a term and r be the root of $\tau(M)$. For any traversal t of the computation tree we have $\varphi(t - @) \restriction [\![\Gamma \to T]\!] = \varphi(t \restriction r)$. Consequently:*

$$\varphi(\mathcal{T}rav^{-@}(M)) \restriction [\![\Gamma \to T]\!] = \varphi(\mathcal{T}rav^{\restriction r}(M)).$$

Proof. From the definition of φ, the nodes of the computation tree that are mapped by φ to moves of the arena $[\![\Gamma \to T]\!]$ are exactly the nodes that are hereditarily justified by r. The result follows from the fact that @-nodes are not hereditarily justified by the root. □

The function φ regarded as a function from the set of vertices $V_\lambda \cup V_{var}$ of the computation tree to moves in arenas is not injective. For instance the two occurrences of x in the computation tree of the term $\lambda f x.fxx$ are mapped to the same question. However the function φ regarded as a function from sequences of nodes to sequences of moves is injective:

Lemma 3.2.19 (φ is injective)**.** *φ regarded as a function defined on the set of sequences of nodes is injective in the sense that for any two traversals t_1 and t_2:*

(i) if $\varphi(t_1 - @) = \varphi(t_2 - @)$ then $t_1 - @ = t_2 - @$;

(ii) if $\varphi(t_1 \restriction r) = \varphi(t_2 \restriction r)$ then $t_1 \restriction r = t_2 \restriction r$.

Proof. (i) The set of traversals of a computation tree verifies the following property:

$$t \cdot n_1, t \cdot n_2 \in \mathcal{T}rav \text{ where } n_1 \neq n_2 \text{ and } n_1, n_2 \text{ are not @-node implies } \varphi(n_1) \neq \varphi(n_2). \qquad (3.3)$$

Indeed, the only possible case where φ maps two different nodes to the same move is when n_1 and n_2 are two nodes labelled with the same variable x. Hence the two traversals $t \cdot n_1$ and $t \cdot n_2$ must have been formed using either rule (Lam) or (App). But these two rules are deterministic and their domain of definition is disjoint. This contradict the fact that $n_1 \neq n_2$.

Now suppose that $t_1 - @ \neq t_2 - @$ then necessarily $t_1 \neq t_2$. Therefore $t_1 = t' \cdot n_1 \cdot u_1$ and $t_2 = t' \cdot n_2 \cdot u_2$ for some sequences t', u_1, u_2 and some nodes $n_1 \neq n_2$. By property 3.3 we have $\varphi(n_1) \neq \varphi(n_2)$. If we regard sequences of nodes and moves as *pointer-less* sequences then we are allowed to write the following:

$$(t' \cdot n_1 \cdot u_1) - @ = (t' - @) \cdot n_1 \cdot (u_1 - @),$$

and since φ_M is a monoid homomorphism (provided that we ignore the justification pointers) we have:

$$\varphi(t_1 - @) = \varphi(t' - @) \cdot \varphi(n_1) \cdot \varphi(u_1) \neq \varphi(t' - @) \cdot \varphi(n_2) \cdot \varphi(u_2) = \varphi(t_2 - @).$$

(ii) Again, suppose that $t \restriction r \neq t' \restriction r$ then $t_1 = t'_1 \cdot n_1 \cdot u_1$ and $t_2 = t'_2 \cdot n_2 \cdot u_2$ for some sequences t'_1, t'_2, u_1, u_2 such that $t'_1 \restriction r = t'_2 \restriction r$ and some nodes $n_1 \neq n_2$ both hereditarily justified by the root. For the same reason as in (i), we must have $\varphi(n_1) \neq \varphi(n_2)$. Hence:

$$\varphi(t_1 \restriction r) = \varphi(t'_1 \restriction r) \cdot \varphi(n_1) \cdot \varphi(u_1 \restriction r) \neq \varphi(t'_1 \restriction r) \cdot \varphi(n_2) \cdot \varphi(u_2 \restriction r) = \varphi(t_2 \restriction r).$$

□

Corollary 3.2.20.

(i) φ defines a bijection from $\mathcal{T}rav(M)^{-@}$ to $\varphi(\mathcal{T}rav(M)^{-@})$;

(ii) φ defines a bijection from $\mathcal{T}rav(M)^{\restriction r}$ to $\varphi(\mathcal{T}rav(M)^{\restriction r})$.

The correspondence theorem

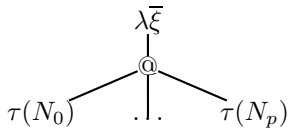
We are now going to state and prove the correspondence theorem for the pure simply-typed λ-calculus without constants ($\Sigma = \emptyset$). The result extends immediately to the simply-typed λ-calculus with *uninterpreted* constants by considering constants as being free variables. We use the cartesian closed category of games $\mathcal{C}$ (defined in section 1.3.3 of the first chapter) as a model of the simply-typed λ-calculus. We write $[\![\Gamma \vdash M : A]\!]$ for the strategy denoting the simply-typed term $\Gamma \vdash M : A$.

Proposition 3.2.21. *Let $\Gamma \vdash M : T$ be a term of the pure simply-typed λ-calculus and r be the root of $\tau(M)$. We have:*

(i) $\varphi_M(\mathcal{T}rav(M)^{-@}) = \langle\!\langle M \rangle\!\rangle$

(ii) $\varphi_M(\mathcal{T}rav(M)^{\restriction r}) = [\![M]\!]$.

Remark 3.2.22. The proof that follows is quite tedious but the idea is simple. Let us give the intuition. We start by reducing the problem to the case of closed terms only. Then the proof proceeds by induction on the structure of the computation tree. It is straightforward to prove the result for term that are abstraction of a single variable. Now consider an application M with the following computation tree $\tau(M)$:



A traversal of $\tau(M)$ proceeds as follows: it starts at the root $\lambda\overline{\xi}$ of the tree $\tau(M)$ (rule (Root)), it then passes the node @ (rule (Lam)). After this initialization part, it proceeds by traversing the term N_0 (rule (App)). At some point, while traversing N_0, some variable y_i bound by the root of N_0 is visited. The traversal of N_0 is interrupted and there is a jump (rule (Var)) to the root of $\tau(N_i)$. The process goes on by traversing $\tau(N_i)$. When traversing N_i, if the traversal encounters

a variable bound by the root of $\tau(N_i)$ then the traversal of N_i is interrupted and the traversal of N_0 resumes. This schema is repeated until the traversal of $\tau(N_0)$ is completed[1].

The traversal of M is therefore made of an initialization part followed by an interleaving of a traversal of N_0 and several traversals of N_i for $i = 1..p$. This schema is reminiscent of the way the evaluation copycat map ev works in game semantics.

The key idea is that every time the traversal pauses the traversal of a subterm and switches to another one, the jump is permitted by one of the four copycat rules (Var), (CCAnswer-@), (CCAnswer-λ) or (CCAnswer-var). We show by (a second) induction that these copycat rules defines exactly what the copycat strategy ev performs on sets of moves.

Let us fix some notation: we write $s \upharpoonright A, B$ for the sequence obtained from s by keeping only the moves that are in A or B and by removing any link pointing to a move that has been removed. If m is an initial move, we write $s \upharpoonright m$ to denote the thread of s initiated by m, i.e. the sequence obtained from s by keeping all the moves hereditarily justified by m. We also write $s \upharpoonright A, B, m$ where m is an initial move for the sequence obtained from $s \upharpoonright A, B$ by keeping all moves hereditarily justified by m.

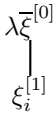
Proof. (i) Suppose $\Gamma = \xi_1 : X_1, \dots \xi_n : X_n$. Then we have:

$$\begin{aligned} \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle &= \Lambda^n(\langle\!\langle \emptyset \vdash \lambda\xi_1 \dots \xi_n.M : (X_1, \dots, X_n, T) \rangle\!\rangle) \\ &\simeq \langle\!\langle \emptyset \vdash \lambda\xi_1 \dots \xi_n.M : (X_1, \dots, X_n, T) \rangle\!\rangle. \end{aligned}$$

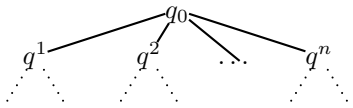
Similarly the computation tree $\tau(M)$ is isomorphic to $\tau(\lambda\xi_1 \dots \xi_n.M)$ (up to a renaming of the root of the computation tree) therefore $\mathcal{T}rav(M)$ is also isomorphic to $\mathcal{T}rav(\lambda\xi_1 \dots \xi_n.M)$. Hence we can make the assumption that M is a closed term. If we prove that the property is true for all closed terms of a given height then it will be automatically true for any open term of the same height.

Let us assume that M is already in η-long normal form. We proceed by induction on the height of the tree $\tau(M)$ and by case analysis on the structure of the computation tree:

- (abstraction of a variable): $M \equiv \lambda\overline{\xi}.x$. Since M is in η-long normal form, x must be of ground type and since M is closed we have $x = \xi_i \in \overline{\xi}$ for some i. Hence $\tau(M)$ has the following shape:



  The arena is of the following form (only question moves are represented):



  Let π_i denote the ith projection of the interaction game semantics. We have:

$$\begin{aligned} \langle\!\langle M \rangle\!\rangle &= \langle\!\langle \emptyset \vdash \lambda\overline{\xi}.\xi_i \rangle\!\rangle \\ &= \Lambda^n(\langle\!\langle \overline{\xi} \vdash \xi_i \rangle\!\rangle) \\ &= \Lambda^n(\pi_i) \\ &\cong \pi_i \\ &= \mathsf{Pref}(\{q_0 \cdot q^i \cdot v_{q^i} \cdot v_{q_0} \mid v \in \mathcal{D}\}). \end{aligned}$$

[1] Since we are considering simply-typed terms, the traversal does indeed terminate. However this will not be true anymore in the PCF case.

Since M is in β-normal we have $\mathcal{T}rav(M)^{-@} = \mathcal{T}rav(M)$. It is easy to see that the set of traversals of M is the set of prefix of the traversal $\lambda\overline{\xi} \cdot \xi_i \cdot v_{\xi_i} \cdot v_{\lambda\overline{\xi}}$:

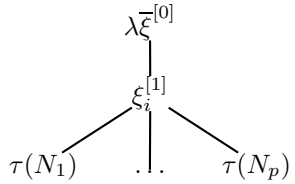
$$\mathcal{T}rav^{-@}(M) = \mathcal{T}rav(M) = \mathsf{Pref}(\lambda\overline{\xi} \cdot \xi_i \cdot v_{\xi_i} \cdot v_{\lambda\overline{\xi}})$$

The pointers of the traversal $\lambda\overline{\xi} \cdot \xi_i \cdot v_{\xi_i} \cdot v_{\lambda\overline{\xi}}$ are the same as the play $q_0 \cdot q^i \cdot v_{q^i} \cdot v_{q_0}$, therefore since $\varphi_M(\lambda\overline{\xi}) = q_0$ and $\varphi_M(\xi_i) = q^i$ we have:

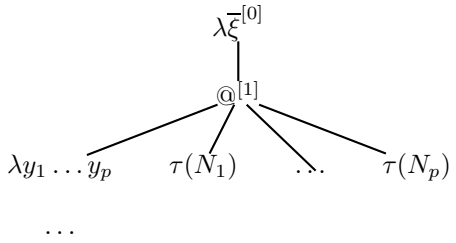
$$\varphi_M(\mathcal{T}rav^{-@}(M)) = \langle\!\langle M \rangle\!\rangle.$$

- (abstraction of an application): we have $M = \lambda\overline{\xi}.N_0N_1 \dots N_p$. Let Γ be the context $\Gamma = \overline{\xi} : \overline{X}$. Then we have the following sequents: $\emptyset \vdash M : (X_1, \dots, X_n, o)$, $\Gamma \vdash N_0N_1 \dots N_p : o$, $\Gamma \vdash N_i : B_i$ for $i \in 0..p$ with $B_0 = (B_1, \dots, B_p, o)$ and $p \geq 1$.

  There are two subcases, either $N_0 \equiv \xi_i$ where α is a variable in $\overline{\xi}$ and the tree has the following form:



  or N_0 is not a variable and the tree $\tau(M)$ has the following form:



  We only consider the second case since the first one can be treated similarly. Moreover we make the assumption that $p = 1$. It is straightforward to generalize to any $p \geq 1$. We write $\lambda\overline{z}$ to denote the root of the tree $\tau(N_1)$.

  We have:

$$\begin{aligned}
\langle\!\langle M \rangle\!\rangle &= \Lambda^n(\langle\!\langle \Gamma \vdash N_0N_1 : o \rangle\!\rangle) && \text{(game semantics for abstraction)}\\
&\cong \langle\!\langle \Gamma \vdash N_0N_1 : o \rangle\!\rangle && \text{(up to moves retagging)}\\
&= \langle \langle\!\langle \Gamma \vdash N_0 \rangle\!\rangle, \langle\!\langle \Gamma \vdash N_1 \rangle\!\rangle \rangle \,{}^{\circledS^{0..1}}_{\circledS}\, ev && \text{(game semantics for application)}\\
&= \langle \varphi_{N_0}(\mathcal{T}rav^{-@}(N_0)), \varphi_{N_1}(\mathcal{T}rav^{-@}(N_1)) \rangle \,{}^{\circledS^{0..1}}_{\circledS}\, ev && \text{(induction hypothesis)}\\
&= \langle \varphi_M(\mathcal{T}rav^{-@}(N_0)), \varphi_M(\mathcal{T}rav^{-@}(N_1)) \rangle \,{}^{\circledS^{0..1}}_{\circledS}\, ev && (\varphi_M = f(0, q_0) \cup \varphi_{N_0} \cup \varphi_{N_1})\\
&= \underbrace{\langle \varphi_M(\mathcal{T}rav^{-@}(N_0)), \varphi_M(\mathcal{T}rav^{-@}(N_1)) \rangle}_{\sigma} \parallel ev && ({}^{\circledS^{0..1}}_{\circledS} \text{ and } \parallel \text{ are the same operator})
\end{aligned}$$

  The strategies σ and ev are defined on the arena $!A \multimap B$ and $!B \multimap C$ respectively where:

$$\begin{aligned}
A &= \langle\!\langle \Gamma \rangle\!\rangle = \langle\!\langle X_1 \rangle\!\rangle \times \dots \times \langle\!\langle X_n \rangle\!\rangle\\
B &= \langle\!\langle B_0 \rangle\!\rangle \times \langle\!\langle B_1 \rangle\!\rangle = \langle\!\langle B_1' \to o' \rangle\!\rangle \times \langle\!\langle B_1 \rangle\!\rangle\\
C &= \langle\!\langle o \rangle\!\rangle
\end{aligned}$$

We have $u \in \langle\!\langle M \rangle\!\rangle \cong \sigma^\dagger \parallel ev$ if and only if

$$\begin{cases} u \in int(!A, !B, C) \\ u \upharpoonright !A, !B \in \sigma^\dagger \\ u \upharpoonright !B, C \in ev \end{cases}$$

or equivalently

$$\begin{cases} u \in int(!A, !B, C) \\ \text{for any initial } m \text{ in } u \upharpoonright !A, !B \text{ there is } j \in 0..p \text{ such that} \\ \begin{cases} u \upharpoonright !A, B_j, m \in \varphi_M(\mathcal{T}rav^{-@}(N_j)) \\ u \upharpoonright !A, B_k, m = \epsilon \quad \text{for every } k \neq j \end{cases} \end{cases}$$

We first prove that $\langle\!\langle M \rangle\!\rangle \subseteq \varphi_M(\mathcal{T}rav^{-@}(M))$.

Suppose $u \in \langle\!\langle M \rangle\!\rangle$. We give a constructive proof that there exists a sequence of nodes t in N such that $\varphi_M(t - @) = u$ by induction on the length of u. Let q_o be the initial question of the arena $[\![M]\!]$ and q_1 the initial question of $[\![N_0]\!]$.

Base cases:

- $u = \epsilon$ then $\varphi(\epsilon) = u$ where the traversal ϵ is formed with the rule (ϵ).
- If $|u| = 1$ then $u = q_0$ is the initial move in C and $\varphi(\lambda\overline{\xi}) = u$. The traversal $\lambda\overline{\xi}$ is formed with the rule (Root).

Step cases: Suppose that $u' = \varphi_M(t' - @)$ and $u = u' \cdot m \in \langle\!\langle M \rangle\!\rangle$ with $|u| > 1$ for some traversal t' of $\tau(M)$. Let us write m^1 for the last move in u'.

1. Suppose $m \in C$. In C there are no internal moves, the only moves of C are therefore q_0 and v_{q_0} for some $v \in \mathcal{D}$. But q_0 can occur only once in u, therefore since $|u| > 1$ we must have $m = v_{q_0}$ for some $v \in \mathcal{D}$. Since m is an answer move to the initial question, it must be the duplication (performed by the copy-cat evaluation strategy) of the move m^1 played in o'. Hence $m^1 = v_{q_1}$. By the induction hypothesis, n' – the last move in t' – is equal to $\varphi(m^1) = v_{\lambda y_1}$.

   By property 3.2.5(iv), $?(u') = \varphi(?(t' - @))$ and since q_0 is the pending question in u', the first node of t' is also the pending node in t'. This permits us to use the rule (CCAnswer-λ) to produce the traversal $t = t' \cdot v_{\lambda\overline{\xi}}$ where $v_{\lambda\overline{\xi}}$ points to the first node in t'. Clearly, $\varphi(t - @) = u$.
2. Suppose that $m, m^1 \in A \cup B_0$. The strategy ev is responsible for switching thread in B_0 therefore, in the interaction semantics, there must be a copycat move in-between two moves belonging to two different threads. Since m and m^1 are consecutive moves in the sequence u, they must belong to the same thread i.e. there are hereditarily justified by the same initial m_0 in B_0.

   We then have $(u \upharpoonright !A, !B) \upharpoonright m_0 = \varphi_{N_0}(t_0 - @)$ for some traversal t_0 of N_0. Consequently $\varphi_{N_0}(n^1) = m^1$ and $\varphi_{N_0}(n) = m$ where $n^1 \cdot n$ are the last two moves in $t_0 - @$.

   n points to some node in t_0 that also occurs in t'. Let us call n^2 this node. Since $(u \upharpoonright !A, !B) \upharpoonright m_0 = \varphi_{N_0}(t_0 - @)$, n_2 must have the same position in t' as the node justifying m in u'. Hence we just need to take $t = t' \cdot n$ where n points to n^2 in t'.

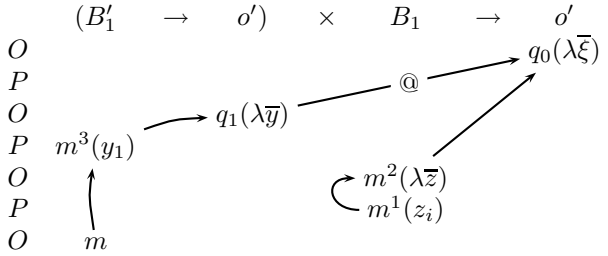
   The sequence t is indeed a valid traversal of $\tau(M)$ because the rule used by the traversal t_0 of $\tau(N_0)$ to visit the node n after n^1 can also be used by the traversal t' of $\tau(M)$ to visit n after n^1. This can be checked formally by inspecting all the traversal rules. The key reason is that all the nodes in $t_0 - @$ are present in t' with the same pointers but with some nodes interleaved in between. However these interleaved nodes are inserted in a way that still permits to use the traversal rule.
3. Suppose that $m, m^1 \in A \cup B_1$. The proof is similar to the previous case.

4. Suppose that $m \in A \cup B_0$ and $m^1 \in A \cup B_1$.

   t is obtained from $t-@$ using the transformation $+@$. We apply the same transformation to u in order to make O-questions and P-questions in u match with λ-nodes and variable nodes in t' respectively. We write this sequence $u+@$. The $+@$ operation inserts nodes in the sequence but not at the end, therefore m^1, the last move in u', is also the last move in $u'+@$. Let us note n^1 for the last move in t'.

   (a) If n^1 is the application node @ then it must be the parent of the node λy_1 since it is the only non-internal @-node present in t'. Therefore $t' = \lambda\overline{\xi} \cdot @$ and $u = q_0 \cdot m$. But m is the copy of q_0 replicated by ev in o' therefore $m = q_1$. Applying the (App) rule on t' produces the traversal $\lambda\overline{\xi} \cdot @ \cdot \lambda y_1$ with $\varphi((\lambda\overline{\xi} \cdot @ \cdot \lambda y_1) - @) = q_0 \cdot q_1 = u$.

   (b) If n^1 is a variable node then m^1 is a P-move and m is an O-move and therefore m is the copy of m^1 duplicated in B_1 by the evaluation strategy. Consequently, m^1 points to some m^2 and m points to the node preceding m^2 denoted by m^3. The diagram below shows an example of such sequence:



   t' and $u + @$ have the following forms:

$$t' = \ldots \cdot n^3 \cdot n^2 \cdot \ldots \cdot n^1$$

$$u + @ = \ldots \cdot m^3 \cdot m^2 \cdot \ldots \cdot m^1 \cdot m$$

   Since n^1 is a variable node, n^2 must be a λ-node. n^3 could be either a variable node or an @-node. In fact n^3 is necessarily a variable node. Indeed, n^3 is mapped to m^3 by φ_{N_0} and m^3 belongs to $[\![B_i']\!]$ (i.e. it is not an internal move of $\langle\!\langle B_i' \rangle\!\rangle$). The function φ_{N_0} is defined in such a way that only nodes which are hereditarily justified by the root of $\tau(N_0)$ are mapped to nodes in $[\![B_1']\!]$. Hence n^3 is hereditarily justified by the root and consequently it cannot be an @-node.
   Hence n^1 is a variable node, n^2 is a λ-node and n^3 is a variable node. We can therefore apply the (Var) rule to t' and we obtain a traversal of the following form:

$$t = \ldots \cdot n^3 \cdot n^2 \cdot \ldots \cdot n^1 \cdot n$$

   We have $\varphi(t' - @) = u'$ by the induction hypothesis and $\varphi(n) = m$ by definition of φ. Therefore since m and n point to the same position we have $\varphi(t - @) = u$.

   (c) If n^1 is the value-leaf of a variable node then we proceed the same way as in the previous case: n^1 is a value-leaf of the variable node n^2 and we can use the (CCAnswer-λ) rule to extend the traversal t'.

   (d) Suppose that n^1 is a lambda node, in which case m^1 is an O-move, then necessarily, m^1 is a move copied by the evaluation strategy from B_1' to B_1. The move following m^1 should also be played in B_1 before being copied back to B_1' by the evaluation strategy. But since $m \in B_0$, this case does not happen.

   (e) If n^1 is a value-leaf of a lambda node then n^2 is a lambda node and n^3 is a variable node. We can therefore use the rule (CCAnswer-var) or (CCAnswer-@) to extend the traversal t'.

5. Suppose $m \in A \cup B_1$ and $m^1 \in A \cup B_0$ then the proof is similar to the previous case.

For the converse, $\varphi_M(\mathcal{T}rav^{-@}(M)) \subseteq \langle\langle M \rangle\rangle$, it is an easy induction on the traversal rules. We omit the details here.

(ii) is an immediate consequence of (i):

$$\begin{aligned}
[\![M]\!] &= \langle\langle M \rangle\rangle \restriction [\![\Gamma \to T]\!] && \text{(eq. 3.2)} \\
&= \varphi_M(\mathcal{T}rav^{-@}(M)) \restriction [\![\Gamma \to T]\!] && \text{(by (i))} \\
&= \varphi_M(\mathcal{T}rav^{\restriction r}(M)) && \text{(lemma 3.2.18)}
\end{aligned}$$

□

Putting corollary 3.2.20 and proposition 3.2.21 together we obtain the following theorem which establish a correspondence between the game-denotation of a term and the set of traversals of its computation tree:

Theorem 3.2.23 (The Correspondence Theorem)**.** *For any pure simply-typed term $\Gamma \vdash M$, φ_M defines a bijection from $\mathcal{T}rav(M)^{\restriction r}$ to $[\![M]\!]$ and a bijection from $\mathcal{T}rav(M)^{-@}$ to $\langle\langle M \rangle\rangle$:*

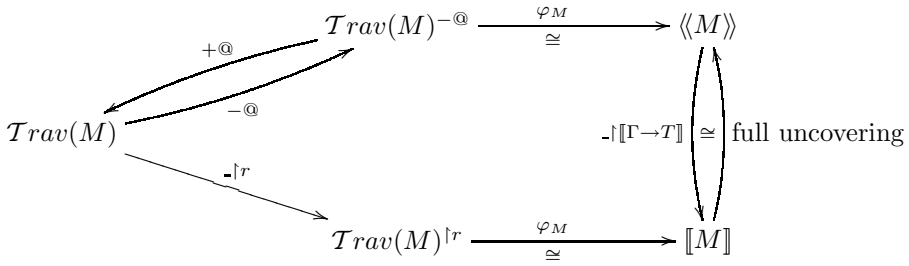
$$\begin{aligned}
\varphi_M &: \mathcal{T}rav(\Gamma \vdash M)^{\restriction r} \xrightarrow{\cong} [\![\Gamma \vdash M]\!] \\
\varphi_M &: \mathcal{T}rav(\Gamma \vdash M)^{-@} \xrightarrow{\cong} \langle\langle \Gamma \vdash M \rangle\rangle
\end{aligned}$$

Moreover if M is in β-normal form and s is a maximal *play then t is a* maximal *traversal.*

Proof. The first part is an immediate consequence of corollary 3.2.20 and proposition 3.2.21.

Finally, if M is in β-normal form then $\mathcal{T}rav(M)^{\restriction r} = \mathcal{T}rav(M)$ therefore φ is a bijection from $\mathcal{T}rav(M)$ to $[\![M]\!]$. Suppose s is a maximal play and suppose $t' \sqsubseteq t$ then since φ is monotonous we have $s = \varphi(t) \sqsubseteq \varphi(t')$. But s is maximal therefore $s = \varphi(t') = \varphi(t)$ and because φ is injective we have $t' = t$. □

The following diagram recapitulates the main results of this section:



Chapter 4

GAME-SEMANTIC CHARACTERISATION OF SAFETY

Safety has been defined as a syntactical constraint. Since Game Semantics is by essence syntax-independent, it seems difficult at first sight to characterise Safety in a game-semantic manner. However, with the help of the tools developed in the previous chapter and using the Correspondence Theorem, we can interpret plays of a strategy as sequences of nodes of some AST of the term. Therefore it is now possible to investigate the impact of the Safety restriction on Game Semantics.

The main theorem of this chapter (theorem 4.1.8) states that pointers in a play of the strategy denotation of a safe term can be uniquely recovered from the underlying sequence of moves. The proof is in several steps. We start by introducing the notion of *incrementally-justified strategies* and prove that for plays of such strategies, pointers can be reconstructed uniquely from the underlying sequences of moves. We then introduce the notion of *incrementally-bound computation trees* and prove that incremental-binding coincides with incremental-justification (proposition 4.1.5). Finally, we show that safe simply-typed terms in β-normal form have incrementally-bound computation trees, consequently the pointers in their game denotation are superfluous.

The first section of this chapter is concerned only with the pure Safe λ-Calculus without interpreted constants. In the next section we extend the result by taking into account the interpreted constants of PCF and IA. We define the language Safe IA (resp. Safe PCF) to be the fragment of IA (resp. PCF) where the application and abstraction rules are constrained the same way as in the Safe λ-Calculus. We show that Safe PCF terms are denoted by incrementally-justified strategies and we give the key elements for a possible extension of the result to Safe Idealized Algol.

4.1 Safe λ-Calculus

Let us consider the Safe λ-Calculus without interpreted constants. Our aim is to prove that pointers in the game semantics of safe terms can be uniquely recovered.

The example of section 1.2.8 gives a good intuition: in order to distinguish the terms $M_1 = \lambda f.f(\lambda x.f(\lambda y.y))$ and $M_2 = \lambda f.f(\lambda x.f(\lambda y.x))$ we have to keep the pointers in the plays of strategies. However, if we limit ourselves to the safe λ-Calculus then the ambiguity disappears because M_1 is safe whereas M_2 is not (in the subterm $f(\lambda y.x)$, the free variable x has the same order as y but x is not abstracted together with y).

Definition 4.1.1 (Incrementally-justified strategy)**.** A strategy $\sigma : A$ is said to be *incrementally-justified* if for any sequence of moves $sq \in P_A$ where q is a question move in M_A we have:

$$\begin{array}{rcl} sq \in \sigma \wedge |s| \text{ even} & \Longrightarrow & q \text{ points to the last P-move in } \llcorner ?(s) \lrcorner \text{ with order strictly greater than } \mathsf{ord}(q); \\ sq \in \sigma \wedge |s| \text{ odd} & \Longrightarrow & q \text{ points to the last O-move in } \ulcorner ?(s) \urcorner \text{ with order strictly greater than } \mathsf{ord}(q). \end{array}$$

Lemma 4.1.2. *Pointers are superfluous for incrementally-justified strategies.*

Proof. Suppose σ is an incrementally-justified strategy. We prove that pointers in a play $s \in \sigma$ are uniquely recoverable by induction on the length of s. *Base case*: if $s \in \sigma$ with $|s| \leq 1$ then there

is no pointer to recover. *Step case*: suppose $sm \in \sigma$. If m is an answer move then thanks to the well-bracketing condition m points to the last unanswered question in s. Suppose m is a question move. If m is a P-move then $|s|$ is odd and since σ is incrementally-justified, m points to the last O-move in $\ulcorner ?(s) \urcorner$ with order strictly greater than $\mathsf{ord}(q)$. Similarly, if m is an O-move then $|s|$ is even and by incremental-justification m points to the last P-move in $\llcorner ?(s) \lrcorner$ with order strictly greater than $\mathsf{ord}(q)$. By the induction hypothesis the pointers in s are recoverable, this ensures that the P-view $\ulcorner ?(s) \urcorner$ and the O-view $\llcorner ?(s) \lrcorner$ can be computed. Consequently the pointer for m is uniquely recoverable. □

Example 4.1.3. The denotation of the evaluation map ev is not incrementally-justified. Indeed consider the play $s = q_0 q_1 q_2 q_3 \in [\![ev]\!]$ shown on the diagram below:

$$\begin{array}{ccccccc} (A & \Longrightarrow & B) & \times & A & \xrightarrow{ev} & B \\ & & & & & & q_0 \\ & & q_1 & & & & \\ q_2 & & & & & & \\ & & & & q_3 & & \end{array}$$

The order of the moves are as follows: $\mathsf{ord}(q_3) = \mathsf{ord}(A)$, $\mathsf{ord}(q_2) = \mathsf{ord}(A)$, $\mathsf{ord}(q_1) = \max(1 + \mathsf{ord}(A), \mathsf{ord}(B))$ and $\mathsf{ord}(q_0) = 1 + \mathsf{ord}(q_1)$. The last O-move in $?(\ulcorner s \urcorner) = s$ with order strictly greater than $\mathsf{ord}(q_3)$ is q_1. But since q_3 points to q_0, $[\![ev]\!]$ is not incrementally-justified.

In a computation tree a binder node always occurs in the path from the bound node to the root. We now introduce a class of computation tree in which binder nodes can be uniquely recovered from the order of the nodes. We write $[n_1, n_2]$ to denote the path from node n_1 to node n_2 if it exists and $]n_1, n_2]$ for the sequence of nodes obtained by removing n_1 from $[n_1, n_2]$.

Definition 4.1.4 (Incrementally-bound computation tree)**.** A variable node x of a computation tree is said to be *incrementally-bound* if either:

1. x is *bound* by the first λ-node in the path to the root that has order strictly greater than $\mathsf{ord}(x)$. Formally:

$$x \text{ bound by } n \quad \Rightarrow \quad n \in [r, x] \wedge \mathsf{ord}(n) > \mathsf{ord}(x) \wedge \forall \lambda\text{-node } n' \in]n, x].\mathsf{ord}(n') \leq \mathsf{ord}(x),$$

2. x is a *free variable* and all the λ-nodes in the path to the root except the root have order smaller or equal to $\mathsf{ord}(x)$. Formally:

$$x \text{ free} \quad \Rightarrow \quad \forall \lambda\text{-node } n' \in]r, x].\mathsf{ord}(n') \leq \mathsf{ord}(x)$$

where r denotes the root of the computation tree.

A computation tree is said to be *incrementally-bound* if all the variable nodes are incrementally-bound.

Proposition 4.1.5 (Incremental-binding coincides with incremental-justification)**.**

(i) If a term in β-normal form has an incrementally-bound computation tree then it is denoted by an incrementally-justified strategy.

(ii) In the pure λ-calculus ($\Sigma = \emptyset$), reciprocally, if a term is denoted by an incrementally-justified strategy then the computation tree of its β-normal is incrementally-bound.

Proof. Let $\Gamma \vdash M : A$ be a simply-typed term in β-normal form and r denotes the root of $\tau(M)$. (i) Suppose that $\tau(M)$ is incrementally-bound. Consider a justified sequence of move $s \in [\![\Gamma \vdash M]\!]$ ending with a question move q (note that q is also the last question in $?(s)$). By proposition 3.2.21, there is a traversal t of $\tau(M)$ such that $\varphi_M(t \restriction r) = s$. We assume that the last node n of t is hereditarily justified by r (otherwise we replace t by its longest prefix verifying this condition). Then n is also the last node in $?(t \restriction r)$ and $t \restriction r$.

- If $|s|$ is even then q is a P-move:
  - Suppose that n is a variable node x bound by a node m occurring in t. Since M is in β-normal form, lemma 3.1.20(i) gives: $\ulcorner ?(t \restriction r) \urcorner = \ulcorner ?(t) \urcorner \restriction r$. By proposition 3.1.17, $\ulcorner ?(t) \urcorner = [r, n]$ and because $\tau(M)$ is incrementally-bound, m is the last λ-node in $[r, n]$ of order strictly greater than $\mathsf{ord}(n)$. Since n is hereditarily justified by the root, so is m and therefore m occurs in $\ulcorner ?(t \restriction r) \urcorner$. But $\ulcorner ?(t \restriction r) \urcorner$ is a subsequence of $\ulcorner ?(t) \urcorner$ therefore m is also the last λ-node in $\ulcorner ?(t \restriction r) \urcorner$ that has order strictly greater than $\mathsf{ord}(n)$.

    By property 3.2.5 (ii), the P-view of $?(s)$ and the P-view of $?(t \restriction r)$ are computed similarly and have the same pointers. This means that node n and move q both point to the same position in the justified sequence $\ulcorner ?(t \restriction r) \urcorner$ and $\ulcorner ?(s) \urcorner$ respectively.

    Finally, since φ maps nodes of a given order to moves of the same order (property 3.2.4), q must point to the last O-move in $\ulcorner ?(s) \urcorner$ whose order is strictly greater than $\mathsf{ord}(q)$.
  - If n is a free variable node x then n is enabled by the root which is the first node in t. By definition of φ, $\varphi(n) = x$ must be a move enabled by the initial move $q_0 = \varphi(r)$ in the arena $[\![\Gamma \to A]\!]$. Therefore $\mathsf{ord}(q_0) > \mathsf{ord}(x)$. Since the computation tree is incrementally-bound, all the λ-nodes in $]r, n]$ have order smaller than $\mathsf{ord}(n)$. Therefore by the correspondence theorem, all the O-moves in $\ulcorner ?(s) \urcorner$ have order smaller than $\mathsf{ord}(x)$.
- If $|s|$ is odd then q is an O-move:

  M is in β-normal form and t is a traversal of $\tau(M)$ whose last node n is hereditarily justified by r. Therefore by lemma 3.1.20 (ii), $\llcorner ?(t \restriction r) \lrcorner = \llcorner ?(t) \lrcorner$.

  A lambda-node always points to its parent node in the computation tree. For terms in β-normal form, this parent node must be a variable node of order strictly greater than $\mathsf{ord}(n)$.

  By inspecting the formation rules for traversals (definition 3.1.13) we remark that a lambda-node occurring in a traversal always points to the last node with order strictly greater that $\mathsf{ord}(n)$ in the O-view of the sequence of unmatched nodes at that point (there are just two cases, n points either to the preceding node or to the third previous node in $\llcorner ?(t) \lrcorner$).

  Similarly, as in the P-move case, we conclude that q points to the last question move in $\llcorner ?(s) \lrcorner$ of order strictly greater than $\mathsf{ord}(q)$.

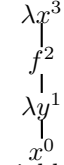
(ii) Suppose that M is β-normal and the strategy $[\![M]\!]$ is incrementally-justified. Let x be a variable node of $\tau(M)$. Since M is β-normal, by lemma 3.1.21, x is either hereditarily justified by the root r or by a constant in N_Σ. In the pure simply-typed λ-calculus we have $\Sigma = \emptyset$, therefore x is hereditarily justified by r.

We remark that for terms in β-normal form, every variable node occurring in the computation tree can be visited by some traversal i.e. there exists a traversal of the form $t \cdot x$ in $\mathcal{T}rav(M)$. The correspondence theorem gives $\varphi((t \cdot x) \restriction r) = \varphi((t \restriction r) \cdot x) \in [\![M]\!]$. Since $[\![M]\!]$ is incrementally-justified, $\varphi(x)$ must point to the last move in $\ulcorner ?(\varphi(t \restriction r)) \urcorner$ with order strictly greater than $\mathsf{ord}(\varphi(x))$. Consequently x points to the last node in $\ulcorner ?(t \restriction r) \urcorner$ with order strictly greater than $\mathsf{ord}(x)$. We have:

$$\begin{aligned}
\ulcorner ?(t \restriction r) \urcorner = \ulcorner ?(t) \restriction r \urcorner = \ulcorner ?(t) \urcorner \restriction r & \qquad \text{(by lemma 3.1.20)} \\
= \ulcorner ?(t) \urcorner & \qquad (M \text{ is a } \beta\text{-nf and } N_\Sigma = \emptyset) \\
= [r, x[& \qquad \text{(by proposition 3.1.17).}
\end{aligned}$$

Therefore if x is a bound variable node then it is bound by the last λ-node in $[r, x[$ with order strictly greater than $\mathsf{ord}(x)$ and if x is a free variable then it points to r and therefore all the λ-node in $]r, x[$ have order smaller than $\mathsf{ord}(x)$. Hence $\tau(M)$ is incrementally-bound. □

Examples: Consider the β-normal term $\lambda x.f(\lambda y.x)$ where $x, y : o$ and $f : (o, o), o$. The figure on the right represents the computation tree with the order of each node in the exponent part. Since node x of order 0 is not bound by the order 1 node λy, $\tau(M)$ is not incrementally-bound and by proposition 4.1.5 $[\![\lambda x.f(\lambda y.x)]\!]$ is not incrementally-justified. Similarly we can check that the denotation of $f(\lambda y.x)$ is not incrementally-justified whereas $\lambda y.x$ has an incrementally-justified denotation. Also for any higher-order variable $x : A$, the computation tree $\tau(x)$ is incrementally-bound, therefore the projection strategies π_i are incrementally-justified. From these examples we observe that application does not preserve incremental-justification: $[\![f]\!]$ and $[\![\lambda y.x]\!]$ are incrementally-justified whereas $[\![f(\lambda y.x)]\!]$ is not.



Lemma 4.1.6 (Safe terms have incrementally-bound computation trees)**.** *Let $\Gamma \vdash M$ be a simply-typed term.*

(i) If M is a safe term then $\tau(M)$ is incrementally-bound ;

(ii) reciprocally, if M is closed *and $\tau(M)$ is incrementally-bound then the η-normal form of M is safe.*

Proof. (i) Suppose that M is safe. The safety property is preserved after taking the η-normal form, therefore $\eta_{\mathsf{nf}}(M)$ is also safe. Hence $\tau(M)$ is the tree representation of a safe term.

When applying the abstraction rule in the Safe λ-Calculus, the variables in the lowest partition (smallest order) of the context must all be abstracted together. Moreover in the computation tree, consecutive abstractions are merged into a single node, therefore the safety of $\eta_{\mathsf{nf}}(M)$ implies that for each λ-node $\lambda\overline{\xi}$, any variable x occurring free in $\kappa(\lambda\overline{\xi})$ has 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. In a computation tree, a binder node always occurs in the path from the bound node to the root, therefore, according to the previous observation, x must be bound by the first λ-node occurring in $[r, x]$ with order strictly greater than $\mathsf{ord}(x)$. Similarly, let x be a free variable node in τ then x is not bound by any of the λ-nodes occurring in $[r, x]$. Once again, by the previous observation, all these λ-nodes have order smaller than $\mathsf{ord}(x)$. Hence τ is incrementally-bound.

(ii) We assume that M is already in η-normal form. Suppose M is closed and $\tau(M)$ is incrementally-bound, we prove that M is safe by induction on its structure: *Base case:* $M = \lambda\overline{\xi}.\alpha$ for some variable or constant α. This term is obviously safe.

Step case: If $M = \lambda\overline{\xi}.N_1 \dots N_p$. Let i range over $1..p$. N_i can be written $\lambda\overline{\eta_i}.N_i'$ where N_i' is not an abstraction. By the induction hypothesis, $\lambda\overline{\xi}.N_i = \lambda\overline{\xi}\,\overline{\eta_i}.N_i'$ is safe. We observe from the formation rules of Safe λ-Calculus that the typing judgment for $\lambda\overline{\xi}\,\overline{\eta_i}.N_i'$ can only be derived using the (abs) rule on the term N_i'. Hence N_i' is necessarily safe. Let z be a variable occurring free in N_i'. Since M is closed, z is either bound by $\lambda\overline{\eta_1}$ or $\lambda\overline{\xi}$. If it is bound by $\lambda\overline{\xi}$ then because $\tau(M)$ is incrementally-bound we have $\mathsf{ord}(z) \geq \mathsf{ord}(\lambda\overline{\eta_1}) = \mathsf{ord}(N_i)$. Hence we can abstract the variables $\overline{\eta_1}$ using the (abs) rule and we obtain that N_i is safe.

Because M is in η-normal form, the application $N_1 \dots N_p$ is total (i.e. N_1 is a function taking $p-1$ parameters and it is applied to $p-1$ arguments), therefore since the N_is are safe, by the (app) rule of the Safe λ-Calculus $N_1 \dots N_p$ is also safe. Finally, using the (abs) rule we conclude that $M = \lambda\overline{\xi}.N_1 \dots N_p$ is safe. □

Note that the hypothesis that M is closed in (ii) is necessary. For instance, the two terms $\lambda xy.x$ and $\lambda y.x$, where $x, y : o$, have (isomorphic) incrementally-bound computation trees. However $\lambda xy.x$ is safe whereas $\lambda y.x$ is not.

Putting proposition 4.1.5 and lemma 4.1.6 together we obtain a game-semantic characterisation of safe terms:

Corollary 4.1.7 (Incrementally-justified strategies characterise closed safe terms)**.** *Let M be a closed pure simply-typed term (with no constants) then:*

$$[\![M]\!] \textit{ is incrementally-justified if and only if } \eta\beta_{\mathsf{nf}}(M) \textit{ is safe,}$$

where $\eta\beta_{\mathsf{nf}}(M)$ denotes the η-normal form of the β-normal form of M.

Theorem 4.1.8 (Pointers are superfluous for safe terms)**.** *Pointers in the game semantics of safe terms are uniquely recoverable.*

Proof. Let M be a safe simply-typed term. The β-normal form of M denoted by M' is also safe. By lemma 4.1.6 (i), $\tau(M')$ is incrementally-bound and by proposition 4.1.5, $[\![M']\!]$ is an incrementally-justified strategy. By lemma 4.1.2, the pointers in $[\![M']\!]$ are uniquely recoverable. Finally, the soundness of the game model gives $[\![M]\!] = [\![M']\!]$. □

4.2 Safe PCF and Safe Idealized Algol

Safe Idealized Algol, or Safe IA for short, is Idealized Algol where the application and abstraction rules are restricted the same way as in the Safe λ-Calculus (see rules of section 2.2).

The properties of the Safe λ-Calculus can be transposed straightforwardly to Safe IA. In particular, it can be shown that safety is preserved by β-reduction and that no variable capture occurs when performing substitution on a safe term.

A natural question to ask is whether we can extend the result about game semantics of safe λ-terms to safe IA-terms. In this section we lay out the key elements permitting to prove that the pointers in the game semantics of safe IA terms can be recovered uniquely.

Such result has potential application in algorithmic game semantics. For instance, by following the framework of Ghica and McCusker [2000], it may be possible to give a characterisation of the game semantics of some higher-order fragments of Safe IA using extended regular expressions. Subsequently, this would lead to the decidability of program equivalence for the considered fragment.

4.2.1 Formation rules of Safe IA

We call safe IA term any term that is typable within the following system of formation rules:

$$(\mathbf{var})\ \frac{}{x : A \vdash x : A} \qquad (\mathbf{wk})\ \frac{\Gamma \vdash M : A}{\Delta \vdash M : A}\quad \Gamma \subset \Delta$$

$$(\mathbf{app})\ \frac{\Gamma \vdash M : (A, \ldots, A_l, B) \qquad \Gamma \vdash N_1 : A_1 \quad \ldots \quad \Gamma \vdash N_l : A_l}{\Gamma \vdash M N_1 \ldots N_l : B}\quad \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(B)$$

$$(\mathbf{abs})\ \frac{\Gamma \cup \overline{x} : \overline{A} \vdash M : B}{\Gamma \vdash \lambda \overline{x} : \overline{A}.M : (\overline{A}, B)} \qquad \forall y \in \Gamma : \mathsf{ord}(y) \geq \mathsf{ord}(\overline{A}, B)$$

$$(\mathbf{num})\frac{}{\Gamma \vdash n : \mathtt{exp}} \qquad (\mathbf{succ})\frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}} \qquad (\mathbf{pred})\frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{pred}\ M : \mathtt{exp}}$$

$$(\mathbf{cond})\frac{\Gamma \vdash M : \mathtt{exp} \qquad \Gamma \vdash N_1 : \mathtt{exp} \qquad \Gamma \vdash N_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2} \qquad (\mathbf{rec})\frac{\Gamma \vdash M : A \to A}{\Gamma \vdash Y_A M : A}$$

$$(\mathbf{seq})\frac{\Gamma \vdash M : \mathtt{com} \quad \Gamma \vdash N : A}{\Gamma \vdash \mathtt{seq}_A\ M\ N\ : A}\quad A \in \{\mathtt{com}, \mathtt{exp}\}$$

$$(\mathbf{assign})\frac{\Gamma \vdash M : \mathtt{var} \quad \Gamma \vdash N : \mathtt{exp}}{\Gamma \vdash \mathtt{assign}\ M\ N\ : \mathtt{com}} \qquad (\mathbf{deref})\frac{\Gamma \vdash M : \mathtt{var}}{\Gamma \vdash \mathtt{deref}\ M\ : \mathtt{exp}}$$

$$(\mathbf{new})\frac{\Gamma, x : \mathtt{var} \vdash M : A}{\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M}\quad A \in \{\mathtt{com}, \mathtt{exp}\}$$

$$(\mathbf{mkvar})\frac{\Gamma \vdash M_1 : \mathtt{exp} \to \mathtt{com} \quad \Gamma \vdash M_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{mkvar}\ M_1\ M_2\ : \mathtt{var}}$$

4.2.2 Small-step semantics of Safe IA

In the first chapter we defined the operational semantics of IA using a big step semantics. The operational semantics of IA can be defined equivalently using a small-step semantics. The reduction rules of the small-step semantics are of the form $s, e \to s', e'$ where s and s' denotes the stores and e and e' denotes IA expressions.

Let us give the rules that tell how to reduce redexes:

- the reduction of safe-redex (relation β_s from definition 2.1.9);
- reduction rules for PCF constants:

$$\begin{array}{rcl} \texttt{succ}\ n & \to & n+1 \\ \texttt{pred}\ n+1 & \to & n \\ \texttt{pred}\ 0 & \to & 0 \\ \texttt{cond}\ 0\ N_1 N_2 & \to & N_1 \\ \texttt{cond}\ n+1\ N_1 N_2 & \to & N_2 \\ Y\ M & \to & M(YM) \end{array}$$

- reduction rules for IA constants:

$$\begin{array}{rcl} \texttt{seq skip}\ M & \to & M \\ s, \texttt{new}\ x\ \texttt{in}\ M & \to & (s|x \mapsto 0), M \\ s, \texttt{assign}\ x\ n & \to & (s|x \mapsto n), \texttt{skip} \\ s, \texttt{deref}\ x & \to & s, s(x) \\ \texttt{assign}\ (\texttt{mkvar}MN)\ n & \to & Mn \\ \texttt{deref}\ (\texttt{mkvar}MN) & \to & N \end{array}$$

Redex can also be reduced when they occur as subexpressions within a larger expression. We make use of evaluation contexts to indicate when such reduction can happen. Evaluation contexts are given by the following grammar:

$$\begin{array}{rcl} E[-] & ::= & -\mid EN \mid \texttt{succ}\ E \mid \texttt{pred}\ E \mid \texttt{cond}\ E\ N_1\ N_2 \mid \\ & & \texttt{seq}\ E\ N \mid \texttt{deref}\ E \mid \texttt{assign}\ E\ n \mid \texttt{assign}\ M\ E \mid \\ & & \texttt{mkvar}\ M\ E \mid \texttt{mkvar}\ E\ M \mid \texttt{new}\ x\ \texttt{in}\ E. \end{array}$$

The small-step semantics is completed with following rule:

$$\frac{M \to N}{E[M] \to E[N]}$$

Lemma 4.2.1 (Reduction preserves safety)**.** *Let M be a safe IA term. If $M \to N$ then N is also a safe term.*

This can be proved easily by induction on the structure of M.

4.2.3 Safe PCF fragment

In this section, we show how to extend the results obtained for the Safe λ-Calculus to the PCF fragment of Safe IA.

The Y combinator needs a special treatment. In order to deal with it, we follow the idea of Abramsky and McCusker [1998b]: we consider the sublanguage PCF_1 of PCF in which the only allowed use of the Y combinator is in terms of the form $Y(\lambda x : A.x)$ for some type A. We will write Ω_A to denote the non-terminating term $Y(\lambda x : A.x)$ for a given type A.

We introduce the *syntactic approximants* to $Y_A M$:

$$\begin{aligned} Y_A^0 M &= \Gamma \vdash \Omega_A : A \\ Y_A^{n+1} M &= M(Y^n M) \end{aligned}$$

For any PCF term M and natural number n, we define M_n to be the PCF_1 term obtained from M by replacing each subterm of the form YN with $Y^n N_n$. We have $[\![M]\!] = \bigcup_{n\in\omega} [\![M_n]\!]$ (Abramsky and McCusker [1998b], lemma 16).

Computation tree

We would like to define a unique computation tree for terms that use the Y combinator.

Let us first define the computation tree for PCF_1 terms. We introduce a special Σ-constant $\bot$ representing the non-terminating computation of ground type Ω_o. Given any type $A = (A_1, \ldots, A_n, o)$, the computation tree $\tau(\Omega_A)$ is defined to be the tree representation of $\lambda x_1 : A_1 \ldots x_n : A_n.\bot$. The computation tree of a PCF_1 term is then computed inductively in the standard way.

We now introduce a partial order on the set of computation trees.

A *tree* t is a labelling function $t : T \to L$ where T, called the domain of t and written $dom(t)$, is a non-empty prefix-closed subset of some free monoid X^* and L denotes the set of possible labels. Intuitively, T represents the structure of the tree (the set of all paths) and t is the labelling function mapping paths to labels. Trees can be ordered using the *approximation ordering* defined in Knapik et al. [2002], section 1: we write $t' \sqsubseteq t$ if the tree t' is obtained from t by replacing some of its subtrees by $\bot$. Formally:

$$t' \sqsubseteq t \quad \iff dom(t') \subseteq dom(t) \wedge \forall w \in dom(t').(t'(w) = t(w) \vee t'(w) = \bot).$$

The set of all trees together with the approximation ordering is a complete partial order.

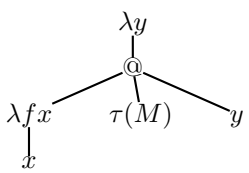
We now consider a strict subset of the set of all trees: the set of computation trees. A computation tree is a tree which represents the η-normal form of some (potentially infinite) PCF term. In other words a tree is a computation tree if it can be written $\tau(M)$ for some infinite PCF term M. The set L of labels is constituted of the Σ-constants, @, the special constant $\bot$, variables and abstractions of any sequence of variables. We will write $(CT, \sqsubseteq)$ to denote the set of computation trees ordered by the approximation ordering $\sqsubseteq$ defined above. $(CT, \sqsubseteq)$ is also a complete partial order.

It is easy to check that the sequence of computation trees $(\tau(M_n))_{n\in\omega}$ is a chain. We can therefore define the computation tree of a PCF term M to be the least upper-bound of the chain of computation trees of its approximants:

$$\tau(M) = \bigcup_{n\in\omega} (\tau(M_n))_{n\in\omega}.$$

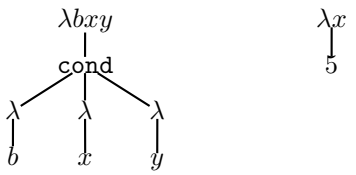
In other words, we construct the computation tree by expanding infinitely any subterm of the form YM. For instance consider the term $M = Y(\lambda fx.fx)$ where $f : (o, o)$ and $x : o$. Its computation tree $\tau(M)$, represented below, is a tree representation of the η-normal form of the infinite term $(\lambda fx.fx)((\lambda fx.fx)((\lambda fx.fx)(\ldots$

$\tau(M) =$



The remaining operators of IA are treated as standard constants and the corresponding computation tree is constructed from the η-normal form of the term in the standard way. For instance

the diagram below shows the computation tree for $\texttt{cond}\ b\ x\ y$ (left) and $\lambda x.5$ (right):



The node labelled 5 has, like any other node, children value-leaves which are not represented on the diagram above for simplicity.

Traversal

New traversal rules accompany the additional constants of IA. There is one additional rule for natural number constants:

- (Nat) If $t \cdot n$ is a traversal where n denotes a node labelled with some numeral constant $i \in \mathbb{N}$ then $t \cdot n \cdot i_n$ is also a traversal where i_n denotes the value-leaf of m corresponding to the value $i \in \mathbb{N}$.

The traversals rules for `pred` and `succ` are defined similarly. For instance, the rules for `succ` are:

- (Succ) If $t \cdot \texttt{succ}$ is a traversal and λ denotes the only child node of `succ` then $t \cdot \overset{1}{\texttt{succ}} \cdot \lambda$ is also a traversal.
- (Succ') If $t_1 \cdot \overset{1}{\texttt{succ}} \cdot \lambda \cdot t_2 \cdot i_\lambda$ is a traversal for some $i \in \mathbb{N}$ then $t_1 \cdot \overset{1}{\texttt{succ}} \cdot \lambda \cdot t_2 \cdot i_\lambda \cdot (i+1)_{\texttt{succ}}$ is also a traversal.

In the computation tree, nodes labelled with `cond` have three children nodes numbered from 1 to 3 corresponding to the three parameters of the operator `cond`. The traversal rules are:

- (Cond-If) If $t_1 \cdot \texttt{cond}$ is a traversal and λ denotes the first child of `cond` then $t_1 \cdot \overset{1}{\texttt{cond}} \cdot \lambda$ is also a traversal.
- (Cond-ThenElse) If $t_1 \cdot \overset{1}{\texttt{cond}} \cdot \lambda \cdot t_2 \cdot i_\lambda$ then $t_1 \cdot \overset{1}{\texttt{cond}} \cdot \lambda \cdot t_2 \cdot i_\lambda \cdot \overset{2+[i>0]}{\lambda}$ is also a traversal.
- (Cond') If $t_1 \cdot \overset{k}{\texttt{cond}} \cdot t_2 \cdot \lambda \cdot t_3 \cdot i_\lambda$ for $k = 2$ or $k = 3$ then $t_1 \cdot \overset{k}{\texttt{cond}} \cdot t_2 \cdot \lambda \cdot t_3 \cdot i_\lambda \cdot i_{\texttt{cond}}$ is also a traversal.

It is easy to verify that these traversal rules are all well-behaved and therefore condition (WB) of section 3.1.4 is met. This completes the definition of traversal for the PCF subset of IA.

Interaction semantics

We need to complete the interaction semantics of section 3.2.2 to take into account the constants of the language. Let $f : (A_1, \ldots, A_p) \in \Sigma$ be a higher-order constant denoted by the strategy $[\![f]\!]$ in the standard semantics. We complete definition 3.2.12 by defining the revealed strategy of a term of the form $\lambda\overline{\xi}.fN_1 \ldots N_p$ as follows:

$$\langle\!\langle \lambda\overline{\xi}.fN_1 \ldots N_p \rangle\!\rangle = \langle \langle\!\langle N_1 \rangle\!\rangle, \ldots, \langle\!\langle N_p \rangle\!\rangle \rangle \fatsemi^{0..p-1} [\![f]\!].$$

Removing Σ-nodes from the traversals

To establish the correspondence with the interaction semantics, we need to remove the superfluous nodes from the traversals. These nodes are the @-nodes and the constant nodes. We will use the operation $-@$ (definition 3.2.15) to filter out the @-nodes and we introduce a similar operation $-\Sigma$ to eliminate the Σ-nodes.

Definition 4.2.2 (Hiding Σ-constants in the traversals)**.** Let t be a traversal of $\tau(M)$. We write $t - \Sigma$ for the sequence of nodes with pointers obtained by

- removing from t all nodes labelled with a Σ-constant or value-leaf justified by a Σ-constant,
- replacing any link pointing to a Σ-constant f by a link pointing to the predecessor of f in t.

Suppose $u = t - \Sigma$ is a sequence of nodes obtained by applying the previously defined transformation on the traversal t, then t can be partially recovered from u by reinserting the Σ-nodes as follows. For each Σ-node f, where p denotes the parent node of f, do the following:

1. replace every occurrence of the pattern $p \cdot n$ in u where n is a λ-node by $p \cdot f \cdot n$;
2. replace any link in u starting from a λ-node and pointing to p by a link pointing to the inserted node f;
3. for each occurrence in u of a value-leaf v_p pointing to p, add the value-leaf v_f immediately before v_p. The links of v_f points to the node immediately following p.

We write $u + \Sigma$ for this second transformation.

These transformations are well-defined since in a traversal, a Σ-node f always follows immediately its parent λ-node p, and an occurrence of a value-node v_p always follows immediately a value-node v_f. In other words, if f occurs in t then t must be a prefix of a traversal of the following form for some $v \in \mathcal{D}$:

$$\ldots \cdot p \cdot f \cdot \ldots \cdot v_f \cdot v_p \cdot \ldots$$

Remark: $t - \Sigma$ is not a proper traversal since it does not satisfy alternation. It is not a proper justified sequence either since after removing a Σ-node f, any λ-node justified by f will become justified by the parent of f which is also a λ-node.

The following lemma follows directly from the definition:

Lemma 4.2.3. *For any traversal t we have $(t - \Sigma) + \Sigma \sqsubseteq t$ and if t does not end with an Σ-node or a value-leaf of a Σ-node then $(t - \Sigma) + \Sigma = t$.*

The operations $-@$ and $-\Sigma$ are commutative: $(t - @) - \Sigma = (t - \Sigma) - @$. We write t^* to denote $(t - @) - \Sigma$ i.e. the sequence obtained from t by removing all the @-nodes as well as the constant nodes together with their associated value-leaves. We introduce the notation $\mathcal{T}rav(M)^* = \{t^* \mid t \in \mathcal{T}rav(M)\}$.

Lemma 4.2.4 (Filtering lemma)**.** *Let $\Gamma \vdash M : T$ be a term and r be the root of $\tau(M)$. For any traversal t of the computation tree we have $\varphi(\mathcal{T}rav^*(M)) \upharpoonright [\![\Gamma \to T]\!] = \varphi(\mathcal{T}rav^{\upharpoonright r}(M))$. Consequently,*

$$\varphi(t^*) \upharpoonright [\![\Gamma \to T]\!] = \varphi(t \upharpoonright r).$$

Proof. From the definition of φ, the nodes of the computation tree that φ maps to moves in the arena $[\![\Gamma \to T]\!]$ are exactly the nodes that are hereditarily justified by r. The result follows from the fact that @-nodes, constant nodes and value-leaves of constant nodes are not hereditarily justified by the root. □

The following lemma is the counterpart of lemma 3.2.19 and it is proved identically.

Lemma 4.2.5 (φ is injective)**.** *φ regarded as a function defined on the set of sequences of nodes is injective in the sense that for any two traversals t_1 and t_2:*

(i) if $\varphi(t_1^*) = \varphi(t_2^*)$ then $t_1^* = t_2^*$;

(ii) if $\varphi(t_1 \upharpoonright r) = \varphi(t_2 \upharpoonright r)$ then $t_1 \upharpoonright r = t_2 \upharpoonright r$.

Corollary 4.2.6.

(i) φ defines a bijection from $\mathcal{T}rav(M)^*$ to $\varphi(\mathcal{T}rav(M)^*)$;

(ii) φ defines a bijection from $\mathcal{T}rav(M)^{\upharpoonright r}$ to $\varphi(\mathcal{T}rav(M)^{\upharpoonright r})$.

Correspondence theorem

We would like to prove the counterpart of proposition 3.2.21 in the context of the simply-typed λ-calculus *with interpreted PCF constants*. The game model of the language PCF is given by the category $\mathcal{C}_b$ of well-bracketed strategies. Hence the well-bracketing assumption stated in section 3.2 is satisfied.

We first prove that $\mathcal{T}rav^{\restriction r}$ is continuous.

Lemma 4.2.7. *Let $(S, \subseteq)$ denote the set of sets of justified sequences of nodes ordered by subset inclusion. The function $\mathcal{T}rav^{\restriction r} : (CT, \sqsubseteq) \to (S, \subseteq)$ is continuous.*

Proof.

Monotonicity: Let T and T' be two computation trees such that $T \sqsubseteq T'$ and let t be some traversal of T. Traversals ending with a node labelled $\bot$ are maximal therefore $\bot$ can only occur at the last position in a traversal. Let us prove the following two properties:

(i) If $t = t \cdot n$ with $n \neq \bot$ then t is a traversal of T';

(ii) if $t = t_1 \cdot \bot$ then $t_1 \in \mathcal{T}rav(T')$.

(i) By induction on the length of t. It is trivial for the empty traversal. Suppose that $t = t_1 \cdot n$ is a traversal with $n \neq \bot$. By the induction hypothesis, t_1 is a traversal of T'.

We observe that for all traversal rules, the traversal produced is of the form $t_1 \cdot n$ where n is defined to be a child node or value-leaf of some node m occurring in t_1. Moreover, the choice of the node n only depends on the traversal t_1 (for the constant rules, this is guaranteed by assumption (WB)).

Since $T \sqsubseteq T'$, any node m occurring in t_1 belongs to T' and the children nodes and leaves of m in T also belong to the tree T'. Hence n is also present in T' and the rule used to produce the traversal t of T can be used to produce the traversal t of T'.

(ii) $\bot$ can only occur at the last position in a traversal therefore t_1 does not end with $\bot$ and by (i) we have $t_1 \in \mathcal{T}rav(T')$.

Hence we have:

$$\begin{aligned}
\mathcal{T}rav(T)^{\restriction r} &= \{t \restriction r \mid t \in \mathcal{T}rav(T)\} \\
&= \{(t \cdot n) \restriction r \mid t \cdot n \in \mathcal{T}rav(T) \wedge n \neq \bot\} \cup \{(t \cdot \bot) \restriction r \mid t \cdot \bot \in \mathcal{T}rav(T)\} \\
\text{(by (i) and (ii))} \quad &\subseteq \{(t \cdot n) \restriction r \mid t \cdot n \in \mathcal{T}rav(T') \wedge n \neq \bot\} \cup \{t \restriction r \mid t \in \mathcal{T}rav(T')\} \\
&= \mathcal{T}rav(T')^{\restriction r}
\end{aligned}$$

Continuity: Let $t \in \mathcal{T}rav\left(\bigcup_{n\in\omega} T_n\right)$. We write t_i for the finite prefix of t of length i. The set of traversals is prefix-closed therefore $t_i \in \mathcal{T}rav\left(\bigcup_{n\in\omega} T_n\right)$ for any i. Since t_i has finite length we have $t_i \in \mathcal{T}rav(T_{j_i})$ for some $j_i \in \omega$. Therefore we have:

$$\begin{aligned}
t \restriction r &= (\bigvee_{i\in\omega} t_i) \restriction r && \text{(the sequence } (t_i)_{i\in\omega} \text{ converges to } t) \\
&= \bigcup_{i\in\omega} (t_i \restriction r) && (_ \restriction r \text{ is continuous, lemma 3.1.8}) \\
&\in \bigcup_{i\in\omega} \mathcal{T}rav^{\restriction r}(T_{j_i}) && (t_i \in \mathcal{T}rav(T_{j_i})) \\
&\subseteq \bigcup_{i\in\omega} \mathcal{T}rav^{\restriction r}(T_i) && (\text{since } \{j_i \mid i \in \omega\} \subseteq \omega)
\end{aligned}$$

Hence $\mathcal{T}rav^{\restriction r}(\bigcup_{n\in\omega} T_n) \subseteq \bigcup_{n\in\omega} \mathcal{T}rav^{\restriction r}(T_n)$.

□

Proposition 4.2.8. *Let $\Gamma \vdash M : T$ be a PCF term and r be the root of $\tau(M)$. Then:*

$$(i) \quad \varphi_M(\mathcal{T}rav(M)^*) = \langle\langle M \rangle\rangle,$$
$$(ii) \quad \varphi_M(\mathcal{T}rav(M)^{\restriction r}) = [\![M]\!].$$

Proof. We first prove the result for PCF_1: (i) The proof is an induction identical to the proof of proposition 3.2.21. However we need to complete the case analysis with the Σ-constant cases:

- The cases `succ`, `pred`, `cond` and numeral constants are straightforward.
- Suppose $M = \Omega_o$ then $\mathcal{T}rav(\Omega_o) = \mathsf{Pref}(\{\lambda \cdot \bot\})$ therefore $\mathcal{T}rav(\Omega_o)^{\restriction r} = \mathsf{Pref}(\{\lambda\})$ and $[\![\Omega_o]\!] = \mathsf{Pref}(\{q\})$ with $\varphi(\lambda) = q$. Hence $[\![\Omega_o]\!] = \varphi(\mathcal{T}rav(\Omega_o)^{\restriction r})$.

(ii) is a direct consequence of (i) and the filtering lemma (lemma 4.2.4).

We now extend the result to PCF. Let M be a PCF term, we have:

$$\begin{aligned}
[\![M]\!] &= \bigcup_{n\in\omega} [\![M_n]\!] && \text{(Abramsky and McCusker [1998b], lemma 16)}\\
&= \bigcup_{n\in\omega} \mathcal{T}rav^{\restriction r}(\tau(M_n)) && (M_n \text{ is a } \mathsf{PCF}_1 \text{ term})\\
&= \mathcal{T}rav^{\restriction r}(\bigcup_{n\in\omega} \tau(M_n)) && \text{(by continuity of } \mathcal{T}rav^{\restriction r}\text{, lemma 4.2.7)}\\
&= \mathcal{T}rav^{\restriction r}(\tau(M)) && \text{(by definition of } \tau(M))\\
&= \mathcal{T}rav^{\restriction r}(M) && \text{(abbreviation).}
\end{aligned}$$

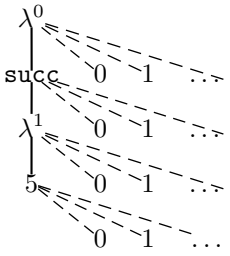
□

Hence by corollary 4.2.6, φ defines a bijection from $\mathcal{T}rav(M)^{\restriction r}$ to $[\![M]\!]$:

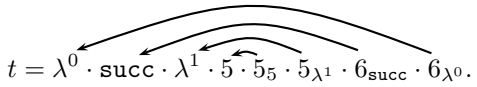
$$\varphi : \mathcal{T}rav(M)^{\restriction r} \xrightarrow{\cong} [\![M]\!].$$

Example: `succ`

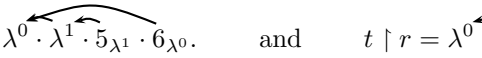
Consider the term $M =$ `succ` 5 whose computation tree is represented below. The value-leaves are also represented on the diagram, they are the vertices attached to their parent node with a dashed line.



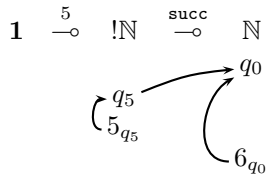
The following sequence of nodes is a traversal of $\tau(M)$:



The subsequences t^* and $t \restriction r$ are given by:

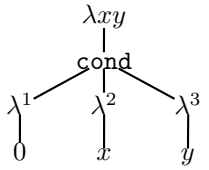


We have $\varphi(t^*) = q_0 \cdot q_5 \cdot 5_{q_5} \cdot 5_{q_0}$ and $\varphi(t \restriction r) = q_0 \cdot 5_{q_0}$ where q_0 and q_5 denote the roots of two flat arenas over $\mathbb{N}$. These two sequences of moves correspond to some play of the interaction semantics and the standard semantics respectively. The interaction play is represented below:

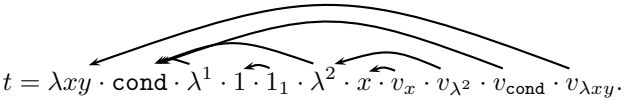


Another example : `cond`

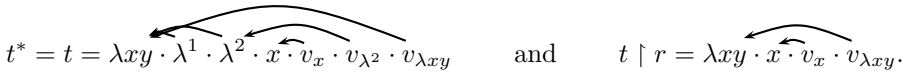
Consider the term $M = \lambda xy.\texttt{cond}\ 1\ x\ y$. Its computation tree is represented below (without the value-leaves):



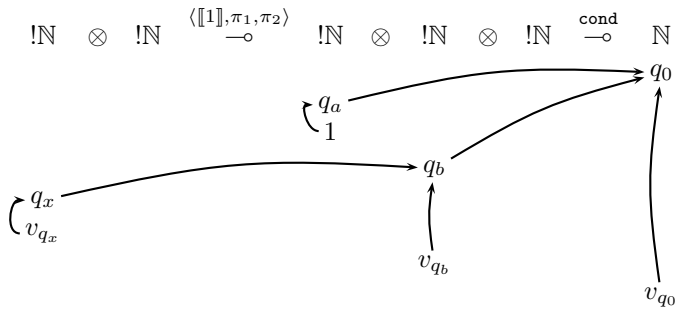
For any value $v \in \mathcal{D}$ the following sequence of nodes is a traversal of $\tau(M)$:



The subsequences t^* and $t \restriction r$ are given by:



The sequence of moves $\varphi(t^*)$ corresponds to some play of the interaction semantics and the sequence $\varphi(t \restriction r)$ is a play of the standard semantics obtained by hiding the internal moves of $\varphi(t^*)$. The interaction play $\varphi(t^*)$ is represented below:



Game characterisation of safe terms

A difficulty arises because of the presence of the Y combinator : computation trees of PCF terms are potentially infinite. Despite this particularity, lemma 4.1.6 still holds in the PCF setting:

Lemma 4.2.9 (Safe terms have incrementally-bound computation tree)**.** *If $\Gamma \vdash M$ is a safe PCF term then $\tau(M)$ is incrementally-bound.*

Proof. Suppose that a variable node z belongs to $\tau(M) = \bigcup_{k\in\omega} \tau(M_k)$, then there exists $k \in \omega$ such that z belongs to $\tau(M_k) \sqsubseteq \tau(M)$. Moreover if we write r_k to denote the root of the tree $\tau(M_k)$ then the path $[r_k, z]$ in $\tau(M_k)$ is equal to the path $[r, z]$ in $\tau(M)$.

Let i denote the number of occurrences of the Y combinator in M. We prove by induction on i that M_k is safe for any $k \in \omega$. *Base case:* $i = 0$ then $M_k = M$. *Step case:* $i > 0$. Let $Y_A N$ be a subterm of M. Since M is safe, N is also safe. The number of occurrences of the Y combinator in N is smaller than i therefore by the induction hypothesis N_k is safe. Consequently the term $Y_A^k N_k = \underbrace{N_k(\ldots(N_k}_{k \text{ times}}\,\Omega)\ldots)$ is also safe and by compositionality so is M_k.

Clearly, lemma 4.1.6 is still valid in PCF_1 (the subterms of the form Ω are just represented by the constant $\bot$ in the computation tree), therefore since M_k is in PCF_1, $\tau(M_k)$ must be incrementally-bound. Consequently the node z is incrementally-bound in $\tau(M_k)$ and since $[r, z] = [r_k, z]$, the corresponding node z in the tree $\tau(M)$ is also incrementally-bound.

Hence $\tau(M)$ is incrementally-bound. □

Theorem 4.2.10 (Pointers are superfluous for Safe PCF)**.** *Pointers in the game denotation of safe PCF terms are uniquely recoverable.*

Proof. Since condition (WB) is verified, lemma 3.1.20 holds in the Safe PCF setting. Therefore proposition 4.1.5(i) also holds: terms in β-nf with an incrementally-bound computation tree are denoted by incrementally-justified strategies.

Safety is preserved by reduction (lemma 4.2.1), therefore by lemma 4.2.9 and soundness of the game denotation, any Safe PCF term must be denoted by an incrementally-justified strategy.

Hence pointers are superfluous in the game denotation of safe PCF terms. □

4.2.4 Safe IA

We are now in a position to consider the full Safe Idealized Algol language. The general idea is the same as for Safe PCF, however there are some difficulties caused by the presence of the two new base types `var` and `com`. We just give indications on how to adapt our framework to the particular case of Safe IA without giving the complete proofs. However we believe that enough indications are given to convince the reader that the argument used in the PCF case can be easily adapted to IA.

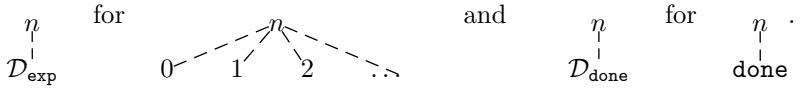
Computation DAG

In PCF, arenas have a single initial move, therefore they can be regarded as trees. In IA, on the other hand, the base type `var` is represented by the infinite product of games $\mathtt{com}^{\mathbb{N}} \times \mathtt{exp}$ which has an infinite number of initial moves. In order to preserve the relationship established between arenas and computation trees, we need to accommodate the definition of computation tree to reflect this property. The consequence is that in IA, "computation trees" become "computation directed acyclic graphs (DAG)": a computation DAG may have (possibly infinitely) many roots and two nodes of a given level can share children at the next level.

We use the notations $\mathcal{D}_{\mathtt{exp}} = \mathbb{N}$ and $\mathcal{D}_{\mathtt{com}} = \{\mathtt{done}\}$ to denote the set of value leaves of type `exp` and `com` respectively. There are two types of value-leaves in the computation DAG: the value-leaf `done` of type `com` and the value-leaves labelled in $\mathcal{D}_{\mathtt{exp}}$ of type `exp`.

Let n be a node. If $\kappa(n)$ is of type $(A_1, \ldots A_n, B)$, we call B the *return type of* n. The set of value-leaves of a node n is given by $\mathcal{D}_{\mathtt{exp}}$ if the return type of n is `exp`, by $\mathcal{D}_{\mathtt{com}}$ if its return type is `com`, and by $\mathcal{D}_{\mathtt{exp}} \cup \{\mathtt{done}\}$ if its return type is `var`.

Table 4.1 shows the computation DAG for each construct of IA. The value-leaves are represented in the DAGs using the following abbreviations:



A term of type `var` has a computation DAG with an infinite number of root λ-nodes. Suppose that M is a term of type `var`, then the computation DAG for $\lambda\overline{\xi}.M$ is obtained by relabelling the

root λ-nodes λ^r, λ^{w_0}, λ^{w_1}, λ^{w_2}, ... into $\lambda^r\overline{\xi}$, $\lambda^{w_0}\overline{\xi}$, $\lambda^{w_1}\overline{\xi}$, $\lambda^{w_2}\overline{\xi}$, ... For a term M of type $\mathtt{exp}$ or $\mathtt{com}$, the computation DAG for $\lambda\overline{\xi}.M$ is computed in the same way as in the Safe λ-Calculus.

Traversals

Let p be a node and suppose that its ith child n has the return type $\mathtt{var}$. Then n is in fact constituted of several λ-nodes : $\lambda^r\overline{\xi}$, $\lambda^{w_0}\overline{\xi}$, ... From p's point of view, these nodes are referenced as follows: $i.r$ refers to $\lambda^r\overline{\xi}$ and $i.w_k$ refers to $\lambda^{w_k}\overline{\xi}$ for $k \in \omega$.

- *The application rule*

  There are two rules ($\mathrm{app}_{\mathtt{exp}}$) and ($\mathrm{app}_{\mathtt{com}}$) corresponding to traversals ending with an @-node of return type $\mathtt{exp}$ and $\mathtt{com}$ respectively. These rules are identical to the rule $\mathtt{exp}$ of section 3.1.4.

  The application rule for @-nodes with return type $\mathtt{var}$ is:

$$(\mathrm{app}_{\mathtt{var}})\frac{t \cdot \lambda^k\overline{\xi} \overset{0}{\leftarrow} @ \in \mathcal{T}rav}{t \cdot \lambda^k\overline{\xi} \overset{0}{\leftarrow} @ \overset{0.k}{\leftarrow} \lambda^k\overline{\eta} \in \mathcal{T}rav}\ k \in \{r, w_0, w_1, \ldots\}$$

- *Input-variable rules*

  There are two rules (InputVar$^{\mathtt{exp}}$) and (InputVar$^{\mathtt{com}}$) which are the counterparts of rule (InputVar^0) of section 3.1.4 and are defined identically.

  Let x be an input-variable of type $\mathtt{var}$:

$$(\mathrm{InputVar}^{\mathtt{var}})\frac{t \cdot \lambda^r\overline{\xi} \cdot x \in \mathcal{T}rav}{t \cdot \lambda^r\overline{\xi} \cdot x \cdot v_x \in \mathcal{T}rav} \qquad (\mathrm{InputVar}'^{\mathtt{var}})\frac{t \cdot \lambda^{w_i}\overline{\xi} \cdot x \in \mathcal{T}rav}{t \cdot \lambda^{w_i}\overline{\xi} \cdot x \cdot \mathtt{done}_x \in \mathcal{T}rav}$$

- *IA constants rules*

  The rules for $\mathtt{new}$ are purely structural, they are defined the same way as the rules ($\mathrm{app}_{\mathtt{exp}}$), ($\mathrm{app}_{\mathtt{com}}$) and ($\mathrm{app}_{\mathtt{done}}$).

  The rules for $\mathtt{deref}$ are:

$$(\mathrm{deref})\frac{t \cdot \mathtt{deref} \in \mathcal{T}rav}{t \cdot \mathtt{deref} \overset{1.r}{\leftarrow} n \in \mathcal{T}rav} \qquad (\mathrm{deref'})\frac{t \cdot \mathtt{deref} \cdot n \cdot t_2 \cdot v_n \in \mathcal{T}rav}{t \cdot \mathtt{deref} \cdot n \cdot t_2 \cdot v_n \cdot v_{\mathtt{deref}} \in \mathcal{T}rav}$$

  The rules for $\mathtt{assign}$ are:

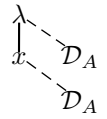
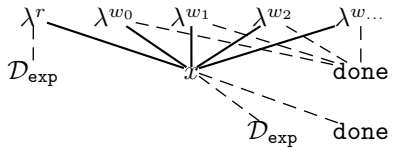
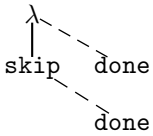
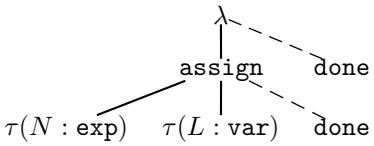
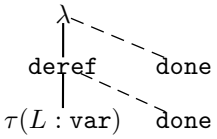
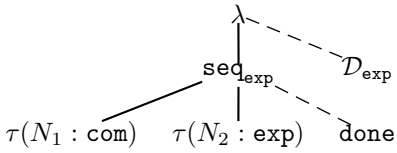
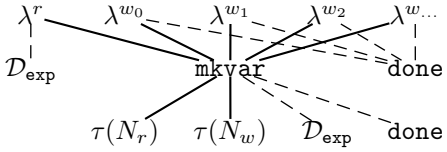
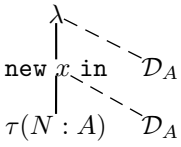
$$(\mathrm{assign})\frac{t \cdot \mathtt{assign} \in \mathcal{T}rav}{t \cdot \mathtt{assign} \overset{1}{\leftarrow} n \in \mathcal{T}rav} \qquad (\mathrm{assign'})\frac{t \cdot \mathtt{assign} \cdot n \cdot t_2 \cdot v_n \in \mathcal{T}rav}{t \cdot \mathtt{assign} \overset{2.w_n}{\leftarrow} n \cdot t_2 \cdot v_n \cdot m \in \mathcal{T}rav}$$

$$(\mathrm{assign''})\frac{t \cdot \mathtt{assign} \overset{2.w_k}{\leftarrow} t_2 \cdot m \cdot t_3 \cdot \mathtt{done}_m \in \mathcal{T}rav}{t \cdot \mathtt{assign} \cdot t_2 \cdot m \cdot t_3 \cdot \mathtt{done}_m \cdot \mathtt{done}_{\mathtt{assign}} \in \mathcal{T}rav}$$

  The rules for $\mathtt{seq}_{\mathtt{exp}}$ are:

$$(\mathrm{seq})\frac{t \cdot \mathtt{seq} \in \mathcal{T}rav}{t \cdot \mathtt{seq} \overset{1}{\leftarrow} n \in \mathcal{T}rav} \qquad (\mathrm{seq'})\frac{t \cdot \mathtt{seq} \cdot n \cdot t_2 \cdot v_n \in \mathcal{T}rav}{t \cdot \mathtt{seq} \overset{2}{\leftarrow} n \cdot t_2 \cdot v_n \cdot m \in \mathcal{T}rav}$$

$$(\mathrm{seq''})\frac{t \cdot \mathtt{seq} \overset{2}{\leftarrow} t_2 \cdot m \cdot t_3 \cdot v_m \in \mathcal{T}rav}{t \cdot \mathtt{seq} \cdot t_2 \cdot m \cdot t_3 \cdot v_m \cdot v_{\mathtt{seq}} \in \mathcal{T}rav}$$

M	$\tau(M)$
$\mathtt{x} : A \in \{\mathtt{com}, \mathtt{exp}\}$	
$\mathtt{x} : \mathtt{var}$	
$\mathtt{skip} : \mathtt{com}$	
$\mathtt{assign}\ L\ N : \mathtt{com}$	
$\mathtt{deref}\ L : \mathtt{exp}$	
$\mathtt{seq}_{\mathtt{exp}}\ N_1\ N_2 : \mathtt{com}$	
$\mathtt{mkvar}\ N_w\ N_r : \mathtt{var}$	
$\mathtt{new}\ x\ \mathtt{in}\ N : A \in \{\mathtt{com}, \mathtt{exp}\}$	

Tab. 4.1: Computation DAGs for the constructs of IA.

The rules for `mkvar` are:

$$(\text{mkvar}_r)\frac{t\cdot\lambda^r\overline{\xi}\cdot\texttt{mkvar}\in\mathcal{T}rav}{t\cdot\lambda^r\overline{\xi}\cdot\texttt{mkvar}\cdot\overset{1}{\curvearrowleft}n\in\mathcal{T}rav}\qquad(\text{mkvar}'_r)\frac{t\cdot\texttt{mkvar}\cdot n\cdot t_2\cdot v_n\in\mathcal{T}rav}{t\cdot\texttt{mkvar}\cdot n\cdot t_2\cdot v_n\cdot v_{\texttt{mkvar}}\in\mathcal{T}rav}$$

$$(\text{mkvar}_w)\frac{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\in\mathcal{T}rav}{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\cdot\overset{2}{\curvearrowleft}n\in\mathcal{T}rav}$$

$$(\text{mkvar}''_w)\frac{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\cdot n\cdot t_2\cdot\texttt{done}_n\in\mathcal{T}rav}{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\cdot n\cdot t_2\cdot\texttt{done}_n\cdot\texttt{done}_{\texttt{mkvar}}\in\mathcal{T}rav}$$

These four rules are not sufficient to model the constant `mkvar`. Indeed, consider the term `assign` (`mkvar` $(\lambda x.M)N$)7. The rule (mkvar''_w) permits to traverse the node `mkvar` and to go on by traversing the computation tree of $\lambda x.M$. The problem is that when traversing $\tau(M)$, if we reach a variable x, we are not able to relate x to the value 7 that is assigned to the variable.

To overcome this problem, we need to define traversal rules for variable in such a way that a variable node bound by the second child of a `mkvar`-node is treated differently from other variables.

- *Variable rules* Let x be a non input-variable node and let b be the binder of x. b is either a "`new` x `in`"-node or a λ-node.

  – Consider the case where b is a λ-node. Take $b=\lambda\overline{x}$. In IA, the only constant nodes of order greater than 1 is `mkvar`, therefore there are two cases: $\lambda\overline{x}$ is either the child of a node in $N_@\cup N_{var}$ or it is the second child of a `mkvar`-node.

  To handle the first case, we define a rule similar to the (Var) rule of section 3.1.4 with some modification to take into account variables x of type `var`(in which case x has multiple parent λ-nodes). We do not give the details here but it is easy to see how to redefine this rule.

  To handle the case where $\lambda\overline{x}$ is the child of a `mkvar`-node, we define the following rule:

$$(\text{Var}_{\texttt{mkvar}})\frac{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\cdot\lambda\overline{x}\cdot t_2\cdot x\in\mathcal{T}rav}{t\cdot\lambda^{w_k}\overline{\xi}\cdot\texttt{mkvar}\cdot\lambda\overline{x}\cdot t_2\cdot x\cdot k_x\in\mathcal{T}rav}$$

  – The case $b=$ `new` x `in` is handle by the following rules.

  We call *overwrite of* x any sequence of nodes of the form $\lambda^{w_k}\overline{\xi}\cdot x$ for some $k\in\mathcal{D}_{\texttt{exp}}$ and node $\lambda^{w_k}\overline{\xi}$ parent of x.

$$(\text{Var}_w)\frac{t\cdot\lambda^{w_k}\overline{\xi}\cdot x\in\mathcal{T}rav}{t\cdot\lambda^{w_k}\overline{\xi}\cdot x\cdot\texttt{done}_x\in\mathcal{T}rav},$$

$$(\text{Var}_r)\frac{t_1\cdot\texttt{new}\ x\ \texttt{in}\cdot t_2\cdot\lambda^r\overline{\xi}\cdot x\in\mathcal{T}rav}{t_1\cdot\texttt{new}\ x\ \texttt{in}\cdot t_2\cdot\lambda^r\overline{\xi}\cdot x\cdot 0_x\in\mathcal{T}rav}\ \text{if } t_2 \text{ contains no overwrite of } x,$$

$$(\text{Var'}_r)\frac{t_1\cdot\texttt{new}\ x\ \texttt{in}\cdot t_2\cdot\lambda^r\overline{\xi}\cdot x\in\mathcal{T}rav}{t_1\cdot\texttt{new}\ x\ \texttt{in}\cdot t_2\cdot\lambda^r\overline{\xi}\cdot x\cdot k_x\in\mathcal{T}rav}\ \text{if } \lambda^{w_k}\cdot x \text{ is the last overwrite of } x \text{ in } t_2.$$

Game semantics correspondence

The properties that we proved for computation trees and traversals of the Safe λ-Calculus with constants can easily be lifted to computation DAGs of IA. In particular:

- constant traversal rules are well-behaved;

- P-view of traversals are paths in the computation DAG;
- the P-view of the reduction of a traversal is the reduction of the P-view, and the O-view of a traversal is the O-view of its reduction (lemma 3.1.20);
- there is a mapping from vertices of the computation DAG to moves in the interaction game semantics;
- there is a correspondence between traversals of the computation tree and plays in interaction game semantics;
- consequently, there is a correspondence between the standard game semantics and the set of justified sequences of nodes $\mathcal{T}rav^{\restriction r}$.

Game-semantic characterisation of safe terms

Clearly, the computation DAG of a safe term is incrementally-bound. By using the correspondence between traversals and plays, it is easy to prove that incrementally-bound computation trees are denoted by incrementally-justified strategies. Consequently, by lemma 4.1.2, the pointers in the game semantics of Safe IA terms are superfluous.

Since the game denotation of an IA term is fully determined by the set of complete plays, this pointer economy suggests that the game denotation of a Safe IA can be represented in a compact way. This raises the question of the decidability of observational equivalence for Safe IA.

Chapter 5

FURTHER POSSIBLE DEVELOPMENTS

In the previous chapter, we have given an account of the game semantics of Safe λ-Calculus. However the nature of this calculus is still not well known. We propose the following possible roadmap for further research:

1. prove or disprove that observational equivalence is decidable for Safe IA;
2. find a categorical interpretation of the Safe λ-Calculus;
3. study the proof theory obtained by the Curry-Howard isomorphism and determine whether it has nice properties that can be helpful in theorem proving;
4. in Leivant and Marion [1993], the λ-calculus is used to give several characterisations of the complexity class P. We would like to investigate whether, by following similar techniques, we can obtain a characterisation of a different complexity class using the Safe λ-Calculus.

More generally, we would like to study the class of languages for which pointers are uniquely recoverable. We name this class PUR for "Pointer Uniquely Recoverable".

We proved that Safe λ-Calculus is a PUR-language. Another example is the Serially Re-entrant Idealized Algol (SRIA) proposed by Abramsky in Abramsky [2001b]. This language allows multiple occurrences or uses of arguments, as long as they do not overlap in time. In the game semantics denotation of a SRIA term there is at most one pending occurrence of a question at any time. Each move has therefore a unique justifier and consequently justification pointers may be ignored. Safe IA is not a sublanguage of SRIA. One reason for this is that none of the two Kierstead terms $\lambda f.f(\lambda x.f(\lambda y.y))$ and $\lambda f.f(\lambda x.f(\lambda y.x))$ are Serially Re-entrant whereas the first one is safe. Conversely, SRIA is not a sublanguage of Safe IA since the term $\lambda fg.f(\lambda x.g(\lambda y.x))$ where $f, g : ((o, o), o)$ belongs to SRIA but not to Safe IA. SRIA and Safe IA are therefore two different examples of languages with pointer-less game semantics.

Finitary IA_2 is also an example of PUR-language for which observational equivalence is decidable. As we indicated in the first chapter, decidability of observational equivalence is a very appealing property which has immediate applications in the domain of program verification. Intuitively, PUR-languages seem to be good candidates of languages for which observational equivalence is decidable. It would be interesting to discover classes of PUR languages having this appealing property.

Another possible way to generate PUR-languages may be to constrain the types of an existing language. In Joly [2001], a notion of "complexity" is defined for λ-terms. It is proved that a type T can be generated from a finite set of combinators if and only if there is a constant bounding the complexity of every closed normal λ-term of type T; consequently, the only inhabited finitely generated types are the type of rank ≤ 2 and the types $(A_1, A_2, \ldots, A_n, o)$ such that for all $i = 1..n$: $A_i = o$, $A_i = o \to o$ or $A_i = o^k \to o \to o$. We know that imposing the first of these two type restrictions to Finitary IA leads to a PUR language. Is it also the case when imposing the second type restriction?

BIBLIOGRAPHY

Samson Abramsky. Algorithmic game semantics: a tutorial introduction. 2001a.

Samson Abramsky. Semantics via game theory. In *Marktoberdorf International Summer School*, 2001b. Lecture slides.

Samson Abramsky and Radha Jagadeesan. A game semantics for generic polymorphism. *Ann. Pure Appl. Logic*, 133(1-3):3–37, 2005.

Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science (FST-TCS'92)*, pages 291–301, New Delhi, India, 1992. URL `citeseer.ist.psu.edu/article/abramsky94games.html`.

Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-like languages*. Birkhaüser, 1997.

Samson Abramsky and Guy McCusker. Call-by-value games. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic: 11th International Workshop Proceedings*. Springer-Verlag, 1998a. URL `citeseer.ist.psu.edu/abramsky97callbyvalue.html`.

Samson Abramsky and Guy McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227(1–2):3–42, 1999. URL `citeseer.ist.psu.edu/abramsky98full.html`.

Samson Abramsky and Guy McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School*. Springer-Verlag, 1998b. Lecture notes.

Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994. URL `citeseer.ist.psu.edu/abramsky95full.html`.

Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *LICS*, pages 334–344, 1998.

Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. Safety is not a restriction at level 2 for string languges. Technical report, University of Oxford, 2004.

Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. Safety is not a restriction at level 2 for string languages. In Sassone [2005], pages 490–504. ISBN 3-540-25388-2.

Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the lambda-calculus. In *LICS*, pages 426–436. IEEE Computer Society, 1994.

Gérard Berry. Stable models of typed lambda-calculi. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 72–89, London, UK, 1978. Springer-Verlag. ISBN 3-540-08860-1.

Gérard Berry. *Modèles Complément Adéquats et Stable des Lambda-calculs typés.* Phd thesis, Université Paris VII, 1979.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999. URL `citeseer.ist.psu.edu/article/biere99symbolic.html`.

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

Roy Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993. ISBN 0521450926.

W. Damm. The IO- and OI-hierarchy. *TCS*, 20:95–207, 1982.

Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of girard's execution formula). In Moshe Vardi, editor, *Proceedings of the Eighth Annual IEEE Symp. on Logic in Computer Science, LICS 1993*, pages 296–306. IEEE Computer Society Press, June 1993.

Jolie G. de Miranda. *Structures generated by higher-order grammars and the safety constraint.* Forthcoming, University of Oxford, 2006.

Aleksandar Dimovski, Dan R. Ghica, and Ranko Lazic. Data-abstraction refinement: A game semantic approach. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2005. ISBN 3-540-28584-9.

Dan R. Ghica and Guy 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. URL `citeseer.ist.psu.edu/ghica00reasoning.html`.

Will Greenland. *Game Semantics for Region Analysis.* PhD thesis, University of Oxford, 2004.

Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs and Lenore Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Lecture Notes in Computer Science, pages 191–205, Edinburgh, Scotland, April 2005. Springer-Verlag.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/357980.358021.

Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *Theoretical Computer Science*, 221(1–2):393–456, 1999. URL `citeseer.ist.psu.edu/article/honda97game.html`.

Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1): 27–57, 1975.

Gérard P. Huet. *Résolution d'équations dans des langages d'ordre 1,2,...,ω.* Thèse de doctorat es sciences mathématiques, Université Paris VII, Septembre 1976.

J. M. E. Hyland and C.-H. L. Ong. Fair games and full completeness for Multiplicative Linear Logic without the MIX-rule. preprint, 1993.

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. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.2000.2917.

D. C. Jensen and Tomasz Pietrzykowski. Mechanizing *mega*-order type theory through unification. *Theor. Comput. Sci.*, 3(2):123–171, 1976.

Thierry Joly. The finitely generated types of the lambda-calculus. In *TLCA*, pages 240–252, 2001.

N.D. Jones and N. Bohr. Termination of the untyped lambda calculus. 2004.

T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.

John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-343-4. doi: http://doi.acm.org/10.1145/96709.96711.

C.S. Lee, N.D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. 2001.

Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 1993. ISBN 3-540-56517-5.

Paul Lorenzen. Ein dialogisches konstruktivitätskriterium. In Warsaw PWN, editor, *Infinitistic Methods.*, pages 193–200, 1961.

Guy McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. PhD thesis, Imperial College, 1996.

Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003. ISBN 3-540-40524-0.

Andrzej S. Murawski. Games for complexity second-order call-by-name programs. *Theoretical Computer Science*, 2005. special issue: Game Theory meets Computer Science, accepted for publication.

Andrzej S. Murawski. On program equivalence in languages with ground-type references, 2003. URL `citeseer.ist.psu.edu/murawski03program.html`.

Andrzej S. Murawski and Igor Walukiewicz. Third-order idealized algol with iteration is decidable. In Sassone [2005], pages 202–218. ISBN 3-540-25388-2. URL `"http://www.labri.fr/publications/l3a/2005/MW05"`.

Andrzej S. Murawski, C.-H. Luke Ong, and Igor Walukiewicz. Idealized algol with ground recursion, and dpda equivalence. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 917–929. Springer, 2005. ISBN 3-540-27580-0.

N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

Hanno Nickau. Hereditarily sequential functionals. In Anil Nerode and Yu. V. Matiyasevich, editors, *Proc. Symp. Logical Foundations of Computer Science: Logic at St. Petersburg*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer-Verlag, 1994.

C.-H. L. Ong. Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 22-25 July 200, Copenhagen Denmark*, pages 245–256. Computer Society Press, 2002.

C.-H. L. Ong. Safe lambda calculus: Some questions. Note on the safe lambda calculus., December 2005.

C.-H. L. Ong. On model-checking trees generating by higher-order recursion schemes (technical report). URL `http://users.comlab.ox.ac.uk/luke.ong/publications/ntrees.pdf`. Preprint, 42 pp, 2006a.

C.-H. L. Ong. On model-checking trees generating by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science*. Computer Society Press, 2006b. Extended abstract.

Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3): 225–255, 1977.

Princeton University. zchaff. `http://www.princeton.edu/~chaff/zchaff.html`.

John 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.

Vladimiro Sassone, editor. *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3441 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-25388-2.

Dana S. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theor. Comput. Sci.*, 121 (1-2):411–440, 1993. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(93)90095-B.

Dana S. Scott. A theory of computable function of higher type. Unpublished seminar notes, University of Oxford, 1969.

Géraud Sénizergues. L(A)=L(B)? decidability results from complete formal systems. *Theor. Comput. Sci.*, 251(1-2):1–166, 2001.

Damien Sereni. Simply typed λ-calculus and sct. 2005.

Colin Stirling. Deciding dpda equivalence is primitive recursive. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 821–832, London, UK, 2002. Springer-Verlag. ISBN 3-540-43864-5.