# Token-passing Optimal Reduction with Embedded Read-back

Anton Salikhmetov

anton.salikhmetov@gmail.com

We introduce a new interaction net implementation of optimal reduction for the pure untyped lambda calculus. Unlike others, our implementation allows to reach normal form regardless of the interaction net reduction strategy using the approach of so-called token-passing nets and a non-deterministic extension for interaction nets. Another new feature is the read-back mechanism implemented without leaving the formalism of interaction nets.

## 1 Introduction

Optimal reduction for the $\lambda$-calculus was defined by Lévy [9]. One of its implementations is done by Lamping [7]. Independently in [6], Lafont introduced interaction nets which are graph rewriting systems similar to those used by Lamping. Later, Lamping's optimal algorithm was redefined using the formalism of interaction nets; see [2] where the problem of optimal reduction is covered in great detail.

Interaction nets were found as capable to encode reduction strategies directly [10]. Also, a new approach of token-passing nets was introduced and demonstrated to implement call-by-value and call-by-name evaluation of $\lambda$-terms using pure interaction nets thanks to Sinot [12]. Finally, he achieved a token-passing net implementation of call-by-need evaluation [13] by leaving the formalism of interaction nets. Almeida, Pinto, and Vilaça analyzed and generalized Sinot's approach to a wider class of systems [1], and also stated a (currently still open) question if token-passing nets are applicable to closed reduction [5].

This paper is a part of an ongoing work[1] on implementation of the pure untyped $\lambda K$-calculus. For that purpose, we have introduced a domain-specific language based on the interaction calculus [4] with a non-deterministic extension [3] and side effects. However, our software implementation of this language has some limitations implied by eventually aiming at a distributed computation network. Specifically, we have chosen not to support any specific interaction net reduction strategy as different computation nodes are meant to work independently with as minimal synchronization to each other as possible.

Hence, the optimal algorithm and many other interaction net implementations of the $\lambda$-calculus are not possible in our setting as they rely upon weak strategies aiming to reach interface-normal form of a net. Also, they often require external garbage collection which we would rather prefer to delegate to interaction nets themselves. Still, we found that token-passing nets fit us perfectly. In particular, we have successfully adapted [11] Sinot's call-by-need for free [13]. Our current work is mostly based on the latter adaptation, thus we also base our system on interaction nets with non-deterministic extension.

In this paper, we apply the approach of token-passing nets to optimal reduction of pure untyped $\lambda K$-terms. Our optimal implementation has two new features: reaching normal form (if any) regardless of the interaction net reduction strategy and producing textual representation of the normal form. The first feature is in contrast with other implementations that require avoiding interactions in disconnected parts of a net. The second feature makes more sophisticated use of eval and return tokens which Sinot's token-passing nets are based on.

---

[1] The latest stable version of our implementation is available online at `https://codedot.github.io/lambda/` and works in modern Web browsers with no server side, computation being performed solely on the client side.

Figure 1: Primitives

(a) Agent                          (b) Wiring                          (c) Tree
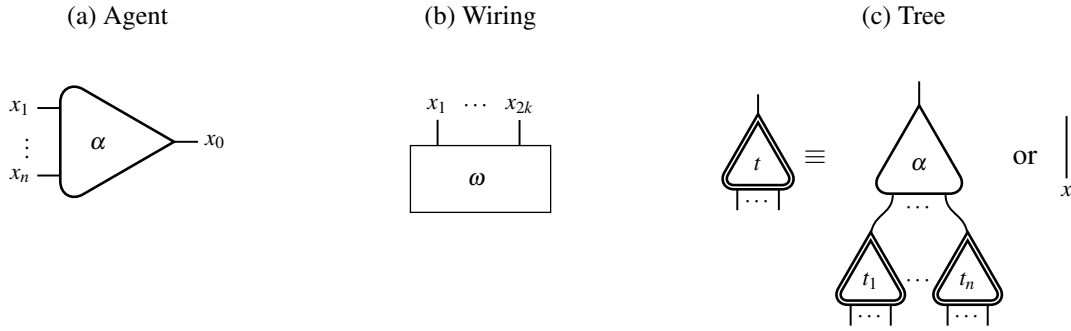


## 2   Preliminaries

This section gives a rather informal brief introduction to interaction nets and their textual representation called the interaction calculus [4]. Also, we will give an overview of our programming language based on the interaction calculus which we use for software experiments with interaction nets.

Interaction nets are graph-like structures consisting of primitives shown in Figure 1. *Agents* of type $\alpha$ can be graphically represented as shown in Figure 1a. Agents have *arity* $\mathrm{ar}(\alpha) \geq 0$. If $\mathrm{ar}(\alpha) = n$, the agent $\alpha$ has $n$ *auxiliary ports* $x_1, \ldots, x_n$ in addition to its *principal port* $x_0$. All agent types belong to a set $\Sigma$ called *signature*. Any port can be connected to at most one edge and ports not connected to any edge are called *free ports*, the latter ones forming the *interface* of an interaction net. *Wiring* $\omega$ on Figure 1b consists solely of edges. Inductively defined *trees* on Figure 1c correspond to *terms* $t ::= \alpha(t_1, \ldots, t_n) \mid x$ in the interaction calculus, where $x$ is called a *name*.

Any net $N$ can be redrawn using the previously defined wiring and tree primitives as follows:
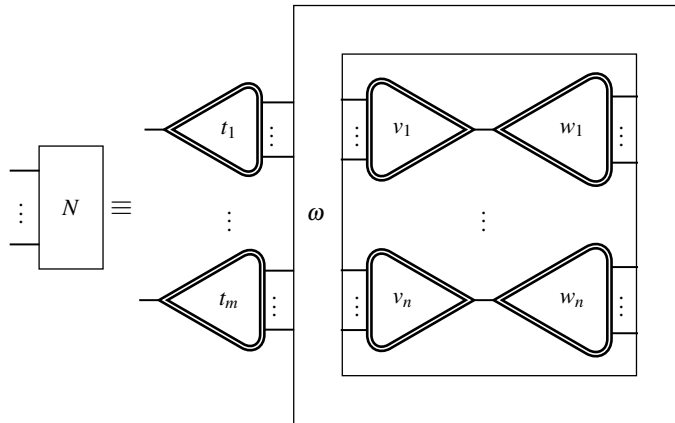


which in the interaction calculus corresponds to a *configuration* $\langle t_1, \ldots, t_m \mid v_1 = w_1, \ldots, v_n = w_n \rangle$, where $t_i$, $v_i$, and $w_i$ are arbitrary terms. The ordered sequence $t_1, \ldots, t_m$ in the left-hand side is called an *interface*, while the right-hand side contains an unordered multiset of *equations* $v_i = w_i$. The wiring $\omega$ translates to names, and each name has to occur exactly twice in a configuration.

Just like in the $\lambda$-calculus, the interaction calculus has the notions of $\alpha$-conversion and substitution naturally defined on configurations. Specifically, both occurrences of any name can be replaced with a new name if the latter does not occur in a given configuration. In turn, *substitution* $t[x := u]$ is the result of replacing the name $x$ in term $t$ with another term $u$ if $x$ has exactly one occurrence in term $t$.

When two agents are connected to each other with their principal ports, they form an *active pair*. For active pairs one can introduce *interaction rules* which describe how the active pair rewrites to another net. Graphically, any interaction rule can be represented as follows:
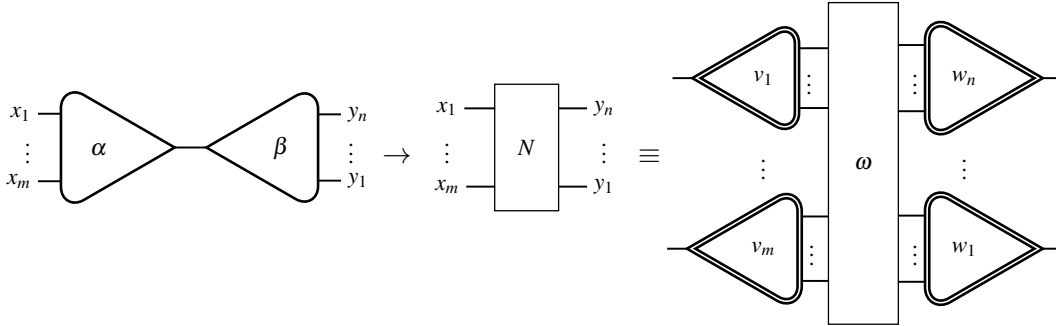


where $\alpha, \beta \in \Sigma$, and the net $N$ is redrawn using primitives of wirings and trees in order to translate the rule into the interaction calculus as $\alpha[v_1, \ldots, v_m] \bowtie \beta[w_1, \ldots, w_n]$ using Lafont's notation. A net with no active pairs is said to be in *normal form*. A signature $\Sigma$ (with mapping ar defined on it) along with a set of interaction rules defined for agents $\alpha \in \Sigma$ together constitute an *interaction system*.

Now, let us consider an example for the notions introduced above in this section. Figure 2 shows two interaction rules for commonly used agents $\varepsilon$ and $\delta$ and a simple interaction net to which these interaction rules are applied. Using Lafont's notation, the erasing rule from Figure 2a is written as $\varepsilon \bowtie \alpha[\varepsilon, \ldots, \varepsilon]$, while the duplication rule given in Figure 2b can be represented as follows:

$$\delta[\alpha(x_1, \ldots, x_n), \alpha(y_1, \ldots, y_n)] \bowtie \alpha[\delta(x_1, y_1), \ldots, \delta(x_n, y_n)].$$

Figure 2c provides an example of a non-terminating net which reduces to itself. In terms of the interaction calculus, one can write this net as a configuration $\langle \varnothing \mid \delta(\varepsilon, x) = \gamma(x, \varepsilon) \rangle$ with no interface.
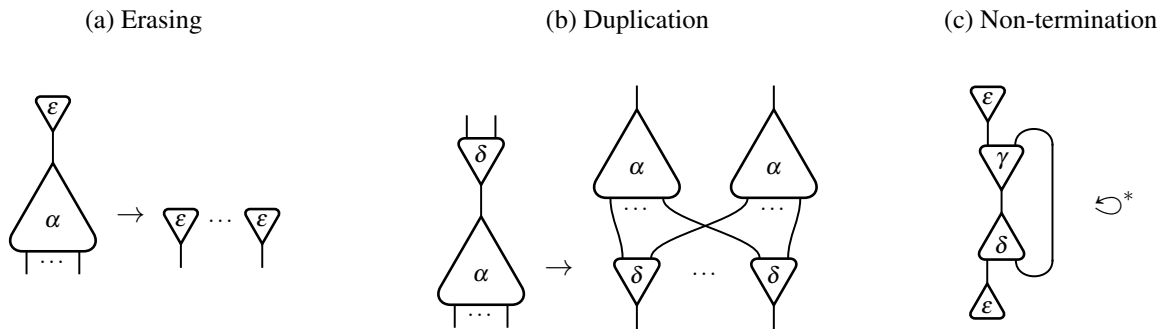
The interaction calculus defines reduction on configurations in more details than seen from graph rewriting defined on interaction nets. Namely, if $\alpha[v_1, \ldots, v_m] \bowtie \beta[w_1, \ldots, w_n]$, the following reduction:

$$\langle \vec{t} \mid \alpha(t_1, \ldots, t_m) = \beta(u_1, \ldots, u_n), \Delta \rangle \to \langle \vec{t} \mid t_1 = v_1, \ldots, t_m = v_m, u_1 = w_1, \ldots, u_n = w_n, \Delta \rangle$$

is called *interaction*. When one of equations has the form of $x = u$, *indirection* can be applied resulting in substitution of the other occurrence of the name $x$ in some term $t$:

$$\langle \ldots t \ldots \mid x = u, \Delta \rangle \to \langle \ldots t[x := u] \ldots \mid \Delta \rangle \quad \text{or} \quad \langle \vec{t} \mid x = u, t = w, \Delta \rangle \to \langle \vec{t} \mid t[x := u] = w, \Delta \rangle.$$

Figure 2: Example

(a) Erasing                          (b) Duplication                          (c) Non-termination

An equation $t = x$ is called a *deadlock* if the name $x$ has occurrence in the term $t$. We only consider deadlock-free nets. Together, interaction and indirection define the reduction relation on configurations. The fact that configuration $c$ reduces to its *normal form* $c'$ with no equations left is denoted as $c \downarrow c'$.

Coming back to the example of a non-terminating net shown in Figure 2c, the infinite reduction sequence starting from the corresponding configuration in the interaction calculus is as follows:

$$\langle \varnothing \mid \delta(\varepsilon, x) = \gamma(x, \varepsilon) \rangle \rightarrow$$
$$\langle \varnothing \mid \varepsilon = \gamma(x_1, x_2),\ x = \gamma(y_1, y_2),\ x = \delta(x_1, y_1),\ \varepsilon = \delta(x_2, y_2) \rangle \rightarrow^*$$
$$\langle \varnothing \mid x_1 = \varepsilon,\ x_2 = \varepsilon,\ x = \gamma(y_1, y_2),\ x = \delta(x_1, y_1),\ x_2 = \varepsilon,\ y_2 = \varepsilon \rangle \rightarrow^*$$
$$\langle \varnothing \mid \delta(\varepsilon, x) = \gamma(x, \varepsilon) \rangle \rightarrow \dots$$

In our DSL[2] which is similar to the UNIX utilities `yacc(1)`/`lex(1)` by structure and lexically close to LaTeX representation for the interaction calculus, the example net can be specified as follows:

```
\epsilon {
        console.log("epsilon >< delta");
} \delta[\epsilon, \epsilon];

\epsilon {
        console.log("epsilon >< gamma");
} \gamma[\epsilon, \epsilon];

\delta[\gamma(x, y), \gamma(v, w)] {
        console.log("delta >< gamma");
} \gamma[\delta(x, v), \delta(y, w)];


$$


\delta(\epsilon, x) = \gamma(x, \epsilon);
```

Note that our programming language allows side effects written in imperative style, thus enabling execution of arbitrary code, including input/output as well as conditional multiple rules for a given pair $\alpha_i \bowtie \beta_j$ depending on $i$ and $j$, while manipulating $i$ and $j$ as arbitrary data attached to agents.

Our implementation of this language does not rely on any external garbage collection as it does not distinguish disconnected subnets. The latter is due to lack of interface as we specify a net by a multiset of equations only. In order to still represent a net with a non-empty interface, one needs to modify the net, for instance, by attaching agents with zero arity to all of its free ports.

An interaction system's signature and arities of agents are derived automatically at compile time based on input interaction rules and initial configuration. The interaction rules along with their side effects are compiled in advance before starting reduction of a net, the compiler preparing a table for a fast $O(1)$ rule search at run time. After that, the initial configuration is pushed into a FIFO queue which is then processed by evaluator until no more equations left.

Finally, implementation of our DSL includes non-deterministic extension for interaction nets in the same form as described in Section 3, namely using McCarthy's `amb` agent which allows to simulate agents with multiple principal ports such as the sharing agent $s$ used by Sinot in [13].

---

[2] Our programming language is provided with a Node.js package called JavaScript Engine for Interaction Nets which is available at `https://www.npmjs.com/package/inet-lib` in the NPM repository.

# 3 Non-deterministic extension

We work in the interaction calculus [4] extended with a non-deterministic agent amb [3]. This agent will be required in the waiting construct which is described in Section 5. Specifically, it is used to handle non-deterministic nature of deciding whether a given sub-term of an encoded $\lambda K$-term is to be reduced and erased. The context where we need the amb agent is similar to where multiple principal ports were necessary for token-passing net implementation of call-by-need evaluation by Sinot.

Figure 3: Non-deterministic agent



We represent this agent in a more conservative fashion than it was suggested in the original paper [3]. Specifically, we prepend the list of auxiliary ports of amb with its extra principal port and introduce the following conversion:

$$\langle \vec{t} \mid t = \mathsf{amb}(u,v,w), \Delta \rangle = \langle \vec{t} \mid u = \mathsf{amb}(t,v,w), \Delta \rangle.$$

We assume that any interaction system's signature $\Sigma$ is implicitly extended by amb with $\mathsf{ar}(\mathsf{amb}) = 3$, while its set of rules is implicitly extended with

$$\forall \alpha \in \Sigma : \alpha[\vec{x}] \bowtie \mathsf{amb}[y, \alpha(\vec{x}), y].$$

Figure 3 gives a graphical representation of amb.

# 4 Optimal reduction

The pure interaction net implementation of optimal reduction upon which we base our interaction system is the main one used through most of the book by Asperti and Guerrini [2, pp. 39–40]. In that system, all agents have box level numbers attached to them. From the viewpoint of the interaction calculus, this fact results in an interaction system with an infinite signature and an infinite set of interaction rules. Specifically, its signature can be defined as follows:
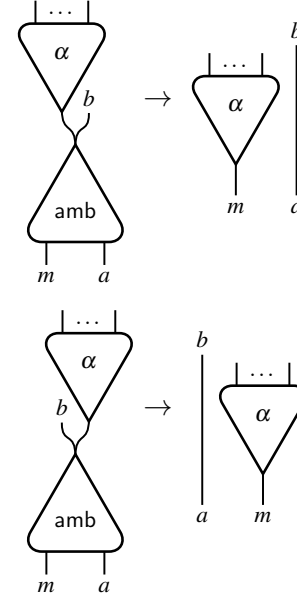
$$\Sigma_O = \{\varepsilon\} \cup \{\lambda_i, @_i, \delta_i, \mathsf{\cap}_i, \sqcup_i \mid i \in \mathbb{N}\},$$

with $\mathsf{ar}(\varepsilon) = 0$, $\mathsf{ar}(\mathsf{\cap}_i) = \mathsf{ar}(\sqcup_i) = 1$, and $\mathsf{ar}(\lambda_i) = \mathsf{ar}(@_i) = \mathsf{ar}(\delta_i) = 2$. In Section 7 where we define the embedded read-back mechanism, we will extend the signature with agents that have $\lambda$-terms and contexts attached rather than natural numbers.

The set of interaction rules consists of two groups. The first group includes *annihilation* rules:

$$\mathsf{\cap}_i[x] \bowtie \mathsf{\cap}_i[x];$$
$$\sqcup_i[x] \bowtie \sqcup_i[x];$$
$$\lambda_i[x,y] \bowtie @_i[x,y];$$
$$\delta_i[x,y] \bowtie \delta_i[x,y].$$

The second group contains the following *propagation* rules, where $\alpha_j \in \Sigma_O$ and $i < j$:

$$\cap_i[\alpha_{j-1}(x_1,\ldots,x_n)] \bowtie \alpha_j[\cap_i(x_1),\ldots,\cap_i(x_n)];$$
$$\sqcup_i[\alpha_{j+1}(x_1,\ldots,x_n)] \bowtie \alpha_j[\sqcup_i(x_1),\ldots,\sqcup_i(x_n)];$$
$$\delta_i[\alpha_j(x_1,\ldots,x_n),\alpha_j(y_1,\ldots,y_n)] \bowtie \alpha_j[\delta_i(x_1,y_1),\ldots,\delta_i(x_n,y_n)].$$

The initial encoding of $\lambda$-terms we base our work on is the one given in [2, pp. 41–42], although other compatible versions can also be used as described in [2, Chapter 8]. We preserve both original interaction rules and original initial encoding with exceptions for the $\beta$-reduction $\lambda_i \bowtie @_i$ rule modified and free variables allowed in a given $\lambda$-term. The former modification is to be covered in Section 5 about the waiting construct, and the latter feature will be a part of the read-back mechanism embedded into our interaction system described in Section 7. Together, they will constitute our token-passing net implementation of optimal reduction.

In order to make it easier to work with the interaction calculus below, we will denote the (graphical) encoding $[M]$ of a given $\lambda$-term $M$ as (textual) configuration $\langle x \mid [M,x] \rangle$, where $x$ is a name for the only free port in its interface, and $[M,x]$ is a multiset of equations that correspond to the initial encoding $[M]$.

## 5   Waiting construct

Our main modification to the optimal algorithm on the way to its token-passing net version is to replace the $\beta$-reduction rule $@_i[x,y] \bowtie \lambda_i[x,y]$ with

$$@_i[x,y] \bowtie \lambda_i[\mathsf{wait}(z,\mathsf{hold}(z,x)),y],$$

extending the original signature $\Sigma_O$ with a set of agents used to represent what we call the *waiting construct*:

$$\Sigma_W = \Sigma_O \cup \{\mathsf{wait},\mathsf{hold},\mathsf{decide},\mathsf{eval},\mathsf{call}\},$$

with $\mathsf{ar}(\mathsf{wait}) = \mathsf{ar}(\mathsf{hold}) = \mathsf{ar}(\mathsf{decide}) = 2$, $\mathsf{ar}(\mathsf{eval}) = 1$, and $\mathsf{ar}(\mathsf{call}) = 0$. Figure 4 provides graphical representation of the modified $\beta$-reduction rule. Note the waiting construct appearing between variable $b$ and argument $d$ is similar to the ! agent used by Mackie in [10] to cut edges corresponding to active pairs.

Figure 4: Waiting construct



The waiting construct propagates through the body of every abstraction after substitution, blocking possibly unnecessary $\beta$-redexes until they are called. This mechanism consists of several additional interaction rules we define through the rest of this section. Since the process of deciding whether a given redex is needed has non-deterministic nature, it is the waiting construct that requires non-deterministic extension for interaction nets we discussed earlier. In particular, interaction between a fan-in agent (denoted below as $\delta_i$) and wait results in creation of an ambiguous decide agent with two principal ports, the latter one being simulated using amb as shown in Figure 5.

Figure 6 illustrates the $@_i \bowtie \mathsf{wait}$ rule. Note that rather than just passing through application, the waiting construct initiates another waiting construct on the way to the root of application. Otherwise, an $\varepsilon$ agent that performs garbage collection from the root of application might be unable to reach either of application's sides. That, in turn, could lead to a disconnected net, which would become blocked non-interacting garbage. We avoid that thanks to the structure of our $@_i \bowtie \mathsf{wait}$ rule.
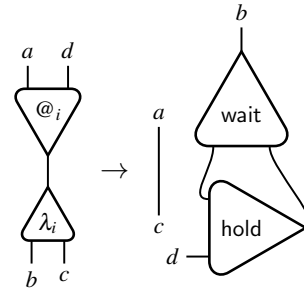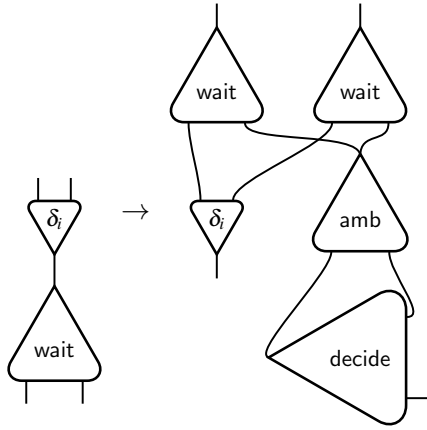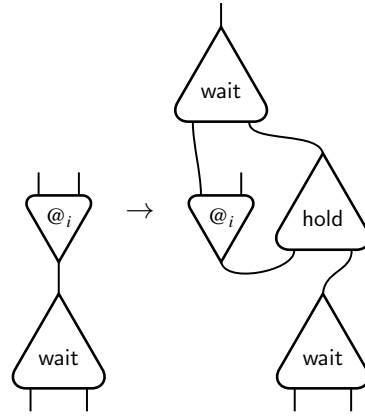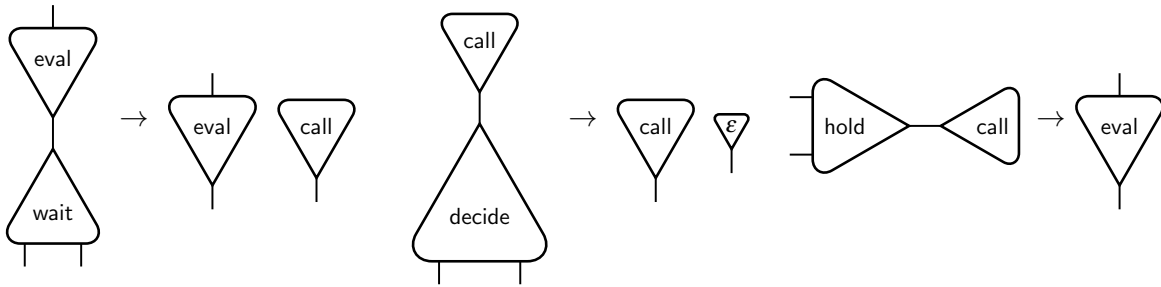
Figure 5: Propagation through fan-in

Figure 6: Propagation through application



Unblocking evaluation happens through wait ⋈ eval interaction which is implemented in a fashion similar to how evaluation strategies are encoded in Mackie's paper [10]. Although, the corresponding interaction rules are different due to the simulation of multiple principal ports:



The rest of propagation and garbage collection interaction rules are more or less straightforward, so we use Lafont's notation instead to put them all together:

$$\mathsf{eval}[\lambda_i(x,y)] \bowtie \lambda_i[x,\mathsf{eval}(y)];$$
$$\mathsf{eval}[\delta_i(x,y)] \bowtie \delta_i[x,y];$$
$$\mathsf{eval}[x] \bowtie \mathsf{wait}[\mathsf{eval}(x),\mathsf{call}];$$
$$\mathsf{call} \bowtie \mathsf{hold}[x,\mathsf{eval}(x)];$$
$$\delta_i[\mathsf{wait}(x,\mathsf{amb}(y,\mathsf{decide}(z,v),v)),\mathsf{wait}(w,y)] \bowtie \mathsf{wait}[\delta_i(x,w),z];$$
$$\mathsf{call} \bowtie \mathsf{decide}[\mathsf{call},\varepsilon];$$
$$\varepsilon \bowtie \mathsf{decide}[x,x];$$
$$@_i[x,\mathsf{wait}(y,\mathsf{hold}(@_i(x,y),\mathsf{wait}(v,w)))] \bowtie \mathsf{wait}[v,w];$$
$$\sqcap_i[\mathsf{wait}(x,y)] \bowtie \mathsf{wait}[\sqcap_i(x),y];$$
$$\sqcup_i[\mathsf{wait}(x,y)] \bowtie \mathsf{wait}[\sqcup_i(x),y].$$

# 6    Remarks on efficiency

Note that our solution is of proof-of-concept nature, and currently we do not directly aim at efficiency. Still, our approach preserves some parallelism of Lamping's optimal algorithm, as there are potentially multiple eval agents operating simultaneously throughout the interaction net. This is in contrast to the original token-passing net implementations [12, 13].

   Our main concern was the capability of our system to handle corner cases for closed reduction [5] and to reach the normal form of essentially $\lambda K$-terms. During development and testing, some of the example terms we used were $(\lambda f x.K x f)\,\Omega$, $\omega\,(\omega\,(\omega\,(\lambda f x.f\,(f\,x))))$ where $\omega = \lambda x.xx$, and $(\lambda x.M)\,(Y I)$ where $x \notin \mathsf{FV}(M)$ with various combinations of terms $M$ and fixed point combinators $Y$.

   As our main test case, we used a complex $\lambda K$-term representing arithmetical expression $3^3 - (2+2)!$ with Church numerals and factorial defined via Turing's fixed point combinator. The chosen $\lambda$-term was meant to cover most of the corner cases, and it also allowed us to collect comprehensive benchmarks. For this test case, the total number of interactions was 2652687 out of which 2621262 are related to oracle nodes. The number of interactions related to the waiting construct was 1182981 out of which 1159057 interactions were against oracle nodes. In other words, the waiting construct occupied less than half of interactions, and the vast majority of its overhead is due to unoptimized oracle representation.

   As shown in Section 5, the waiting construct propagates through the whole body of every abstraction applied to an argument. If the waiting construct were modified to interact with oracle nodes in a more sophisticated way than just propagating through them, it could result in a run-time optimizer for oracle nodes and might possibly solve a long-standing issue about inefficiency of optimal reduction [8].

# 7    Read-back

The process of decoding the normal form of an interaction net into the corresponding $\lambda$-term is called read-back. Usually, read-back is described in prose, and only facilities external with respect to interaction nets are available. Here we embed the read-back mechanism into interaction nets themselves. Essentially, we dissolve application and abstraction agents into textual representation of the corresponding $\lambda$-term.

   Through the rest of this section we will use the following notations: we denote the set of all $\lambda K$-terms as $\Lambda$, and $C[\ ]$ means a context, i. e. a $\lambda$-term with one hole, while $C[M]$ is the result of placing $M$ in the hole of the context $C[\ ]$.

   In order to add the read-back mechanism to our interaction system, we further extend its signature:

$$\Sigma = \Sigma_W \cup \{\top\} \cup \{a_M \mid M \in \Lambda\} \cup \{r_{C[\ ]} \mid C[\ ] \text{ is a context}\},$$

with $\mathsf{ar}(a_M) = 0$ and $\mathsf{ar}(r_{C[\ ]}) = \mathsf{ar}(\top) = 1$, the *atom* agent $a_M$ encoding the textual representation of a $\lambda$-term $M$ and the *read* agent $r_{C[\ ]}$ performing read-back in the context of $C[\ ]$. In particular, agents $a_M$ make it possible to represent free variables in a given $\lambda$-term being encoded into interaction nets. Note that our $\Sigma \setminus \Sigma_W$ extension is infinite as there has to be an agent for every $\lambda$-term and every context.

   Let us recall that the original encoding in [2, pp. 41-42] assumes no free variables in a $\lambda$-term being mapped into its initial encoding. In order to allow free variables, we need to extend the mapping. Specifically, while encoding $\lambda$-terms into our interaction system, we will distinguish their free variables from their bound variables. So, let us mark all free variables in a $\lambda$-term $M$ using the following operation: $M^{\bullet} \equiv M[\vec{x} := \vec{x}^{\bullet}]$, where $(\vec{x}) = \mathsf{FV}(M)$. Then, $\lambda$-term $M$ can be mapped to configuration

$$\langle x \mid \mathsf{eval}(r_{[\ ]}(\top(x))) = y,\ [M^{\bullet}, y]\rangle,$$

where the original encoding $[M, x]$ is extended with $[x^{\bullet}, y] = \{a_x = y\}$.
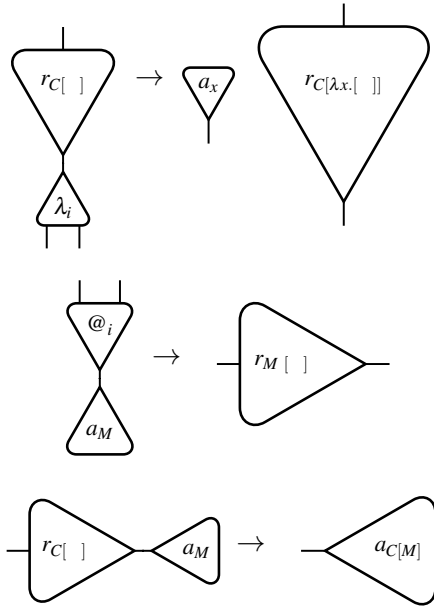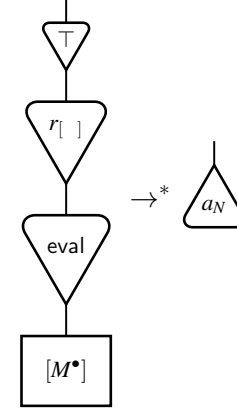
Figure 7: Read-back



Figure 8: Conjecture



Our read-back mechanism mainly consists of the three interaction rules that are shown in Figure 7. More formally in the interaction calculus, the rules related to read-back are as follows:

$$r_{C[\ ]}[x] \bowtie \lambda[a_y, r_{C[\lambda y.[\ ]]}(x)], \quad \text{where } y \text{ is fresh};$$
$$@_i[r_{M[\ ]}(x), x] \bowtie a_M;$$
$$r_{C[\ ]}[a_{C[M]}] \bowtie a_M;$$
$$r_{C[\ ]}[\cap_i(x)] \bowtie \cap_i[r_{C[\ ]}(x)];$$
$$r_{C[\ ]}[\sqcup_i(x)] \bowtie \sqcup_i[r_{C[\ ]}(x)];$$
$$r_{C[\ ]}[\mathsf{wait}(x,y)] \bowtie \mathsf{wait}[r_{C[\ ]}(x), y];$$
$$\mathsf{eval}[a_M] \bowtie a_M;$$
$$\cap_i[a_M] \bowtie a_M;$$
$$\sqcup_i[a_M] \bowtie a_M;$$
$$\top[a_M] \bowtie a_M;$$
$$\top[x] \bowtie \cap_i[\top(x)];$$
$$\top[x] \bowtie \sqcup_i[\top(x)].$$

Now, we believe that the following statement holds true. It is still missing a formal proof. However, no counterexamples have been found while experimenting with software implementation.

**Conjecture.** $\langle x \mid \mathsf{eval}(r_{[\ ]}(\top(x))) = y, \ [M^\bullet, y]\rangle \downarrow \langle a_N \mid \varnothing\rangle$ *iff $N$ is the normal form of $M$.*

That is, the interaction net that encodes a $\lambda$-term $M$ in our interaction system reduces to normal form (if any) with no garbage and only one agent $a_N$ in its interface, $N$ representing the normal form of the encoded $\lambda$-term $M$; see Figure 8.

# 8   Conclusion and further work

To the best of our knowledge, optimal reduction had not yet been implemented using the approach of token-passing nets until this paper. While our implementation of optimal reduction is rather a preliminary result, we still considered it worth sharing. Also, we are not aware of any other existing implementations of read-back embedded into interaction nets and we covered it here in much more details than [11].

   In our opinion, further work should be directed to obtaining a proof for the conjecture given above if the latter holds true. We believe that a proof may be achieved by verifying the following four aspects: non-interference with the original optimal reduction algorithm, the eval agents only triggering needed redexes, no possible deadlocks, and correctness of read-back. Additionally, one needs to quantify how well our implementation preserves the parallelism of Lamping's optimal algorithm. Finally, upgrading the waiting construct into optimization mechanism for oracle nodes and researching its efficiency looks promising for optimal implementations of functional programming languages.

# References

[1]  José Bacelar Almeida, Jorge Sousa Pinto & Miguel Vilaça (2008):  *Token-passing nets for functional languages*.   *Electronic Notes in Theoretical Computer Science* 204, pp. 181–198, doi:10.1016/j.entcs.2008.03.061.

[2]  Andrea Asperti & Stefano Guerrini (1998): *The optimal implementation of functional programming languages*. Cambridge Tracts in Theoretical Computer Science No. 45, Cambridge University Press.

[3]  Maribel Fernández & Lionel Khalil (2003): *Interaction Nets with McCarthy's Amb: Properties and Applications*. *Nordic J. of Computing* 10(2), pp. 134–162.

[4]  Maribel Fernández & Ian Mackie (1999): *A calculus for interaction nets*. In: *Principles and Practice of Declarative Programming*, Springer, pp. 170–187, doi:10.1007/10704567.

[5]  Maribel Fernández, Ian Mackie & François-Régis Sinot (2005):  *Closed reduction:  explicit substitutions without α-conversion*.   *Mathematical Structures in Computer Science* 15(2), pp. 343–381, doi:10.1017/S0960129504004633.

[6]  Yves Lafont (1990): *Interaction nets*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 95–108, doi:10.1145/96709.96718.

[7]  John Lamping (1990): *An algorithm for optimal lambda calculus reduction*.  In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 16–30, doi:10.1145/96709.96711.

[8]  Julia L Lawall & Harry G Mairson (1996): *Optimality and inefficiency: what isn't a cost model of the lambda calculus?*  In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96), Philadelphia, Pennsylvania, May 24–26, 1996*, Pearson Education, p. 92, doi:10.1145/232627.232639.

[9]  Jean-Jacques Lévy (1978): *Réductions correctes et optimales dans le lambda-calcul*. Ph.D. thesis.

[10]  Ian Mackie (2005): *Encoding strategies in the lambda calculus with interaction nets*. In: *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science 4015, Springer, pp. 19–36, doi:10.1007/11964681.

[11]  Anton Salikhmetov (2015): *Macro Lambda Calculus*. Available at `http://arxiv.org/abs/1304.2290`.

[12]  François-Régis Sinot (2005): *Call-by-name and call-by-value as token-passing interaction nets*. In: *Typed Lambda Calculi and Applications*, Springer, pp. 386–400, doi:10.1007/11417170.

[13]  François-Régis Sinot (2006): *Token-passing nets: Call-by-need for free*. *Electronic Notes in Theoretical Computer Science* 135(3), pp. 129–139, doi:10.1016/j.entcs.2005.09.027.