An interaction net encoding of Gödel's System $\mathcal{T}$

Declarative Pearl

Ian Mackie and Shinya Sato

Abstract. The graph rewriting system of interaction nets has been very successful for the implementation of the lambda calculus. In this paper we show how the ideas can be extended and simplified to encode Gödel's System $\mathcal{T}$—the simply typed λ-calculus extended with numbers. Surprisingly, using some results about System $\mathcal{T}$, we obtain a very simple system of interaction nets that is significantly more efficient than a direct encoding for the evaluation of programs.

1 Introduction

Gödel's System $\mathcal{T}$ [7] is the simply typed λ-calculus, with functions and product types, extended with natural numbers. It is a very simple system, yet has enormous expressive power—well beyond that of primitive recursive functions.

Interaction nets [9] are a model of computation, based on graph rewriting. They are user defined rewrite systems and because we can write systems which correspond to term rewriting systems we can see them as specification languages. But, because we must also explain all the low-level details (such as copying and erasing) then we can see them as a low-level operational semantics or more specifically, as an implementation language. Supporting this latter point, we remark that in general graph rewriting, locating (by graph matching) a reduction step is considered an expensive operation, but in interaction nets there is a very simple mechanism to locate a redex (called an active pair in interaction net terminology), and there is no need to use expensive matching algorithms. There are interesting aspects of interaction nets for parallel evaluation—we will hint at some of these aspects later in the paper.

Over the last years there have been several implementations of the λ-calculus using interaction nets. These include optimal reduction [8], encodings of existing strategies [12, 15], and new strategies [13, 14]. One of the first algorithms to implement Lévy's [11] notion of optimal reduction for the λ-calculus was presented by Lamping [10]. Asperti et al. [3] devised BOHM (Bologna Optimal Higher-Order Machine) building on the ideas of Lamping.

The purpose of this paper is to add to this list of interaction net implementations and to bring together on one hand the successful study of encoding λ-calculus and related systems into interaction nets mentioned above, together with the result that Gödel's System $\mathcal{T}$ can be encoded with the linear λ-calculus and an iterator [2]. Specifically, there are redundancies in System $\mathcal{T}$—copying and erasing can be done either by the iterator or by the λ-calculus. We can remove the copy and erasing power of the λ-calculus, and still keep the expressive

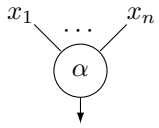
power. Taking this further, we can also get primitive recursive functions as a subset of this system. The key motivation for bringing these works together is that the linear λ-calculus can be very easily encoded into interaction nets, and therefore there is a hope for a very efficient implementation of this language.

The rest of this paper is structured as follows. In the next section we recall the basic notations of interaction nets, to fix notation, and also give the definition of linear System $\mathcal{T}$. In Section 3 we give a compilation of the calculus into interaction nets and give the dynamics of the system together with some examples. In Section 4 we discuss some aspects of this work, and finally we conclude in Section 5.

2 Background

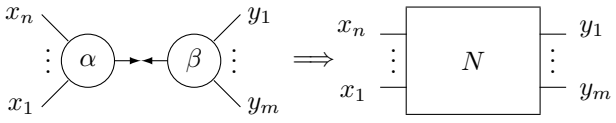
2.1 Interaction nets

In the graphical rewriting system of interaction nets [9], we have a set Σ of *symbols*, which are names of the nodes in our diagrams. Each symbol has an arity ar that determines the number of *auxiliary ports* that the node has. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n + 1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.



Nodes are drawn as circles. A *net* built on Σ is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.

Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction nets analogue of a redex. A rule $((\alpha, \beta) \Longrightarrow N)$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of nodes. The following diagram illustrates the idea, where N is any net built from Σ.



The most powerful property of this graph rewriting system is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length (in fact they are permutations). This has practical consequences: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer. We write $\Longrightarrow$ for a single interaction, and $\Longrightarrow^*$ for the transitive reflexive closure. An interaction net is in normal form

when there are no active pairs. The notation $N \Downarrow N'$ indicates that there exists a finite sequence of interactions $N \Longrightarrow^* N'$ such that N' is a net in normal form. Thus N is (strongly) normalising if $N \Downarrow N'$.

2.2 System $\mathcal{T}$

In this section we recall the main notions of Gödel's System $\mathcal{T}$. This is an applied λ-calculus with, in addition to function types, products and natural numbers. Intuitively, we can think of it as a minimal higher-order language that is an extension to the simply typed λ-calculus. From an alternative perspective, it is a language that has greater computational power than primitive recursive functions—we can define Ackermann's function for instance.

We refer the reader to [7] for a detailed description of System $\mathcal{T}$. In [2] it was shown that there are redundancies in this calculus: copying and erasing can be done either in the λ-calculus or using the iterator. This leads to a simplified presentation using the linear λ-calculus. In this paper we refine the calculus further by introducing pattern matching. There is nothing deep in this step, but it allows us to present the same computational power as System $\mathcal{T}$ in a very precise syntax. In the following we assume familiarity with the λ-calculus [4], and also some basic recursion theory.

Table 1 summarises the syntax of this linear System $\mathcal{T}$. The first four lines give the linear λ-calculus with pairs. The construct $\lambda p.t$ is the usual abstraction, extended to allow patterns of variables or pairs of patterns (as defined at the bottom of the table). The remaining three rules define the syntax for constructing numbers and the iteration. We work with terms modulo α-conversion as usual.

The pattern notation requires a little explanation. In the term $\lambda p.t$, if the pattern p is a variable, say x, then we have the usual abstraction. However, we allow richer patterns built from pairs. It is through these patterns that we are able to access the components of the pairs constructed in the syntax (so we do not need explicit projection functions). Thus we can write terms such as $\lambda\langle x, y\rangle.t$, $\lambda\langle x, \langle y, z\rangle\rangle.t$, etc. Because the language is typed, we will always have arguments of the correct shape for the pattern matching.

In Figure 1 we give the (linear) typing rules for this calculus. We write judgements as $p_1 : A_1, \ldots, p_n : A_n \vdash t : B$. The typing rules also capture the linear variable constraints in an alternative way.

Our version of linear System $\mathcal{T}$ has a number of useful properties: it is confluent, strongly normalising and reduction preserves types. Reduction also preserves the variable constraints, and is adequate to give normal forms for programs of type nat. We first define reduction, then give explanations below.

Definition 1 (Reduction). *The main reduction rules for this calculus are given in the following table:*

Reduction		*Condition*
$(\lambda p.t)v$	$\longrightarrow [p \ll v].t$	$\mathsf{fv}(\lambda p.t) = \varnothing$
$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$	$\longrightarrow \mathsf{iter}\ t\ (vu)\ v$	$\mathsf{fv}(v) = \varnothing$
$\mathsf{iter}\ 0\ u\ v$	$\longrightarrow u$	$\mathsf{fv}(v) = \varnothing$

Context

$$\frac{}{x : A \vdash x : A}\ (\mathsf{Var}) \qquad \frac{\Gamma, p : A, q : B \vdash t : C}{\Gamma, \langle p, q\rangle : A \otimes B \vdash t : C}\ (\mathsf{Pattern\ Pair})$$

$$\frac{\Gamma, p : A, q : B, \Delta \vdash t : C}{\Gamma, q : B, p : A, \Delta \vdash t : C}\ (\mathsf{Exchange})$$

Logical Rules:

$$\frac{\Gamma, p : A \vdash t : B}{\Gamma \vdash \lambda p.t : A \multimap B}\ (\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash t : A \multimap B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}\ (\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u\rangle : A \otimes B}\ (\mathsf{Pair})$$

Numbers:

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{S}\ t : \mathsf{nat}}\ (\mathsf{Succ})$$

$$\frac{\Gamma \vdash t : \mathsf{nat} \quad \Delta \vdash u : A \quad \Theta \vdash v : A \multimap A}{\Gamma, \Delta, \Theta \vdash \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Iter})$$

Fig. 1. Linear System $\mathcal{T}$

Terms	**Variable Constraint**	**Free Variables** (fv)
x	$-$	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda p.t$	$\mathsf{bv}(p) \subseteq \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \mathsf{bv}(p)$
$\langle p, q\rangle$	$\mathsf{fv}(p) \cap \mathsf{fv}(q) = \varnothing$	$\mathsf{fv}(p) \cup \mathsf{fv}(q)$
0	$-$	$\varnothing$
$\mathsf{S}\ t$	$-$	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ v$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \varnothing$ $\mathsf{fv}(t) \cap \mathsf{fv}(v) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v)$
Pattern	**Variable Constraint**	**Bound Variables** (bv)
x	$-$	$\{x\}$
$\langle p, q\rangle$	$\mathsf{bv}(p) \cap \mathsf{bv}(q) = \varnothing$	$\mathsf{bv}(p) \cup \mathsf{bv}(q)$

Table 1. Terms

The conditions on the rules are used to constrain the possible reductions and preserve the linearity of the terms. The matching operation $[p \ll u].t$ is inspired by that of the ρ-calculus [5]. $\lambda p.t$ is a generalised abstraction—it can be seen as a λ-abstraction on a pattern p instead of a single variable. $[p \ll u].t$ is a matching constraint *denoting a matching problem $p \ll u$ whose solutions will be applied to t. The reduction rules for the matching construct are:*

$$\begin{array}{ll} [x \ll v].t & \longrightarrow t[v/x] \\ [\langle p, q\rangle \ll \langle t, u\rangle].t & \longrightarrow [p \ll t].[q \ll u].t \end{array}$$

Thus matching creates substitutions. Substitution is a meta-operation defined as usual, and reductions can take place in any context. Matching forces evaluation of terms, and will always succeed.

This calculus has a number of properties: it is terminating and confluent, and reduction preserves types. We will not give an extensive study of those properties here however, as we are interested in implementing this calculus in interaction nets. We will simply give an important property of reduction that will be essential to prove any results about the encoding later. Define $t \Downarrow n$ if $t \longrightarrow^* u$ and u is a normal form (*i.e.*, no further reduction is possible).

Lemma 1. *Let t be a closed linear System $\mathcal{T}$ term. If $t \Downarrow u$ then exactly one of the following occurs:*

1. *if $t : \mathsf{nat}$, then $u = S^n(0)$.*
2. *if $t : A \otimes B$, then $u = \langle v, w\rangle$, for some terms v and w.*
3. *if $t : A \multimap B$, then $u = \lambda x.v$, for some term v.*

Proof. We show the case for nat. Since t is closed and reduction preserves types, u can only be a number, an application or an iter construct. If u is an application, say $(\lambda x.a)b$, then since u is closed, $(\lambda x.a)$ must also be closed, and therefore it is not a normal form as a reduction can take place (contradiction). If u is an iter, say $\mathsf{iter}\ n\ a\ b$, then since u is closed, n must also be closed, and therefore it is not a normal form as a reduction can take place (contradiction). Therefore, u must be a number. The other two cases follow similar reasoning.

2.3 Examples

Here we give a several examples to illustrate how to use the syntax and what terms look like.

– Pairs and pattern matching:

$$\begin{array}{l} \lambda\langle x, y\rangle.\langle y, x\rangle : A \otimes B \multimap B \otimes A \\ \lambda\langle x, \langle y, z\rangle\rangle.\langle\langle x, y\rangle, z\rangle : A \otimes (B \otimes C) \multimap (A \otimes B) \otimes C \end{array}$$

– Addition, multiplication and exponentiation can be defined as:

$$\begin{aligned}\mathsf{add} &= \lambda mn.\mathsf{iter}\ m\ n\ (\lambda x.\mathsf{S}x) : \mathsf{nat} \multimap \mathsf{nat} \multimap \mathsf{nat}\\ \mathsf{mult} &= \lambda mn.\mathsf{iter}\ m\ 0\ (\mathsf{add}\ n) : \mathsf{nat} \multimap \mathsf{nat} \multimap \mathsf{nat}\\ \mathsf{exp} &= \lambda mn.\mathsf{iter}\ n\ (\mathsf{S}\ 0)\ (\mathsf{mult}\ m) : \mathsf{nat} \multimap \mathsf{nat} \multimap \mathsf{nat}\end{aligned}$$

  Note in particular that each function satisfies the linearity constraints.
– When we need to copy or erase, we can do that as shown in the following examples for numbers:

$$\begin{aligned}C &= \lambda x.\mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda\langle a,b\rangle.\langle \mathsf{S}a, \mathsf{S}b\rangle) : \mathsf{nat} \multimap \mathsf{nat} \otimes \mathsf{nat}\\ \mathsf{fst} &= \lambda\langle n,m\rangle.\mathsf{iter}\ m\ n\ (\lambda x.x) : \mathsf{nat} \otimes \mathsf{nat} \multimap \mathsf{nat}\\ \mathsf{snd} &= \lambda\langle n,m\rangle.\mathsf{iter}\ n\ m\ (\lambda x.x) : \mathsf{nat} \otimes \mathsf{nat} \multimap \mathsf{nat}\end{aligned}$$

– Ackermann's function is a standard example of a non primitive recursive function:

$$\begin{aligned}ack(0,n) &= \mathsf{S}\ n\\ ack(\mathsf{S}\ n,0) &= ack(n,\mathsf{S}\ 0)\\ ack(\mathsf{S}\ n,\mathsf{S}\ m) &= ack(n,ack(\mathsf{S}\ n,m))\end{aligned}$$

  In a higher-order functional language, there is an alternative definition that we can write in our syntax:

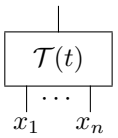
$$\mathsf{ack} = \lambda m.\lambda n.(\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x))n$$

  We can simplify this definition slightly by using the usual η-rule: $\lambda x.tx = t$, (we remark that $x \notin \mathsf{fv}(t)$ because of the linearity constraint).

$$\mathsf{ack} = \lambda m.\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x)$$

3 Interaction net encoding

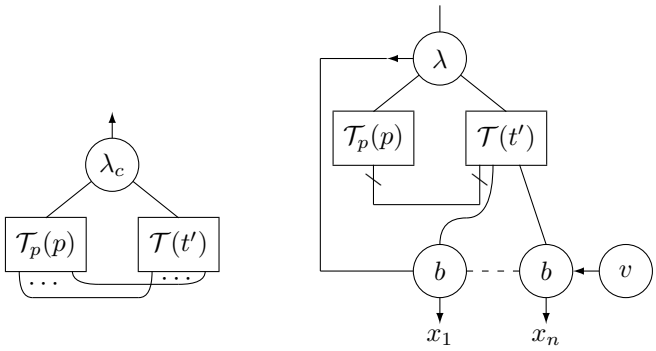
In this section we give a translation $\mathcal{T}(\cdot)$ of linear System $\mathcal{T}$ into interaction nets. A term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated as a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables:



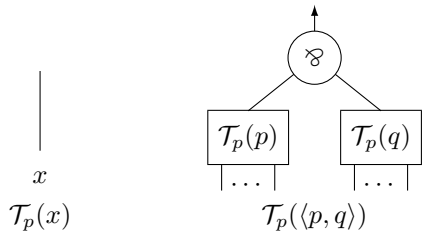
The labelling of free edges is just for the translation (and convenience), and is not part of the system. The nodes needed for this compilation will be introduced on demand, and we give the interaction rules later in the section. We will occasionally make some assumptions about the order of the free edges to make the diagrams simpler below.

Variable. When t is a variable, say x, then $\mathcal{T}(t)$ is translated into an edge:

Abstraction. If t is an abstraction, say $\lambda p.t'$, then there are two alternative translations of the abstraction, which are given as follows:



In these diagrams, we use an auxiliary function for the translation of patterns $\mathcal{T}_p(p)$ which is given by the following two rules.



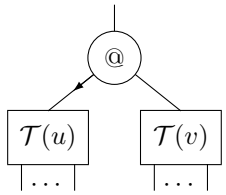
If p is a variable, then it is translated into an edge. Otherwise, if it is a pair pattern, then it is translated as shown in the right hand diagram above.

Returning to the compilation of abstraction, in the first case, shown on the left in the above diagram, is when $\mathsf{fv}(\lambda p.t') = \varnothing$. Here we use a node λ_c to represent a *closed abstraction* and we explicitly connect the occurrence of the variable of the body of the abstraction to the λ_c node.

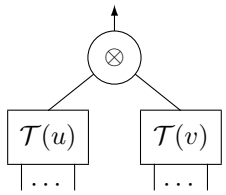
The second case, shown on the right, is when $\mathsf{fv}(\lambda p.t') = \{x_1, \ldots, x_n\}$. Here we introduce three different kinds of node: λ of arity 3, for abstraction, and two kinds of node representing a list of free variables. A node b is used for each free variable, and we end the list with a node v. The idea is that there is a pointer to the free variables of an abstraction; the body of the abstraction is encapsulated in a box structure. We assume, without loss of generality, that the (unique) occurrence of the variable x is in the leftmost position of $\mathcal{T}(t')$.

It is worth noting that a closed term will never become open during reduction, but crucially for this system to work, terms may become closed during reduction. The distinction between open and closed terms is important in the dynamics of the interaction system that is given later.

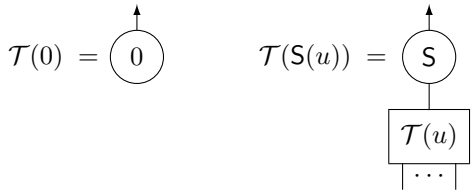
Application. If t is an application, say uv, then $\mathcal{T}(uv)$ is given by the following net, where we have introduced a node @ of arity 2 corresponding to an application.



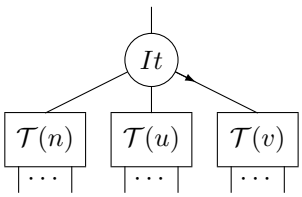
Pair. If t is a pair, say $\langle u, v\rangle$, then $\mathcal{T}(\langle u, v\rangle)$ is given by the following net, where we have introduced a node $\otimes$ of arity 2 corresponding to a pair.



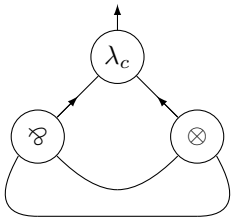
Numbers. A number will be represented by a chain of successor nodes (S), terminating with a zero (0) node. S has one auxiliary port, and 0 has none. Therefore, if t is a number, it is either 0 or $\mathsf{S}(u)$, for some term u. These two cases are translated as follows:



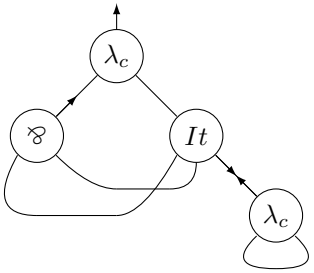
Iterator. If t is $\mathsf{iter}\ n\ u\ v$, then we introduce one new node. The principal port of this node points to the function v, because we must wait for this to become a closed term before starting the interaction process.



This completes the compilation function. A closed term will be translated as a net with one edge at the root of the term. We give some examples before defining the reduction rules for the interaction nodes that we introduced in the above compilation. The first example is the net $\mathcal{T}(\lambda\langle x, y\rangle.\langle y, x\rangle)$, which illustrates the pattern and pairing construct:



The second example shows the function snd defined previously. The compilation $\mathcal{T}(\lambda\langle m, n\rangle.\mathsf{iter}\ m\ n\ (\lambda x.x))$ gives the following net:



The final example, given in Figure 2 is the net corresponding to the Ackermann function: $\mathcal{T}(\lambda m.\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x))$.

3.1 Reduction

We next give the rules to complete the interaction net system. In Figure 3 we give the first seven interaction rules that encode β-reduction, pattern matching and substitution. The first rule starts the implementation of β-reduction, connecting the body of the abstraction to the result, and the argument becomes a substitution. The second rule implements the matching. The remaining rules propagate a substitution through a net, and an important rule is v interacting with λ, where a closed abstraction is created.

The rules so far are similar to some other interaction net systems for the λ-calculus, so could be considered standard. The next three rules implement the iterator operation that we explain in more detail. When the iterator node interacts with a closed abstraction we have the following rule:

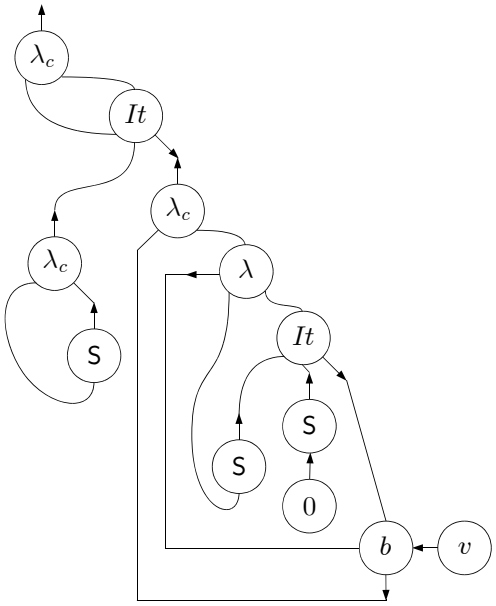
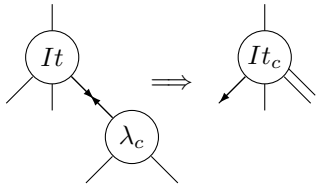
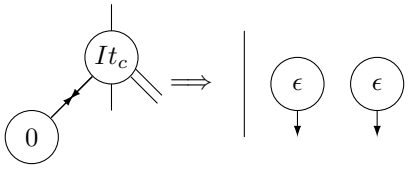


Fig. 2. Ackermann function



This rule creates a new node It_c that will interact with numbers. The node also holds on to the body and the variable edge of the abstraction. The two rules for the It_c node are as follows. The first rule is when we erase the function, and connect the result to the base value.



The final rule is when we unfold one level of iteration. Here the function is duplicated with δ nodes, and one copy is applied to the base value as required. Because the function being duplicated is closed, the duplication process is easily proved to be correct.

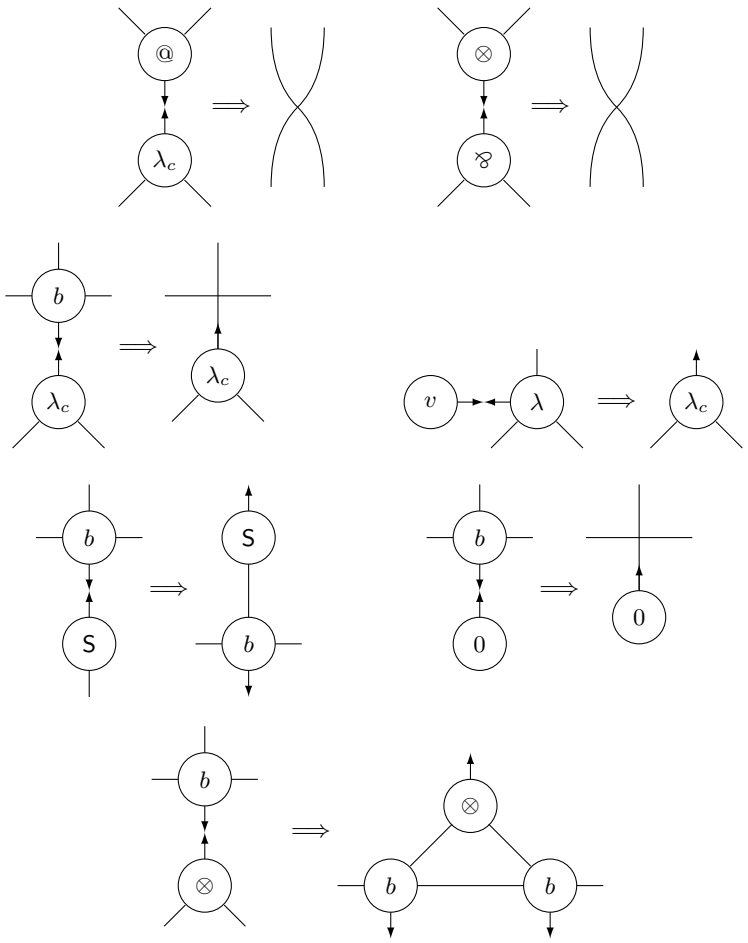
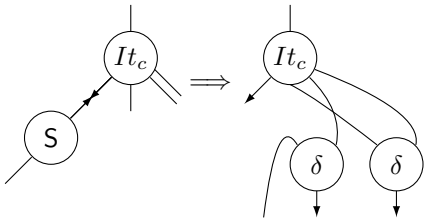


Fig. 3. Interaction Rules



In Figure 4 we give the final rules for duplication and erasing, where we use α to range over all other nodes in the system.

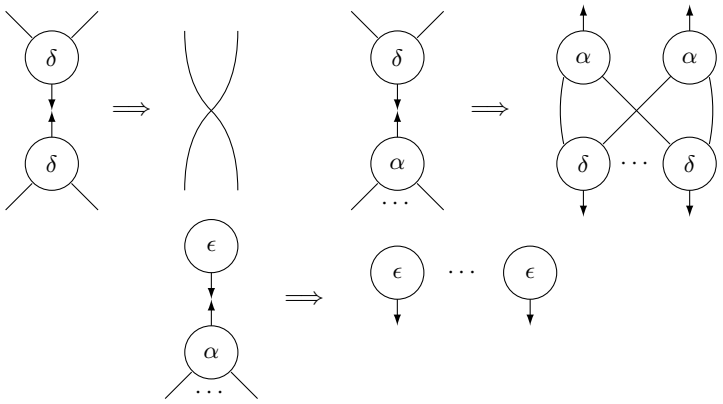


Fig. 4. Duplication and Erasing Interaction Rules

These rules are all that we need to implement our linear System $\mathcal{T}$. By showing that we simulate the reduction rules we get the following result.

Theorem 1. *Let* $t : \mathsf{nat}$ *be a closed linear System* $\mathcal{T}$ *term with normal form* u*, then* $\mathcal{T}(t) \Downarrow \mathcal{T}(u)$*.*

It is possible to give an encoding of Gödel's System $\mathcal{T}$ directly to interaction nets, however this comes at a cost as we must incorporate the non-linear aspects of the λ-calculus: copying and erasing. With our encoding, we have isolated the copying and erasing to closed functions, which is a much simpler operation (and we do not need many of the so-called bookkeeping operations of the general systems). Thus, using a result from [1], stating that the linear version is as powerful as the non-linear version, gives a greatly simplified interaction system.

Moreover, we can go further. Using a result of Dal Lago [6]: if we take the linear λ-calculus where iterated functions must be *closed by construction* (*i.e.*, $\mathsf{fv}(v) = \varnothing$ in $\mathsf{iter}\ t\ u\ v$) then this system captures exactly the primitive recursive functions. If we are building functions to iterate that must be closed by construction, then we no longer need the box structure to identify when a term becomes

closed. The consequence of this result here is that we can eliminate b, v and λ nodes (and the associated interaction rules), so that λ_c and @ are sufficient to encode the linear λ-calculus.

Theorem 2. *An interaction system built from the nodes* 0, S, *It,* It_c, δ, ϵ, λ_c, *and* @ *is complete for primitive recursive functions.*

The encoding of the linear λ-calculus as a system of interaction nets is particularly simple, since substitution is implemented for free: β-reduction is a constant time operation. This is a consequence of the fact that substitution is essentially implemented as an assignment. What we have achieved therefore is a very simple, with no overheads, implementation of Gödel's System $\mathcal{T}$ and primitive recursive functions in interaction nets.

4 Discussion

Very few people write programs with unary arithmetic (zero and successor). Nevertheless, the same techniques are used to represent lists and other data-structures that are ubiquitous. All our results can be adapted to work for a version of System $\mathcal{T}$ with built-in numbers (and also richer data-types), together with operations that work directly with these numbers. Our belief is that to understand complex languages and make them efficient, it is fruitful to start with simple subsets and build up. This is the approach we have taken in this paper.

Implementations of the λ-calculus are made complicated by the non-linear aspects of the calculus. Using the linear λ-calculus with iterators gives a simpler formulation of many algorithms, and even simpler when the iterated function is closed. It is possible to use compilation techniques to transform a non-linear algorithm in System $\mathcal{T}$ to our linear version, and in addition find ways to close functions. This approach uses some standard ideas from compilation such as continuations, but applied to a linear setting. We hope to report on some of these details in a future work, and also the impact on implementation efficiency.

Finally, we mention that there are a number of parallel implementations of interaction nets, and consequently the system presented in this paper can directly take advantage of this. However, some choices in the encoding can give different results: the most efficient sequential system is not necessarily the best system to try to run on parallel hardware. Again, we hope to be able to understand this aspect better through extensive benchmark testing.

5 Conclusion

We have given a very simple and efficient implementation of Gödel's System $\mathcal{T}$ using the graph rewriting formalism of interaction nets. The aim of this paper was initially to apply some of the ideas used for the representation of the λ-calculus in interaction nets to the the linear version of System $\mathcal{T}$ to investigate if the resulting system provides a useful implementation technique.

This work is a building block in a larger programme of research to investigate when interaction nets are useful for the evaluation of programs (either because they are more efficient than standard techniques, of if they offer some other advantage such as parallelism, small run-time system, etc.). A first step in this direction is the extension of the language with data-types, in particular lists, and investigate if other algorithms can also benefit from the techniques used here.

References

1. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In Z. Ésik, editor, *Proceedings of the 15th EACSL Conference on Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.
2. S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system T revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.
3. A. Asperti, C. Giovannetti, and A. Naletto. The Bologna Optimal Higher-order Machine. *Journal of Functional Programming*, 6(6):763–810, Nov. 1996.
4. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.
5. H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.
6. U. Dal Lago. The geometry of linear higher-order recursion. In P. Panangaden, editor, *Proceedings of the 20th Annual IEEE Symposium on logic in Computer Science, LICS 2005*, pages 366–375. IEEE Computer Society Press, June 2005.
7. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
8. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.
9. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
10. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, Jan. 1990.
11. J.-J. Lévy. Optimal reductions in the lambda calculus. In J. P. Hindley and J. R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
12. S. Lippi. λ-calculus left reduction with interaction nets. *Mathematical Structures in Computer Science*, 12(6), 2002.
13. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.
14. I. Mackie. An interaction net implementation of closed reduction. In S.-B. Scholz and O. Chitil, editors, *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2008.
15. F.-R. Sinot. Call-by-name and call-by-value as token-passing interaction nets. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.