Synchronous Games, Simulations and λ-calculus

Pierre Hyvernat[1,2]

[1] Institut mathématique de Luminy, Marseille, France
[2] Chalmers Institute of Technology, Göteborg, Sweden
`hyvernat@iml.univ-mrs.fr`

Abstract. We refine a model for linear logic based on two well-known ingredients: games and simulations. We have already shown that usual simulation relations form a sound notion of morphism between games; and that we can interpret all linear logic in this way. One particularly interesting point is that we interpret multiplicative connectives by synchronous operations on games.
We refine this work by giving computational contents to our simulation relations. To achieve that, we need to restrict to intuitionistic linear logic. This allows to work in a constructive setting, thus keeping a computational content to the proofs.
We then extend it by showing how to interpret some of the additional structure of the exponentials.

To be more precise, we first give a model for the typed λ-calculus; and then give a model for the differential λ-calculus of Ehrhard and Regnier. Both this models are proved correct constructively.

Introduction

Transition systems and simulation relations are well known tools in computer science. More recent is the use of games to give models for different programming languages [1, 9, 2], or as an interesting tool for the study of other programming notions [3]. We have devised in [12] a denotational model of linear logic based on those two ideas. Basically, a formula was interpreted by an alternating transition system (called an *interaction system*) and a proof was interpreted by a *safety property* for this interaction system. Those concepts which were primarily developed to model imperative programming and interfaces turned out to be a rather interesting games model: a formula is interpreted by a game (the interaction systems), and a proof by a "non-loosing strategy" (the safety property).

Part of the interest is that the notion of safety property is very simple: it is only a subset of the set of states. However, in terms of games, the associated strategy (whose existence is guaranteed by the condition satisfied by the subset of states) is usually not computable. We will show that it is possible to overcome this problem by restricting to intuitionistic linear logic. More precisely, we will model typed λ-calculus (seen as a subsystem of intuitionistic linear logic) within a constructive setting. The model for full intuitionistic linear logic (ILL) can easily be derived the present work and the additive connectives defined in [12].

The structure of safety properties is in fact richer than the structure of λ-terms. In particular, safety properties are closed under unions. Since there is no sound notion of "logical sum" of proofs, this doesn't reflect a logical property. However, it is important in programming since it can be used to interpret non-determinism. The differential λ-calculus of Ehrhard and Regnier ([6]) is an extension to the λ-calculus, which has a notion of non deterministic sum. We show how to interpret this additional structure.

1 Interaction Systems

1.1 The Category of Interaction Systems

We briefly recall the important definitions. For more motivations, we refer to [8] and [12].

Definition 1. *Let S be a set (of* states*); an* interaction system *on S is given by the following data:*

- *for each $s \in S$, a set $A(s)$ of* possible actions;
- *for each $a \in A(s)$, a set $D(s,a)$ of* possible reactions *to a;*
- *for each $d \in D(s,a)$, a* new state *$n(s,a,d) \in S$.*

We usually write $s[a/d]$ instead of $n(s,a,d)$.

Following standard practise within computer science, we distinguish the two "characters" by calling them the Angel (choosing actions, hence the A) and the Demon (choosing reactions, hence the D). Depending on the authors' background, other names could be Player and Opponent, Eloise and Abelard, Alice and Bob, Master and Slave, Client and Server, System and Environment, etc.

One of the original goals for interaction systems (Hancock) was to represent real-life programming interfaces. Here is for example the interface of a stack of booleans:

- $S = \mathsf{List}(\mathbf{B})$;
- $A(\_) = \{\mathsf{Push}(b) \mid b \in \mathbf{B}\} \cup \{\mathsf{Pop}\}$;
- $\begin{cases} D(\_, \mathsf{Push}(b)) = \{*\} \\ D([], \mathsf{Pop}) = \{\mathsf{error}\} \\ D(b : s, \mathsf{Pop}) = \{*\} \end{cases}$
- $\begin{cases} n(s, \mathsf{Push}(b)) = b : s \\ n([], \mathsf{Pop}, \mathsf{error}) = [] \\ n(b : s, \mathsf{Pop}) = s \end{cases}$

This gives in full details the specification of the stack interface. This is more precise than classical interfaces which are usually given by a collection of types: compare with this poor description of stacks:

- $\mathsf{Pop} : \mathbf{B}$
- $\mathsf{Push} : \mathbf{B} \to ()$

which doesn't specify what the command actually *do*; but only tells how they can be used.

The notion of morphism between such interaction systems is an extension of the usual notion of simulation relation:

Definition 2. *If w_1 and w_2 are two interaction systems on S_1 and S_2 respectively; a relation $r \subseteq S_1 \times S_2$ is called a simulation if:*

$$
\begin{array}{lll}
(s_1, s_2) \in r & \Rightarrow & \big(\forall a_1 \in A_1(s_1)\big) \\
 & & \big(\exists a_2 \in A_2(s_2)\big) \\
 & & \big(\forall d_2 \in D_2(s_2, a_2)\big) \\
 & & \big(\exists d_1 \in D_1(s_1, a_1)\big) \\
 & & \qquad \big(s_1[a_1/d_1], s_2[a_2/d_2]\big) \in r \ .
\end{array}
$$

This definition is very similar to the usual definition of simulation relation between labelled transition systems, but adds one layer of quantifiers to deal with reactions. That $(s_1, s_2) \in r$ means that "s_2 simulates s_1". By extension, if a_2 is a witness to the first existential quantifier, we say that "a_2 simulates a_1". Note that the empty relation is *always* a simulation. In practise, to prevent this degenerate case, we would add a notion of initial state(s) and require that initial states are related through the simulation.

To continue on the previous example, programming a stack interface amounts to implementing the stack commands using a lower level interface (arrays and pointer for examples). If we interpret the quantifiers constructively, this amounts to providing a (constructive) proof that a non-empty relation is a simulation from this lower level interaction system to stacks. (See [8] for a more detailed description of programming in terms of interaction systems.)

Recall that the composition of two relations is given by:

$$(s_1, s_3) \in r_2 \cdot r_1 \Leftrightarrow (\exists s_2)\ (s_1, s_2) \in r_1 \text{ and } (s_2, s_3) \in r_2$$

It should be obvious that the composition of two simulations is a simulation and that the equality relation is a simulation from any w to itself. Thus, we can put:

Definition 3. *We call* **Int** *the category of interaction systems with simulations.*

Note that everything has a computational content: the composition of two simulations is just given by the composition of the two "algorithms" simulating w_3 by w_2 and w_2 by w_1; and that the algorithm for the identity from w to w is simply the "copycat" strategy.

1.2 Notation

Before diving in the structure of interaction systems, let's detail some of the notation.

– An element of the indexed cartesian product $\prod_{a \in A} D(a)$ is given by a function f taking any $a \in A$ to an $f(a)$ in $D(a)$. When the set $D(a)$ doesn't depend on a, it amounts to a function $f : A \rightarrow D$.

- An element of the indexed disjoint sum $\sum_{a\in A} D(a)$ is given by a pair (a,d) where $a \in A$ and $d \in D(a)$. When the set $D(a)$ doesn't depend on a, this is simply the cartesian product $A \times D$.
- We write $\mathsf{List}(S)$ for the set of "lists" over set S. A list is simply a tuple $(s_1, s_2, \dots s_n)$ of elements of S. The empty list is denoted $()$.
- The collection $\mathcal{M}_f(S)$ of finite multisets over S is the quotient of $\mathsf{List}(S)$ by permutations. We write $[s_1, \dots s_n]$ for the equivalence class containing $(s_1, \dots s_n)$. We write "$+$" for the sum of multisets. It simply corresponds to concatenation on lists.

Concerning the product and sum operators, it should be noted that they have a computational content if one works in a constructive setting: an element of $\prod_{a\in A} D(a)$ is an algorithm with input $a \in A$ and output $f(a) \in D(a)$; and an element of $\sum_{a\in A} D(a)$ is simply a pair as above. This is in fact the basis of dependent type theory frameworks like Martin-Löf's type theory or the calculus of construction.

Remark: even if it was an important motivation for this work, we do not insist too much on the "constructive mathematics" part. Readers familiar with constructive frameworks should easily see that everything makes computational sense; and classical readers can skip the comments about computational content.

1.3 Constructions

We now define the connectives of multiplicative exponential linear logic. With those, making **Int** into a denotational model of intuitionistic multiplicative exponential linear logic more or less amounts to showing that it is symmetric monoidal closed, with a well behaved comonad.

Constant. A very simple, yet important interaction system is "skip", the interaction system without interaction. Following the linear logic convention, we call it $\perp$:

Definition 4. *Define $\perp$ (or skip) to be the following interaction system on the Singleton set $\{*\}$:*

$$\begin{aligned} A_\perp(*) &= \{*\} \\ D_\perp(*,*) &= \{*\} \\ n_\perp(*,*,*) &= \{*\} \ . \end{aligned}$$

Depending on the context, this interaction system is also denoted by $\mathbf{1}$.

Note that it is very different from the two following interaction systems (on the same set of states) which respectively deadlock the Angel and the Demon:

$$\begin{aligned} A_a(*) &= \emptyset & A_d(*) &= \{*\} \\ D_a(*,\_) &= \_ & D_d(*,*) &= \emptyset \\ n_a(*,\_,\_) &= \_ & n_d(*,*,\_) &= \_ \quad . \end{aligned}$$

Those two systems play an important rôle in the general theory of interaction systems (the first one is usually called abort, while the second one is usually called magic) but they do not appear in the model presented below.

Synchronous Product. There is an obvious product construction reminiscent of the synchronous product found in SCCS (synchronous calculus of communicating systems, [13]):

Definition 5. *Suppose w_1 and w_2 are interaction systems on S_1 and S_2. Define the interaction system $w_1 \otimes w_2$ on $S_1 \times S_2$ as follows:*

$$\begin{array}{lcl} A_{w_1\otimes w_2}\big((s_1,s_2)\big) & = & A_1(s_1) \times A_2(s_2) \\ D_{w_1\otimes w_2}\big((s_1,s_2),(a_1,a_2)\big) & = & D_1(s_1,a_1) \times D_2(s_2,a_2) \\ n_{w_1\otimes w_2}\big((s_1,s_2),(a_1,a_2),(d_1,d_2)\big) & = & \big(s_1[a_1/d_1], s_2[a_2/d_2]\big)\ . \end{array}$$

This is the *synchronous parallel composition* of w_1 and w_2: the Angel and the Demon exchange pairs of actions/reactions.

For any sensible notion of morphism, skip should be a neutral element for this product. It is indeed the case, for the following reason: the components of $w \otimes$ skip and w are isomorphic by dropping the second (trivial) coordinate:

$$\begin{array}{lll} w \otimes \mathbf{1} & & w \\ S \times \{*\} & & S \\ A\big((s,*)\big) & = A(s) \times \{*\} & A(s) \\ D\big((s,*),(a,*)\big) & = D(s,a) \times \{*\} & D(s,a) \\ n\big((s,*),(a,*),(d,*)\big) & = \big(s[a/d],*\big) & s[a/d] \end{array}$$

This implies trivially that $\{((s,*),s) \mid s \in S\}$ is an isomorphism. For similar reasons, this product is transitive and commutative.

Lemma 1. *"$\_ \otimes \_$" is a commutative tensor product in the category* **Int**. *Its action on morphisms is given by:*

$$\big((s_1,s_1'),(s_2,s_2')\big) \in r \otimes r' \Leftrightarrow \left\{ \begin{array}{r} (s_1,s_2) \in r \\ \textit{and } (s_1',s_2') \in r' \end{array} \right.$$

Checking that $r \otimes r'$ is indeed a simulation is easy.

Linear Arrow. The definition of the interaction system $w_1 \multimap w_2$ is not as obvious as the definition of the tensor ($\otimes$):

Definition 6. *If w_1 and w_2 are interaction systems on S_1 and S_2, define the interaction system $w_1 \multimap w_2$ on $S_1 \times S_2$ as follows:*

$$\begin{array}{l} A\big((s_1,s_2)\big) = \displaystyle\sum_{f \in A_1(s_1) \to A_2(S_2)} \ \prod_{a_1 \in A_1(s_1)} D_2\big(s_2, f(a_1)\big) \to D_1(s_1,a_1) \\ D\big((s_1,s_2),(f,G)\big) = \displaystyle\sum_{a_1 \in A_1(s_1)} D_2\big(s_2, f(a_1)\big) \\ n\big((s_1,s_2),(f,G),(a_1,d_2)\big) = \big(s_1[a_1/G_{a_1}(d_2)]\,,\ s_2[f(a_1)/d_2]\big)\ . \end{array}$$

It may seem difficult to get some intuition about this interaction system; but it is *a posteriori* quite natural: (see Proposition 1)

– An action in state (s_1, s_2) is given by:
  (1) a function f (the index for the element of the disjoint sum) translating actions from s_1 into actions from s_2;
  (2) for any action a_1, a function G_{a_1} translating reactions to $f(a_1)$ into reactions to a_1.
– A reaction to such a "translating mechanism" is given by:
  (1) an action a_1 in $A_1(s_1)$ (which we want to simulate);
  (2) and a reaction d_2 in $D_2(s_2, f(a_1))$ (which we want to translate back).
– Given such a reaction, we can simulate a_1 by $a_2 \in A_2(s_2)$ obtained by applying f to a_1; and translate back d_2 into $d_1 \in D_1(s_1, a_1)$ by applying G_{a_1} to d_2. The next state is just the pair of states $s_1[a_1/d_1]$ and $s_2[a_2/d_2]$.

It thus looks like the interaction system $w_1 \multimap w_2$ is related to simulations from w_1 to w_2. It is indeed the case:

Proposition 1. *In* **Int***, "$\_ \otimes \_$" is left adjoint to "$\_ \multimap \_$".*

Proof. The proof is not really difficult, but is quite painful to write (or read). Here is an attempt.
Note that the following form of the axiom of choice is constructively valid:[3]

$$\mathsf{AC}: \quad \big(\forall a \in A\big)\big(\exists d \in D(a)\big)\varphi(a,d) \Leftrightarrow \big(\exists f \in \textstyle\prod_{a\in A} D(a)\big)\big(\forall a \in A\big)\varphi\big(a, f(a)\big)$$

When the domain $D(a)$ for the existential quantifier doesn't depend on $a \in A$, we can simplify it into:

$$\mathsf{AC}: \quad \big(\forall a \in A\big)\big(\exists d \in D\big)\varphi(a,d) \Leftrightarrow \big(\exists f \in A \to D\big)\big(\forall a \in A\big)\varphi\big(a, f(a)\big)$$

In the sequel, the part of the formula being manipulated will be written in bold. That r is a simulation from $w_1 \otimes w_2$ to w_3 takes the form[4]

$$\begin{aligned}(s_1,s_2,s_3) \in r \Rightarrow\ & \big(\forall a_1 \in A_1(s_1)\big)\big(\forall \boldsymbol{a_2} \in \boldsymbol{A_2(s_2)}\big)\\ & \big(\exists \boldsymbol{a_3} \in \boldsymbol{A_3(s_3)}\big)\\ & \big(\forall d_3 \in D_3(s_3, \boldsymbol{a_3})\big)\\ & \big(\exists d_1 \in D_1(s_1,a_1)\big)\big(\exists d_2 \in D_2(s_2,a_2)\big)\\ & \quad \big(s_1[a_1/d_1], s_2[a_2/d_2], s_3[\boldsymbol{a_3}/d_3]\big) \in r\end{aligned}$$

Using AC on the $\forall a_2 \exists a_3$, we obtain:

$$\begin{aligned}(s_1,s_2,s_3) \in r \Rightarrow\ & \big(\forall a_1 \in A_1(s_1)\big)\\ & \big(\exists f \in A_2(s_2) \to A_3(s_3)\big)\\ & \big(\forall a_2 \in A_2(s_2)\big)\big(\forall \boldsymbol{d_3} \in \boldsymbol{D_3(s_3, f(a_2))}\big)\\ & \big(\exists d_1 \in D_1(s_1,a_1)\big)\big(\exists \boldsymbol{d_2} \in \boldsymbol{D_2(s_2,a_2)}\big)\\ & \quad \big(s_1[a_1/d_1], s_2[a_2/\boldsymbol{d_2}], s_3[f(a_2)/d_3]\big) \in r\end{aligned}$$

[3] This form of the axiom of choice is provable in Martin-Löf's type theory or in the calculus of construction...
[4] modulo associativity $(S_1 \times S_2) \times S_3 \simeq S_1 \times (S_2 \times S_3) \simeq S_1 \times S_2 \times S_3$...

We can now apply AC on $\forall d_3 \exists d_2$:

$$\begin{aligned}(s_1, s_2, s_3) \in r \Rightarrow\ & \big(\forall a_1 \in A_1(s_1)\big)\\ & \big(\exists f \in A_2(s_2) \to A_3(s_3)\big)\\ & \boldsymbol{\big(\forall a_2 \in A_2(s_2)\big)}\\ & \boldsymbol{\big(\exists g \in D_3(s_3, f(a_2)) \to D_2(s_2, a_2)\big)}\\ & \big(\forall d_3 \in D_3(s_3, f(a_2))\big)\\ & \big(\exists d_1 \in D_1(s_1, d_1)\big)\\ & \quad \big(s_1[a_1/d_1], s_2[a_2/\boldsymbol{g}(d_3)], s_3[f(a_2)/d_3]\big) \in r\end{aligned}$$

and apply AC one more time on $\forall a_2 \exists g$ to obtain:

$$\begin{aligned}(s_1, s_2, s_3) \in r \Rightarrow\ & \big(\forall a_1 \in A_1(s_1)\big)\\ & \big(\exists f \in A_2(s_2) \to A_3(s_3)\big)\\ & \big(\exists G \in \textstyle\prod_{a_2 \in A_2(s_2)} D_3(s_3, f(a_2)) \to D_2(s_2, a_2)\big)\\ & \big(\forall a_2 \in A_2(s_2)\big)\big(\forall d_3 \in D_3(s_3, f(a_2))\big)\\ & \big(\exists d_1 \in D_1(s_1, d_1)\big)\\ & \quad \big(s_1[a_1/d_1], s_2[a_2/G_{a_2}(d_3)], s_3[f(a_2)/d_3]\big) \in r\end{aligned}$$

which is equivalent to

$$\begin{aligned}(s_1, s_2, s_3) \in r \Rightarrow\ & \big(\forall a_1 \in A_1(s_1)\big)\\ & \left(\exists (f, G) \in \begin{matrix}\sum_{f \in A_2(s_2) \to A_3(s_3)} \\ \prod_{a_2 \in A_2(s_2)} D_3(s_3, f(a_2)) \to D_2(s_2, a_2)\end{matrix}\right)\\ & \big(\forall (a_2, d_3) \in \textstyle\sum_{A_2(s_2)} D_3(s_3, f(a_2))\big)\\ & \big(\exists d_1 \in D_1(s_1, d_1)\big)\\ & \quad \big(s_1[a_1/d_1], s_2[a_2/G_{a_2}(d_3)], s_3[f(a_2)/d_3]\big) \in r\end{aligned}$$

By definition, this means that r is a simulation from w_1 to $w_2 \multimap w_3$.

Once more, all this formal manipulation keeps the computational content of the simulations. (Because AC is constructively valid.) $\square$

The notion of safety property from [12] corresponds to simulations from $\mathbf{1}$ to w, or equivalently, subsets x of S such that:

$$s \in x \Rightarrow \big(\exists a \in A(s)\big)\big(\forall d \in D(s, a)\big)\ s[a/d] \in x\ .$$

The analogy with strategies should be obvious: if x is a safety property, and $s \in x$ then the Angel has a strategy to avoid deadlocks, starting from s.

Multithreading. We now come to the last connective needed to interpret the λ-calculus. Its computational interpretation is related to the notion of *multi-threading, i.e.* the possibility to run several instances of a program in parallel. Let's start by defining synchronous multithreading in the most obvious way:

Definition 7. *If w is an interaction system on S, define $Ł(w)$, the multithreaded version of w to be the interaction system on* $\mathsf{List}(S)$ *with:*

$$\begin{aligned} Ł.A\big((s_1, \dots s_n)\big) &= A(s_1) \times \dots A(s_n)\\ Ł.D\big((s_1, \dots s_n), (a_1, \dots a_n)\big) &= D(s_1, a_1) \times \dots D(s_n, d_n)\\ Ł.n\big((s_1, \dots s_n), (a_1, \dots a_n), (d_1, \dots d_n)\big) &= \big(s_1[a_1/d_1], \dots s_n[a_n/d_n]\big)\ .\end{aligned}$$

This interaction system is just an "n-ary" version of the synchronous product. To get the abstract properties we want, we need to "quotient" multithreading by permutations. Just like multisets are list modulo permutation, so is $\mathbf{!}w$ the multithreaded $Ł(w)$ modulo permutations. This definition is possible because $Ł(w)$ is "compatible" with permutations: if σ is a permutation, we have

$$\begin{gathered}\sigma \cdot \big((s_1, \dots s_n)\big[(a_1, \dots a_n)/(d_1, \dots d_n)\big]\big) \\ = \\ \big(\sigma \cdot (s_1, \dots s_n)\big)[\sigma \cdot (a_1, \dots a_n)/\sigma \cdot (d_1, \dots d_n)] \ .\end{gathered}$$

The final definition is:

Definition 8. *If w is an interaction system on S, define $Ł(w)$, define $\mathbf{!}w$ to be the following interaction system on $\mathcal{M}_f(S)$:*

$$\begin{array}{ll} \mathbf{!}A(\mu) & = \sum_{\overline{s} \in \mu} Ł.A(\overline{s}) \\ \mathbf{!}D\big(\mu, (\overline{s}, \overline{a})\big) & = Ł.D(\overline{s}, \overline{a}) \\ \mathbf{!}n\big(\mu, (\overline{s}, \overline{a}), \overline{d}\big) & = \mathfrak{S} \cdot Ł.n(\overline{s}, \overline{a}, \overline{d}) \ . \end{array}$$

Unfolded, it gives:

- an action in state μ (a multiset) is given by an element $\overline{s}$ of the equivalence class μ (a list) together with an element $\overline{a}$ in $Ł.A(\overline{s})$ (a list of actions);
- a reaction is given by a list of reactions $\overline{d}$ in $Ł.D(\overline{s}, \overline{a})$;
- the next state is the equivalence class containing the list $\overline{s}[\overline{a}/\overline{d}]$ (the orbit of $\overline{s}[\overline{a}/\overline{d}]$ under the action of the group of permutations).

This operation enjoys a very strong algebraic property:

Proposition 2. *"$\mathbf{!}\_$" is a comonad in* **Int**.

Proof. We need to find two operations:

- $\varepsilon_w : \mathbf{!}w \to w$ defined as $\varepsilon_w = \big\{\big([s], s\big) \mid s \in S\big\}$;
- and $\delta_w : \mathbf{!}w \to \mathbf{!!}w$ defined as the converse of the graph of the "`concat`" function:
$$\delta_w = \big\{\big(\sum_{i \in I} \mu_i, [\mu_i]_{i \in I}\big) \mid \forall i \in I \ \mu_i \in \mathcal{M}_f(S)\big\}$$

For any w, those operations are indeed simulations: for ε_w, it is quite obvious, and for δ_w, it is quite painful to write. Let's only give an example from which the general case can easily be inferred:

1. we have $([[s_1, s_2, s_3], [t_1], []], [s_1, s_2, s_3, t_1]) \in \delta_w$
2. for any command $((a_1, a_2, a_3), (b_1), ())$ in state $[[s_1, s_2, s_3], [t_1], []]$, we need to find an action in $[s_1, s_2, s_3, t_1]$: simply take (a_1, a_2, a_3, b_1);
3. for any reaction (d_1, d_2, d_3, e_1) to this action, we need to find a reaction to the original command, *i.e.* to $((a_1, a_2, a_3), (b_1)())$: take $((d_1, d_2, d_3), (e_1), ())$;
4. the next states are respectively
   - $[[n(s_1, a_1, d_1), n(s_2, a_2, d_2), n(s_3, a_3, d_3)], [n(t_1, b_1, e_1)], []]$

– and $[n(s_1, a_1, d_1), n(s_2, a_2, d_2), n(s_3, a_3, d_3), n(t_1, b_1, e_1)]$.

They are indeed related through δ_w.

To be really precise, one would need to manipulate lists of states (representative of the multisets); but this only makes the proof even less readable.

Checking that the appropriate diagrams commute is immediate. It only involves the underlying sets and relations, and not the interaction systems or simulation conditions. (In fact , finite multisets form a comonad in the category of sets and relations...) □

2 Interpreting the λ-Calculus

We now have all the ingredients to give a denotational model for the typed λ-calculus: a type T will be interpreted by an interaction system T^*; and a judgement "$x_1 : T_1, \dots x_n : T_n \vdash t : T$" will be interpreted by simulation from $!T_1^* \otimes \dots !T_n^*$ to T^*.

2.1 Typing rules

The typing rules for the simply typed λ-calculus are given below:

1. $\dfrac{}{\Gamma \vdash x : \omega}$ if $x : \omega$ appears in Γ;
2. $\dfrac{\Gamma \vdash t : \omega \to \omega' \qquad \Gamma \vdash u : \omega}{\Gamma \vdash (t)u : \omega'}$;
3. $\dfrac{\Gamma, x : \omega \vdash t : \omega'}{\Gamma \vdash \lambda x.t : \omega \to \omega'}$.

We follow Krivine's notation for the application and write "$(t)u$" for the application of t to u.

2.2 Interpretation of Types

We assume a set of type variables ("propositional variables"): $X, \dots$ Nothing depend on the valuation we give to those type variables, so that we are almost interpreting Π^1 λ-calculus.[5]

For a valuation ρ from type variables to interaction systems, the interpretation of types is defined in the usual way:

Definition 9. *Let ω be a type. Define the interpretation ω^* of ω as:*

– $X^* = \rho(X)$;
– $(\omega \to \omega')^* = !\omega^* \multimap \omega'^*$.

[5] System-F in which all the quantifiers appear at the beginning of the term. To get an idea on how to get a real model of system-F, refer to [10].

2.3 Interpretation of Terms

If ω is a type, write $|\omega|$ for the set of states of its interpretation:

- $|X_i| = S_i$ (set of states of $\rho(X_i)$);
- $|\omega \to \omega'| = \big(\mathcal{M}_f|\omega|\big) \times |\omega'|$.

A valuation is a way to interpret typed variables from the context:

Definition 10. *If $\varGamma = x_1 : \omega_1, \dots x_n : \omega_n$ is a context, an environment for $\varGamma$ is a tuple γ in $\mathcal{M}_f|\omega_1| \times \dots \mathcal{M}_f|\omega_n|$. To simplify notation, we may write the tuple $\gamma = (\mu_1, \dots \mu_n)$ as "$x_1 := \mu_1, \dots x_n := \mu_n$". We may also write $\gamma(x)$ for the projection of γ on the appropriate coordinate. Sum of tuples of multisets is defined pointwise.*

We now interpret judgements: if we can type $\varGamma \vdash t : \omega'$ and if γ is an environment for $\varGamma$, the interpretation $[\![t]\!]_\gamma$ of term t in environment γ is a subset of $|\omega|$ defined as follows:

Definition 11. *We define $[\![t]\!]_\gamma$ by induction on t:*

1. *if we have* $\dfrac{}{\varGamma \vdash x : \omega}$ *with $x : \omega$ in $\varGamma$,*
   then $[\![x]\!]_\gamma = \begin{cases} \{s\} \text{ if } \gamma(x) = [s] \text{ and } \gamma(y) = [] \text{ whenever } x \neq y \\ \emptyset \quad \text{otherwise} \end{cases}$;
2. *if we have* $\dfrac{\varGamma \vdash t : \omega \to \omega' \qquad \varGamma \vdash u : \omega}{\varGamma \vdash (t)u : \omega'}$,
   then $s \in [\![(t)u]\!]_\gamma$ iff $(\mu, s) \in [\![t]\!]_{\gamma_0}$ for some $\mu = [s_1, \dots s_n] \in \mathcal{M}_f|\omega|$ s.t. $s_i \in [\![u]\!]_{\gamma_i}$ for all $i = 1, \dots n$ and $\gamma = \gamma_0 + \gamma_1 + \dots \gamma_n$;
3. *if we have* $\dfrac{\varGamma, x : \omega \vdash t : \omega'}{\varGamma \vdash \lambda x.t : \omega \to \omega'}$,
   then $[\![\lambda x.t]\!]_\gamma = \{(\mu, s) \mid \mu \in \mathcal{M}_f|\omega|,\ s \in [\![t]\!]_{\gamma, x:=\mu}\}$.

It is immediate to check that this definition is well formed.

If $\varGamma = x_1 : \omega_1, \dots x_n : \omega_n$, write $!\varGamma$ for $!\omega_1^* \otimes \dots !\omega_n$; similarly, we omit the superscript $\_^*$ and write ω for ω^*. The interpretation of terms is correct in the following sense:

Proposition 3. *Suppose that $\varGamma \vdash t : \omega'$, then the relation "$\_ \in [\![t]\!]_\_$" is a simulation relation from $!\varGamma$ to ω'.*
In other words, if $s \in [\![t]\!]_\gamma$, then s (in ω') simulates γ (in $!\varGamma$).

This is quite surprising because the interpretation of t doesn't depend on the interaction systems used to interpret the types but only the underlying set of states.[6]

[6] The interpretation is called the relational interpretation: it can be defined in the category of sets and relations...

Proof. We work by induction on the structure of the type inference.

1. Axiom: it amount to showing that $\{([], \dots [], [s], [], \dots [], s) \mid s \in |\omega|\}$ is a simulation from $!\Gamma$ to ω. This is easy: the only actions available in state $([], \dots [s], [], \dots)$ are of the form $((), \dots (a), () \dots)$ where $a \in A(s)$, and they are simulated by the action a. The reaction d is translated back into reaction $((), \dots, (d), (), \dots)$; and the rest is obvious.
2. Application: suppose we have $s \in [\![(t)u]\!]_\gamma$. By definition, we know that we have $(\mu, s) \in [\![t]\!]_{\gamma_0}$ for some $\mu = [s_1, \dots s_n]$ s.t. each s_i is in $[\![u]\!]_{\gamma_i}$ for a partition $\gamma = \gamma_0 + \gamma_1 + \dots \gamma_n$.
   By induction hypothesis, we thus know that (μ, s) (in $\omega \to \omega'$) simulates γ_0 (in $!\Gamma$); and that any s_i (in ω) simulates γ_i (in $!\Gamma$).
   Rather than doing the full formal proof (which involves many indices), we'll show how it works on an example. The general case can easily be deduced from that.
   Suppose Γ is reduced to a single assumption $x : \nu$ so that γ is reduced to a single multiset, $[v_1, v_2, v_3]$ for our example. Suppose $s \in [\![(t)u]\!]_{x:=[v_1,v_2,v_3]}$ because:
   - $([t_1, t_2], s) \in [\![t]\!]_{x:=[v_2]}$
   - $t_1 \in [\![u]\!]_{x:=[v_1,v_3]}$ and $t_2 \in [\![u]\!]_{x:=[]}$.

   We need to show that s simulates $[v_1, v_2, v_3]$:
   (a) suppose $a_1 \in A_\nu(v_1)$, $a_2 \in A_\nu(v_2)$ and $a_3 \in A_\nu(v_3)$;
   (b) we need to find an action in $A_{\omega'}(s)$ simulating (a_1, a_2, a_3):
      (1) by induction hypothesis, t_1 simulates $[v_1, v_3]$, so that we can find an action $b_1 \in A_\omega(t_1)$ simulating (a_1, a_3);
      (2) similarly, t_2 simulates $[]$, so that we can find an action $b_2 \in A_\omega(t_2)$ simulating $()$;
      (3) we also have that $([t_1, t_2], s)$ (in $\omega \to \omega'$) simulates $[v_2]$ (in $!\nu$). By proposition 1, this is equivalent to saying that s (in ω') simulates $([v_2], [t_1, t_2])$ (in $!\nu \otimes !\omega$).
      Thus, we can find an action $a \in A_{\omega'}(s)$ simulating $\big((a_2), (b_1, b_2)\big)$.
      By composing the above two simulations on the right ((b_1, b_2) simulates (a_1, a_3)), we thus obtain that a simulates (a_1, a_2, a_3).
      We now need to translate the reactions back: let $d \in D_{\omega'}(s, a)$,
      (3) by induction, we can translate d into a reaction $\big((d_2), (e_1, e_2)\big)$ to $\big((a_2), (b_1, b_2)\big)$;
      (2) we can translate e_2 into a reaction $()$ to b_2;
      (1) and finally we can translate e_1 into a reaction (d_1, d_3) to (a_1, a_3).
      Thus, we obtain reactions $d_1 \in D_\nu(v_1, a_1)$, $d_2 \in D_\nu(v_2, a_2)$ and $d_3 \in D_\nu(v_3, a_3)$.
   (c) The new states we get from those actions/reactions are: $s[a/d]$ on one side; and $[v_1[a_1/d_1], v_2[a_2/d_2], v_3[a_3/d_3]]$ on the other side. They are indeed related because:
      (1) $t_1[b_1/e_1] \in [\![u]\!]_{x:=[v_1[a_1/d_1], v_3[a_3/d_3]]}$;
      (2) $t_2[b_2/e_2] \in [\![u]\!]_{x:=[]}$;
      (3) and finally $[t_1[b_1/e_1], t_2[b_2/e_2]] \in [\![t]\!]_{x:=v_2[a_2/d_2]}$.

3. Abstraction: this is immediate. Suppose $(\mu, s) \in [\![\lambda x.t]\!]_\gamma$; we need to show that (μ, s) (in $\omega \to \omega'$) simulates γ (in $!\Gamma$). By proposition 1, this is equivalent to showing that s (in ω') simulates (γ, μ) (in $!\Gamma \otimes !w$). This is exactly the induction hypothesis.

□

To summarise all this, here is a tentative rewording of the above: if $\Gamma \vdash t : \omega$,

(1) each type represent a process;
(2) each process in the context can be run in parallel multiple times;
(3) the environment γ represents the initial states for the context;
(4) if $s \in [\![t]\!]_\gamma$ then s can be used as an initial state to simulate γ;
(5) the algorithm for the simulation is contained in t.

To finish the justification that we have a denotational model, we now need to check that the interpretation is invariant by β-reduction.

Proposition 4. *For all terms t and u and environment γ, we have*

$$[\![(\lambda x.t)u]\!]_\gamma = [\![t[u/x]]\!]_\gamma \ .$$

The proof works by induction and is neither really difficult nor very interesting. It can be found on `http://iml.univ-mrs.fr/~hyvernat/academics.html`.

3 Interpreting the Differential λ-calculus

Simulation relations from w to w' enjoy the additional property that they form a complete sup-lattice:

Lemma 2. *The empty relation is always a simulation from any w to w'; and if $(r_i)_{i\in I}$ is a family of simulations from w to w', then $\bigcup_{i\in I} r_i$ is also a simulation from w to w'.*

The proof is immediate...

Unfortunately, this doesn't reflect any property of λ-terms. The reason is that *(1)* not every type is inhabited, and *(2)* we do not see *a priori* how to take the union of two terms. For example, what is the meaning of $\lambda x\lambda y.x \cup \lambda x\lambda y.y$ in the type $X \to X \to X$?[7]

Ehrhard and Regnier's *differential λ-calculus* ([6]) extends the λ-calculus by adding a notion of differentiation of λ-terms. One consequence is that we need to have a notion of sum of arbitrary terms, interpreted as a non-deterministic choice. It is not the right place to go into the details of the differential λ-calculus and we refer to [6] for motivations and a complete description.

In the typed case, we have the following typing rules:

[7] In terms of usual datatypes translation, this term would be $\mathtt{t} \cup \mathtt{f}$ in the type $\mathbf{B}$.

1. $\dfrac{}{\Gamma \vdash 0 : \omega}$ and $\dfrac{\Gamma \vdash t : \omega \qquad \Gamma \vdash u : \omega}{\Gamma \vdash t + u : \omega}$;
2. $\dfrac{\Gamma \vdash t : \omega \to \omega' \qquad \Gamma \vdash u : \omega}{\Gamma \vdash \mathrm{D}\, t \cdot u : \omega \to \omega'}$.

The intuitive meaning is that "$\mathrm{D}\, t \cdot u$" is the result of (non-deterministically) replacing *exactly one occurrence* of the first variable of t by u. We thus obtain a sum of terms, depending on which occurrence was replaced. This gives a notion of differential substitution (or linear substitution) which yields a *differential-reduction.* The rules governing this reduction are more complex than usual β-reduction rules. We refer to [6] for a detailed description.

We extend the interpretation of terms in the following way:

Definition 12. *Define the interpretation of a typed differential λ-term by induction on the type inference:*

1. *if we have* $\dfrac{}{\Gamma \vdash 0 : \omega}$ *, then we put* $[\![0]\!]_\gamma = \emptyset$*;*
2. *if we have* $\dfrac{\Gamma \vdash t : \omega \qquad \Gamma \vdash u : \omega}{\Gamma \vdash t + u : \omega}$ *,*
   then we put $[\![t + u]\!]_\gamma = [\![t]\!]_\gamma \cup [\![u]\!]_\gamma$*;*
3. *if we have* $\dfrac{\Gamma \vdash t : \omega \to \omega' \qquad \Gamma \vdash u : \omega}{\Gamma \vdash \mathrm{D}\, t \cdot u : \omega \to \omega'}$ *,*
   then we put $(\mu, s') \in [\![\mathrm{D}\, t \cdot u]\!]_\gamma$ *iff* $(\mu + [s], s') \in [\![t]\!]_{\gamma_1}$ *for some* $s \in [\![u]\!]_{\gamma_2}$ *s.t.* $\gamma = \gamma_1 + \gamma_2$.

Proposition 3 extends as well:

Proposition 5. *Suppose that $\Gamma \vdash t : \omega'$ where Γ is a context and t a differential λ-term. The relation "$\_ \in [\![t]\!]_\_$" is a simulation relation from $!\Gamma$ to ω'.*

Proof. The proof for the sum and the 0 are contained in proposition 2.

For differentiation, suppose we have $(\mu, s') \in [\![\mathrm{D}\, t \cdot u]\!]_\gamma$, *i.e.* $(\mu + [s], s') \in [\![t]\!]_{\gamma_1}$ for some $s \in [\![u]\!]_{\gamma_2}$, with $\gamma = \gamma_1 + \gamma_2$. We need to show that (μ, s') (in $\omega \to \omega'$) simulates γ (in $!\Gamma$). Since $\gamma = \gamma_1 + \gamma_2$, it is enough to show that we can simulate (γ_1, γ_2) (in $!\Gamma \otimes !\Gamma$).
By proposition 1, this is equivalent to showing that s' (in ω') simulates $(\gamma_1, \gamma_2, \mu)$ (in $!\Gamma \otimes !\Gamma \otimes !\omega$).
Let $a_{\gamma_1} \in !A_\Gamma(\gamma_1)$, $a_{\gamma_2} \in !A_\Gamma(\gamma_2)$ and $a_\mu \in !A_\omega(\mu)$; we need to find an action in $A_{\omega'}(s')$ to simulate $(a_{\gamma_1}, a_{\gamma_2}, a_\mu)$:

(1) by induction hypothesis, we know that s (in ω) simulates γ_2 (in $!\Gamma$); so that we can find an action $a \in A_\omega(s)$ simulating a_{γ_2};

(2) by induction, we know that s' (in ω') simulates $(\gamma_1, \mu + [s])$ (in $!\Gamma \otimes !\omega$), so that we can find an action $a' \in A_{\omega'}(s')$ simulating $\big(a_{\gamma_1}, (a_\mu, a)\big)$.
Since a simulates a_{γ_2}, by composition, a simulates $\big(a_{\gamma_1}, (a_\mu, a_{\gamma_2})\big)$; and by associativity and commutativity, we can thus simulate $(a_{\gamma_1}, a_{\gamma_2}, a_\mu)$.

To translate back a reaction d' to a' into a reaction $(d_{\gamma_1}, d_{\gamma_2}, d_\mu)$, we proceed similarly:

(2) by induction, we can translate d' into a reaction (d_{γ_1}, d_μ, d) to $\big(a_{\gamma_1}, (a_\mu, a)\big)$;
(1) by induction, we can also translate the reaction d (in $D_\omega(s, a)$) into a reaction d_{γ_2} (in $!D_\Gamma(s, a_{\gamma_2})$).

We thus obtain reactions d_{γ_1}, d_{γ_2} and d_μ as desired. That the resulting next states are still related is quite obvious... $\square$

We now need to check that the interpretation is invariant by β-reduction and differential reduction.

Proposition 6. *For all differential terms t and u and environment γ, we have:*

$$\begin{aligned} [\![(\lambda x.t)u]\!]_\gamma &= [\![t[u/x]]\!]_\gamma \\ [\![\mathrm{D}(\lambda x.t) \cdot u]\!]_\gamma &= [\![\lambda x \ . \ (\partial t/\partial x) \cdot u]\!]_\gamma \end{aligned}$$

Just like for Proposition 4, the proof is quite easy but tedious. The interested reader can find it at `http://iml.univ-mrs.fr/~hyvernat/academics.html`.

Conclusion

Technically speaking, this work is not very different from [12], which is itself quite close to [11]. The main reasons for producing it are:

- first, it shows that we can give a computational content to the notion of simulation if we do not try to interpret all of linear logic;
- second, it shows that some of the additional structure of interaction systems and simulation does have a logical significance. We showed that by interpreting the differential λ-calculus.

Even if we haven't done it formally, it is quite easy to extend the model to full intuitionistic linear logic while keeping the computational content of simulations. To define the additive, we use the definition of $\oplus$ from [12].

It is in principle possible to formalise all the above in a proof assistant (Agda [4] or Coq [5] come to mind).[8] From such a system, one could extract the simulations. For example, a term of type $T \to T'$ would give an algorithm simulating many synchronous occurrences of T by a single occurrence of T'.

[8] One needs to be careful to be able to deal with the notion of equivalence classes used in the definition of $!w$. The idea is to use interaction systems on "setoids", where the equivalence relation is a simulation...

It is however difficult to apply this to obtain real-life simulations. The problem is that we only get "purely logical" simulations. Simulations of interest for application rely heavily on the different interaction systems used. One way to get more interesting simulations (from a practical point of view) might be to use constant interaction systems (booleans, natural numbers, or more practical ones like stacks, memory cells, etc..) as ground types, together with specific simulations (the values true and false, successor function, or more practical simulations) as inhabitant of specific types.

In pretty much the same way as [12] makes **Int** into a denotational model for classical linear logic, we can make interaction systems into a denotational model for "classical differential linear logic": differential interaction nets [7]. This system doesn't make much sense logically speaking, but seems to enjoy relationship with process calculi. This is an encouraging direction of research.

References

1. Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.
2. Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for IDEALIZED ALGOL with active expressions. In *ALGOL-like languages, Vol. 2*, Progr. Theoret. Comput. Sci., pages 297–329. Birkhäuser Boston, Boston, MA, 1997.
3. Sansom Abramsky, Dan Ghica, Luke Ong, and Andrzej Murawski. Applying game semantics to compositional software modelling and verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 421–435. Springer-Verlag, 2004.
4. The Agda proof assistant. http://www.cs.chalmers.se/∼catarina/agda/.
5. The Coq proof assistant. http://coq.inria.fr/.
6. Thomas Ehrhard and Laurent Regnier. The differential lambda calculus. *Theoret. Comput. Sci.*, 309(1):1–41, 2003.
7. Thomas Ehrhard and Laurent Regnier. Differential interaction nets. Invited paper, Workshop on Logic, Language, Information and Computation (WoLLIC), 2004.
8. Peter Hancock and Pierre Hyvernat. Programming as applied basic topology. to be published in Annals of Pure and Applied logic, 2004.
9. J. Martin E. Hyland and Luke Chih-Hao Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(2):285–408, 2000.
10. Pierre Hyvernat. Predicate transformers and linear logic: second order. unpublished note, 2004.
11. Pierre Hyvernat. Predicate transformers and linear logic: yet another denotational model. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Workshop CSL 2004*, volume 3210 of *LNCS*, pages 115–129. Springer-Verlag, September 2004.
12. Pierre Hyvernat. Synchronous games, deadlocks and linear logic. unpublished note, 2005.
13. Robin Milner. Calculi for synchrony and asynchrony. *Theoret. Comput. Sci.*, 25(3):267–310, 1983.