An Encoding of Interaction Nets in OCaml

Nikolaus Huber
Uppsala University
nikolaus.huber@it.uu.se

Wang Yi
Uppsala University
wang.yi@it.uu.se

Interaction nets constitute a visual programming language grounded in graph transformation. Owing to their distinctive properties, they inherently facilitate parallelism in the rewriting step. This paper showcases a simple and concise approach to encoding interaction nets within the programming language OCaml, emphasising correctness guarantees. To achieve this objective, we encode not only the interaction net primitives, but also Lafont's original type system.

1 Introduction

Interaction nets, introduced by Lafont in [12], are a model of computation based on graph rewriting. They have, among other things, been used as a basis for optimal [3, 13] and efficient implementations [14] of the λ-calculus. Interaction nets offer a number of desirable properties, such as locality of reduction, strong confluence, and Turing completeness. Owing to the locality of reduction, highly parallel implementations can be devised, while the confluence guarantees determinism. They have been studied both as a model for the dynamics of computation and also as a programming language itself. However, only a few implementations are still actively maintained.

In this paper, we study a method of encoding interaction nets into the general purpose programming language OCaml. Unlike previous efforts, our focus is not on the efficiency of reduction, but rather on embedding Lafont's original type system to provide stronger correctness guarantees. By utilising advanced type system features of the host language, we can statically enforce Lafont's typing discipline without relying on dynamic checks.

To the best of our knowledge, this is the first encoding of interaction nets in OCaml, and the first attempt at embedding Lafont's type system into the type system of another language. Given OCaml's recent addition of native support for expressing parallelism, we therefore also explore whether interaction nets can serve as a general implementation scheme for parallel algorithms in the language.

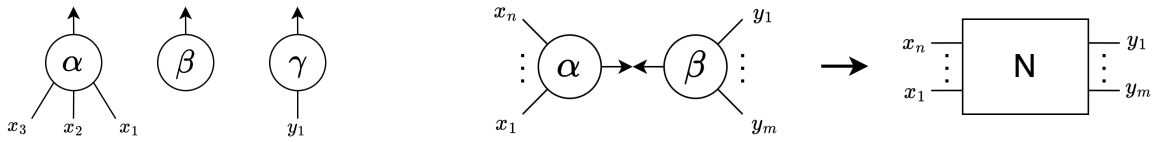
The structure of this paper is as follows: In Section 2 we give an overview of the model of interaction nets together with the graphical notation we will be using throughout the paper, followed by a brief introduction to OCaml and an overview of previous work. In Section 3 we illustrate our method of encoding interaction nets into OCaml by way of showing a concrete example. In Section 4 we look at the runtime behaviour of different encoded nets, and investigate if they can make use of OCaml's support for parallelism. Finally, we give directions for future work and conclude in Section 5.

2 Preliminaries

2.1 Interaction nets

We briefly recall the basic notions of interaction nets. For a more complete presentation, the interested reader is referred to [12]. Interaction nets consist of the following elements:

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 1–16, doi:10.4204/EPTCS.417.1



(a) Example agents

(b) Example rule

Figure 1: Graphical notations for agents and rules

- A set Σ of *symbols*. These symbols are used as labels for nodes, which are referred to as *agents*. Each agent has a fixed set of *ports*, through which they connect with other agents. An agent has exactly one distinguished *principle port*, and a (possibly empty) set of *auxiliary ports*. The number of auxiliary ports is fixed for each symbol. We assume the existence of a function $ar : \Sigma \to \mathbb{N}$, such that, for all $\alpha \in \Sigma$, $ar(\alpha)$ is the number of auxiliary ports associated with the symbol α. Figure 1a illustrates the usual graphical notation for agents of three different symbols α, β, and γ. The principle port is depicted as an arrow, auxiliary ports are numbered clockwise from the principle port. In the given example $ar(\alpha) = 3$, $ar(\beta) = 0$, and $ar(\gamma) = 1$.
- A net N is an undirected graph built from agents over the set Σ. Each edge of the graph connects two agent ports together, such that there is only one edge per port. A port that is not connected is called a *free port*. The set of free ports of a net is called its *interface*.
- A set $\mathscr{R}$ of *rules*. A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ is called an *active pair* if they are connected through their principle ports (this is related to the concept of a *redex* in term rewriting). The application of an interaction rule $((\alpha, \beta) \to N) \in \mathscr{R}$ replaces an active pair by a net N in such a way that the interface of N coincides exactly with the auxiliary ports of α and β. Intuitively, this means, that all auxiliary connections of the active pair must be connected to the new net, and no new free ports can be introduced in the process. Figure 1b shows an example of the graphical notation for rules.

Rewriting happens through the application of interaction rules to active pairs. Since every agent only has one distinct principle port, the rewriting process is local, and two different active pairs can be rewritten in parallel. Some additional requirements are necessary to guarantee determinism: For each pair (α, β) there can only be at most one rule in $\mathscr{R}$, and if there is no rule for a specific active pair it cannot be further reduced. Since there is no notion of orientation, rules are symmetric, meaning that $(\alpha, \beta) \to N$ and $(\beta, \alpha) \to N$ describe the same rule. With these restrictions in place it can be shown [12], that the reduction sequence is strongly confluent.

2.2 Typed interaction nets

Already in the original paper [12], Lafont remarks that not all possible connections between agents have a valid semantic interpretation. In the next section, we will introduce agents representing boolean and integer values, as well as agents representing functions over those values. A function agent expecting a particular type at its principle port cannot be reduced when paired with an agent of the wrong type. Therefore, Lafont introduced a rudimentary type system, where each port of an agent is assigned both a *value type*, and a *polarity*. Value types can include integers, booleans, floats, lists, etc.

The polarity encodes if a port is meant to be an input or an output. An example can be seen in Figure 2. It is up to convention if inputs receive positive, or negative polarity, as long as the assignment

is consistent throughout. In this paper, we will follow the same convention as in [12], where inputs have negative, and outputs positive polarity.

Only ports of the same type, but opposite polarity, can be connected to each other. Rules are well typed if their left-hand side is well typed (i.e., the principle ports of the active pair have the same type and opposite polarity), and if the net on the right-hand side is well typed, taking into account the interface types provided by the auxiliary connections of the active pair.

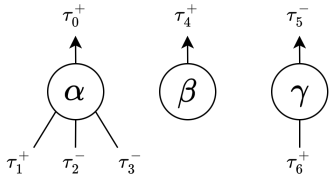


Figure 2: Examples of typed agents

2.3 OCaml

OCaml is an industrial-strength (primarily functional) programming language which arose as an extension of the Caml dialect of the ML language family. While it is a general-purpose language, it has a strong background in theorem proving (e.g., Coq [1]), static analysis (e.g., Frama-C [11]), and formal methods software. Interestingly, even though it is a language used in industry, it only recently added native support for expressing parallel evaluation [20].

The basic unit of parallelism in OCaml is called *domains*. A single domain is created at the start of a program, and more domains can be spawned later on. Domains map directly to operating system threads, and are therefore costly to create and tear down. It is thus recommended to not spawn more domains than cores available on the local processor. While the standard library of OCaml provides the `Domain` module, it only offers low-level primitives for managing domains. This opens the possibility of providing higher-level libraries to be developed outside the core compiler. Different such libraries have been developed, all of them with different design trade-offs and different APIs. In order not to be too restricted by the choice of library, we will show our encoding method against a simplified API, which can be provided by different libraries:

```
type pool
type 'a promise
type 'a resolver
val resolve : 'a resolver -> 'a -> unit
val await : 'a promise -> 'a
val block : 'a promise -> 'a
val make_future : unit -> 'a promise * 'a resolver
val create_pool : int -> pool
val run_async : pool -> (unit -> unit) -> unit
```

Type `pool` describes the pool of worker threads, it is created by a call to `create_pool` with the number of desired workers as an argument. Types `'a promise` and `'a resolver` are two ends of a simple communication primitive. They are always created together by a call to `make_future`. A value

of type `'a promise` acts as a placeholder for a value of type `'a`, which will later be supplied through the corresponding `resolver` by calling the `resolve` function. Function `await` takes a promise and returns its value once it is resolved. Function `block` does the same. However, it blocks the currently running thread until the promise is resolved. Finally, the function `run_async` takes a pool and a continuation, and puts that continuation into the work-queue of the pool for asynchronous execution. Such an API can be provided by different already existing libraries, the one we chose for this paper is called Moonpool [2]. For those unfamiliar with the OCaml language, we give a short overview of the syntax in appendix A.

2.4 Previous work and contributions

A number of different evaluators/compilers for interaction nets have been developed. Some have built upon each other, such as AMINE [17], PIN [4], INET [5], and amineLight [15]. A comparison of these is given in [18], and the most recent and still actively maintained version of this lineage is available in the inpla project [19].

Due to the inherent possibility of parallel reduction, running interaction nets on GPUs was investigated in [8]. The code for the project is still available [7]. However, it has not been updated in 13 years and needs all rules encoded as custom CUDA kernels.

Encoding interaction nets in a functional programming language was done in [10], where interaction nets were embedded into Concurrent Haskell [16]. The code for this evaluator is available [9], but it also has not been updated since 2015, and requires a rather old version of Haskell to still compile.

Our method is closest to the embedding into Haskell. However, we have taken multiple different design choices. In the Haskell encoding, the arity constraint for each label is not enforced by the type system, as ports are encoded as a list of directed references to other agents. In our encoding, not all connections are implemented as references, and utilising a dedicated constructor for each agent the type system guarantees the arity constraints. The Haskell encoding uses polarity to figure out the direction for each reference, i.e., which side will provide the reference, and which side waits until the reference is provided. It relies on a heuristic to figure out the polarity of each port. In this paper, we bypass the inference of polarities, and instead assume that they are given as part of the agent definitions. In the Haskell encoding, polarities are carried around as concrete values, and are dynamically checked. We encode polarities in the type system of OCaml directly, therefore preventing the construction of incorrect nets already during compile time, and removing the need for dynamic checks.

3 Encoding

In this section, we showcase the encoding of the different constituent parts of interaction nets into OCaml. We start by defining a dedicated type for agents, then show how we encode rule application. After introducing agent attributes, we then show how we embedded Lafont's original type system into the type system of OCaml.

3.1 Encoding agents

There are many ways of how agents might be encoded within a given language. In the interaction net to C compiler presented in [5], agents are encoded as a structure, where the first field relates to the label of the respective agent, while the second encodes the ports as an array of references to other agents. The encoding of interaction nets in Haskell presented in [10] uses a similar approach, where each agent is described as a record with a field for the label and a field for the list of connections to other agents.

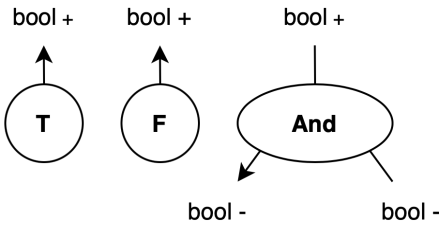


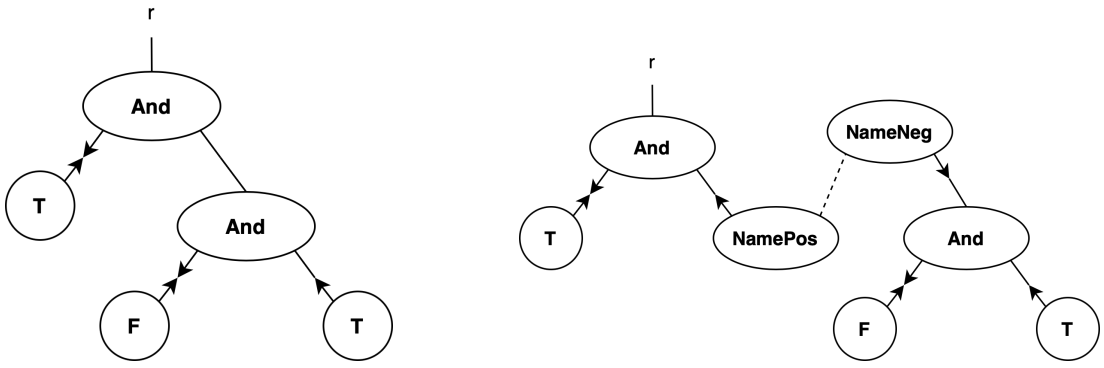
Figure 3: Boolean agents

These connections make use of mutable variables, as introduced by Concurrent Haskell [16]. Indeed, the way that mutual connections of agents are encoded seems to be one of the most important differences between different interaction net evaluators. In his PhD thesis [18], Sato gives a nice overview of these different encodings. According to his nomenclature, our encoding method follows the principle of *single link encoding*, which seems to also be the basis of the Haskell encoding.

For a given interaction net system with a label set Σ, we create a *variant data type* with one constructor case per symbol. As a simple example, we will start by encoding the agents shown in Figure 3:

```
type agent =
    | T
    | F
    | And of agent * agent
```

Each symbol $\alpha \in \Sigma$ is translated to one constructor case in the type `agent`, using α as the label, and carrying $ar(\alpha)$ agents as attributes (encoding the appropriate number of auxiliary ports). When creating a value of type `agent` by using one of these constructors, the resulting value represents the principle port of the respective agent. Every auxiliary port used as an attribute is a connection to the principal port of another agent.



(a) Using common connection

(b) Using name nodes

Figure 4: Connection between auxiliary ports

Nets are created by simply applying constructors recursively to each other. However, encoding a net in this way does not allow for two auxiliary ports to be connected directly, so the net in Figure 4a could not be encoded in this way. To allow the mutual connection of auxiliary ports, we break the link and

create two agents with new symbols `NamePos` and `NameNeg` (see Figure 4b). We utilise the information about the polarity of the connection in order to decide which port needs to be connected to which name agent. We can therefore extend our `agent` type accordingly:

```
type agent =
    | T
    | F
    | And of agent * agent
    | NamePos of agent promise
    | NameNeg of agent resolver
```

We use the same mechanism for creating connections for the interface of the initial network (i.e., the network to be rewritten). We also introduce a function to create these pairs of name agents:

```
let new_name () =
    let promise, resolver = make_future () in
    NamePos promise, NameNeg resolver
```

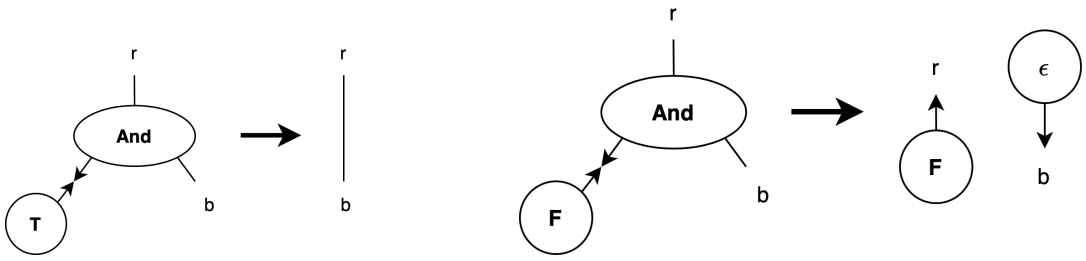
3.2 Encoding rule application

Apart from agents, we also need to encode rules. All rules are collected and translated together into a function `apply_rule`, which internally uses pattern matching to match the different active pairs. So for the two rules shown in Figure 5 we get:

```
let rec apply_rule a1 a2 = match a1, a2 with
    (* using an or-pattern to list both orientations *)
    | T, And (r, b)
    | And (r, b), T          -> b -><- r
    | F, And (r, b)
    | And (r, b), F          -> ignore b; F -><- r
    | NamePos v, a
    | a, NamePos v           -> await v -><- a
    | NameNeg v, a
    | a, NameNeg v           -> resolve v a
    (* match cases must be exhaustive in OCaml *)
    | _, _                   -> failwith "No rule for this pair"

(* custom infix operator to run rewriting asynchronously in thread pool *)
and ( -><- ) a1 a2 = run_async pool (fun _ -> apply_rule a1 a2)
```

Since all rules are symmetric in the constituents of the active pair, we have to list both orientations. OCaml has a shorthand syntax for when two cases have the same right-hand side, as long as we name the attributes of matched constructors the same. The functions `apply_rule` and `-><-` are mutually recursive, the latter uses OCaml's support for custom infix operators and just puts the application of `apply_rule` into the pool of threads to run asynchronously with all other rewritings. While this is not



(a) Rule T →← And

(b) Rule F →← And

Figure 5: Rules for conjunction

strictly necessary, it simplifies the code in the `apply_rule` method considerably. We assume that the variable `pool` has been globally defined at the start of the program. To satisfy the exhaustiveness of the pattern match, we need to include a catch-all case (`_, _`), which just fails with an error message. As we will see in Section 3.5, once we introduce Lafont's typing discipline, this catch-all case will become unnecessary.

The careful reader will have realised that we have not introduced the agent with label ε as shown in Figure 5b. This is a special agent, usually referred to as the *delete agent*, which just recursively deletes the net connected to it. As OCaml is a managed language, we can leave this deletion process to the garbage collector. However, the compiler will complain if we name a port on the left-hand side of a case and not use it on the right-hand side. We therefore use the OCaml function `ignore` to mark that we do not care about a particular value. Another option would have been to use the special `_` pattern in order not to give a name to the unused port:

```
...
    | F, And (r, _)
    | And (r, _), F    -> F ->< r
...
```

3.3 Agent attributes

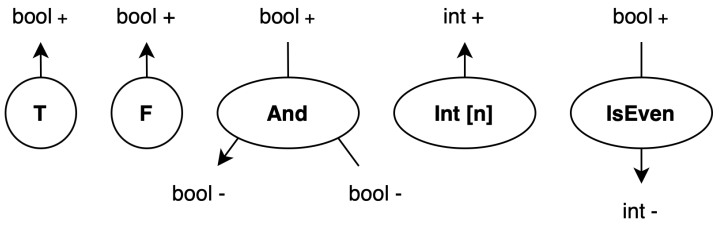


Figure 6: Boolean agents extended with integer agents

Representing booleans with two different labels works fine. However, with numbers such a unary encoding becomes already quite tedious and inefficient. Therefore, we extend our definition of agents, allowing them to carry attributes, i.e., values. As shown in [5], this does not contradict the strong confluence of the reduction. Figure 6 shows an extension of our previous label set with `Int` agents, carrying

a value of their namesake, as well as an agent `IsEven`. Encoding agents with attributes in OCaml is straightforward, as we can just extend the attributes of the respective constructor:

```
type agent =
    | Int of int
    | IsEven of agent
    | T
    | F
    | And of agent * agent
    | NamePos of agent promise
    | NameNeg of agent resolver
```

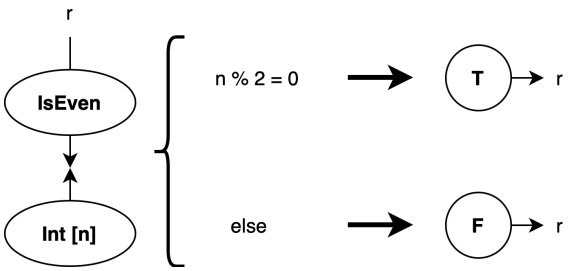


Figure 7: Rule Int[n] →← IsEven

We can use the attributes of the agents on the left-hand side of a rule to define guarded right-hand sides, as shown in Figure 7. Encoding such rules in `apply_rule` is easy, as OCaml allows match cases to be guarded by a boolean expression:

```
let rec apply_rule a1 a2 = match a1, a2 with
    ...
    | IsEven r, Int n
    | Int n, IsEven r when n mod 2 = 0    -> T -><- r
    | IsEven r, Int _
    | Int _, IsEven r                     -> F -><- r
    ...
```

3.4 Type compatibility

With the introduction of integers into our example, we now also have to consider the types of ports. Indeed, the way we have defined agents through an ordinary variant type does not enforce this, so the OCaml compiler will happily accept `And (r, Int 0)` as a valid expression, even though `Int 0` is not a boolean. We could, of course, try and include a deep embedding of the type of an agent by doing something similar to the following:

```
type ty =
    | Int
    | Bool
```

```
type agent =
    | Int of ty * int
    | IsEven of ty * agent
    | T of ty
    | F of ty
    | And of ty * agent * agent
    | ...
```

However, encoding the type in this way incurs an overhead both in execution time and code conciseness, as we would need to include runtime checks at appropriate positions to check for type compatibility.

It would be nicer, if we could track the type of an agent in OCaml's type system directly, in a way that would prevent the construction of such faulty nets in the first place. Luckily, OCaml offers exactly this in the form of *generalised algebraic data types* (GADTs). A GADT is similar to a parameterised variant type, but it allows expressing constraints on the type variable for each constructor case individually. For our example, the agent type can be defined in the following way:

```
type _ agent =
  | Int : int -> int agent
  | IsEven : bool agent -> int agent
  | T : bool agent
  | F : bool agent
  | And : bool agent * bool agent -> bool agent
  | NamePos : 'a agent promise -> 'a agent
  | NameNeg : 'a agent resolver -> 'a agent
```

The `:` after the constructor label indicates the definition of a GADT. The `_` before `agent` is an *anonymous type variable*, as it does not show up in the definition of the type `agent` itself. However, for each constructor case this type variable can be constrained to a concrete type, thereby limiting the allowed agents at specific auxiliary ports. For example, the constructor `And` expresses, that both auxiliary agent connections must be of type `bool`, the principle port has type `bool` as well (which, by lucky syntactic coincidence, follows after the `->`). With this in place, the OCaml compiler will now reject `And (r, Int 0)` as a valid net.

The type checker usually needs additional information when GADTs are paired with recursive functions (such as `apply_rule`). In general, if a recursive function uses pattern matching, the type checker fails to unify the type variable with different types in the different cases. It also does not allow for a recursive call to use different type variables than the outer call. This can be circumvented by providing explicit type annotations that mark the function as polymorphic in the type parameter of the types of its input:

```
let rec apply_rule : type a. a agent -> a agent -> unit =
    fun a1 a2 -> ...

and ( ->><- ) : type a. a agent -> a agent -> unit  =
    fun a1 a2 -> ...
```

Intuitively, the annotation `type a` can be read as a for-all quantifier.

3.5 Polarity

Type compatibility between ports is only one part of Lafont's type system. It does not prevent us from creating active pairs that do not fit together in terms of their polarity, so currently, the compiler will accept `Int 0 ->><- Int 1` as valid. To encode the polarity, we can introduce a second anonymous type variable for the definition of the polarity type. If we, for now, assume that there are defined types `pos` and `neg`, we can express the agent type in the following way:

```
type (_, _) agent =
  | Int : int -> (int, pos) agent
  | IsEven : (bool, neg) agent -> (int, neg) agent
  | T : (bool, pos) agent
  | F : (bool, pos) agent
  | And : (bool, neg) agent * (bool, pos) agent -> (bool, neg) agent
  | NamePos : ('a, pos) agent promise -> ('a, pos) agent
  | NameNeg : ('a, pos) agent resolver -> ('a, neg) agent
```

The astute reader will have surely realised, that all auxiliary ports in the above definition have opposite polarity to what they had in their graphical depiction in Figure 6. This change in polarity is due to the fact that we need to think in terms of the polarity of the agents that will be connected to these ports, therefore we need to invert them. With these types in place, we can refine the definition of `apply_rule`:

```
let rec apply_rule : type a. (a, pos) agent -> (a, neg) agent -> unit =
  fun a1 a2 -> match a1, a2 with
  | T, And (r, b)                    -> b ->><- r
  | F, And (r, b)                    -> ignore b; F ->><- r
  | Int n, IsEven r when n mod 2 = 0 -> T ->><- r
  | Int _, IsEven r                  -> F ->><- r
  | T, If (r, t, _)                  -> t ->><- r
  | F, If (r, _, e)                  -> e ->><- r
  | NamePos v, a                     -> await v ->><- a
  | a, NameNeg v                     -> resolve v a

and ( ->><- ) : type a. (a, pos) agent -> (a, neg) agent -> unit =
  fun a1 a2 -> run_async pool (fun _ -> apply_rule a1 a2)
```

Interactions can only happen between agents that have the same type but opposite polarity. The choice of having the first agent be positive, and the second negative is arbitrary, and we could have introduced them just as well the other way around.

With these type annotations in place, the OCaml compiler will now also inform us, that the catch-all case is superfluous, as we have accounted for all possible active pair patterns. It also means that only one orientation of the agents of an active pair is permitted in the pattern match any more, so we can shorten the code quite a bit.

In it worth noticing, that the OCaml compiler performs not only exhaustiveness checks (see A), but also checks for overlaps in the case patterns. The order of cases matters (e.g., in the above example it is important that the case with the `when` guard is listed first), so later case patterns can generalise prior ones, but if two cases are overlapping completely, the OCaml compiler would complain. This further increases confidence, that the encoding is correct.

It remains to define the two types `pos` and `neg`. In principle, any two types that cannot be unified would work, so we could make `pos` a synonym for `int` and `neg` a synonym for `float`, for example. However, since we never create values of these types (they are only used as *tags* in the type definition), the most idiomatic choice is an *uninhabited* or *empty type*. OCaml allows the definition of such a type as a variant without constructors:

```
type pos = |
type neg = |
```

Since they are defined as two separate types, the type checker can never unify them. We could have, of course, encoded the polarity dynamically as another attribute of all agent constructors again, and put checks at relevant places (e.g., at agent construction and `apply_rule` applications). This would have come at an additional cost in both execution time and code size again. By tracking the polarity in the type system we incur neither of these, since types are removed by the compiler after type checking, and the compiler will reject expressions such as `Int 0 -><- Int 1` already at compile time.

4 Benchmarks

While the main goal of this paper is to express a clean, concise, and correct encoding of interaction nets into OCaml, it is of course interesting to look at the runtime behaviour of the encoded nets as well. To this end, we have encoded three example nets, taken from the inpla project [19]. Specifically, we have encoded nets for calculating Fibonacci numbers, and sorting lists of integers with Quicksort and Mergesort. Our code is available online [6].

All examples were run on a 3.70 GHz Intel Core i9 processor with 10 cores and 2 hardware-threads per core. For each measurement we average over 100 runs. The first question is, of course, if the interaction net encoding can make use of the available cores. Figure 8 shows the relative speed-up ratios when run on pools of different sizes. It is important to notice that the library we are using (Moonpool [2]) works in terms of thread pools, not domain pools. There is always exactly one domain pool created at the start of the program (with the recommended number of domains, i.e., the number of cores/hardware-threads available on the local processor), the library then creates pools of worker threads which share these domains. There are a couple of interesting observations:

Fibonacci scales extremely well with regard to the size of the provided pool, while Quicksort's and Mergesort's performances only marginally increase, before they deteriorate for higher thread counts.

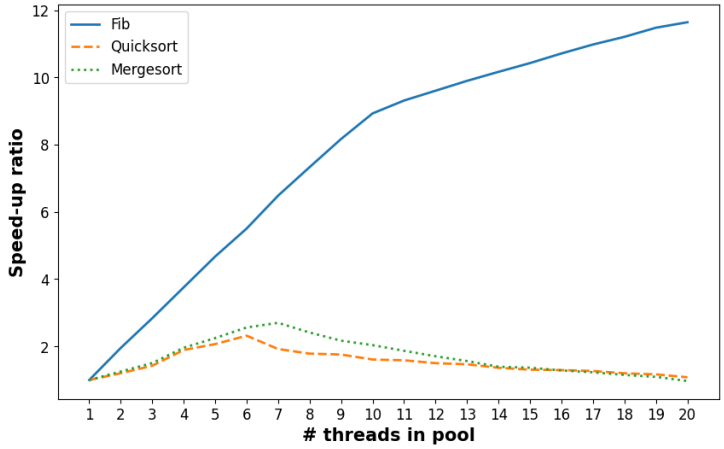


Figure 8: Relative speed-up factors for different pool sizes

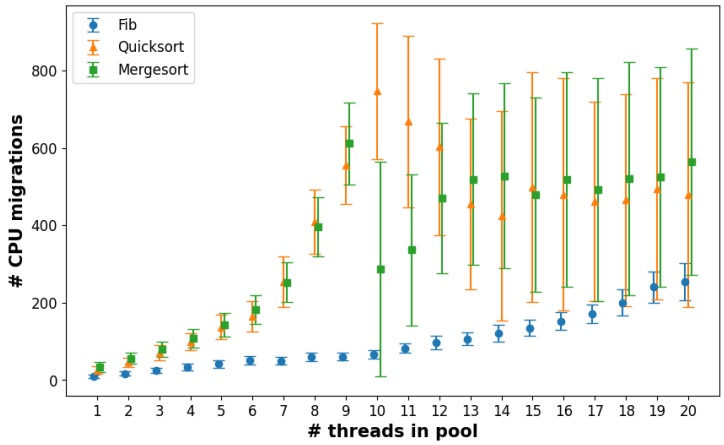
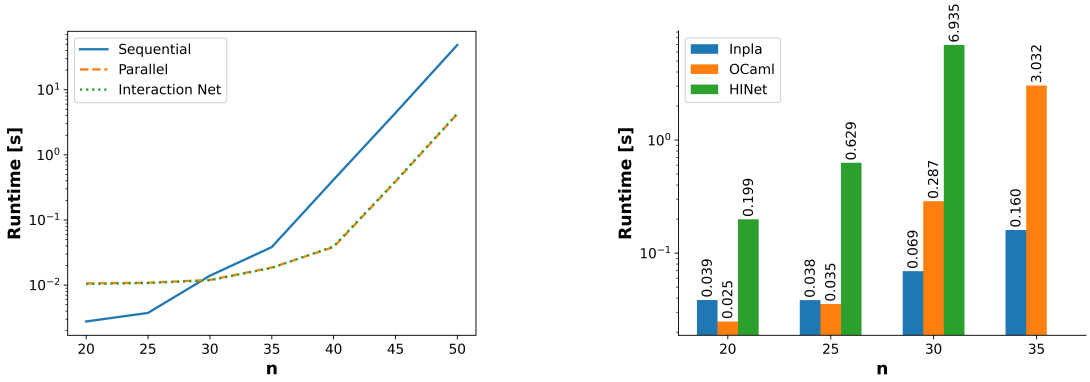


Figure 9: Number of CPU migrations for different pool sizes



(a) Runtime compared to handwritten OCaml

(b) Tool comparison

Figure 10: Fibonacci benchmark

There are several possible reasons for this. As a first indicator, we can look at the average amount of CPU migrations (threads moving from one CPU to another) for different thread counts, as shown in Figure 9. The average number of CPU migrations for Fibonacci increases only slightly for higher thread counts, while showing exponential increases for both sorting algorithms until the number of physical cores is reached. We can also see that the average number of CPU migrations for both sorting algorithms fluctuates significantly for higher thread counts, as indicated by the standard deviations. While this gives a possible explanation for the lack of relative speed-up, further investigation is needed to look at other performance indicators such as cache misses and garbage collection behaviour.

The speed-up ratios are relative to the performance of the sequential execution, they do not allow to reason about absolute performance. We have therefore also compared the performance of the Fibonacci encoding against the sequential and a handwritten parallel OCaml implementation. The results are shown in Figure 10a. Parallelism incurs a price in terms of runtime, so for small inputs (≤ 20) we revert to the sequential algorithm for both the interaction net encoding and the parallel implementation. This shows another advantage of embedding interaction nets in a host language, as such tricks can easily be encoded.

It is interesting to notice that the interaction net encoding is fairly on par with the handwritten parallel code. This suggests, that at least for some algorithms, interaction nets can be a valid candidate for implementing parallel algorithms with fine-grained parallelism in OCaml.

For completeness, Figure 10b compares the runtime of the Fibonacci net for different inputs n in our encoding with two other interaction net evaluators. This comparison is not entirely fair, as they use vastly different strategies for running the provided net. Inpla [19] compiles interaction nets into bytecode for a low-level virtual machine dedicated to the execution of interaction nets. HINet [9] encodes interaction nets in Haskell dynamically, and then interprets these. It is therefore not surprising, that it is slower than the two other approaches. For the last test input, HINet did not produce an output within 10 minutes, and was therefore stopped.

5 Conclusion

In this paper, we presented a method of encoding interaction nets in the programming language OCaml. Our main focus was on an encoding of both the interaction net primitives (agents, nets, rules), as well as Lafont's type system.

Compared to previous work, our encoding offers stronger guarantees of correctness. By defining a variant type with a constructor for each symbol, the type system guarantees the arity constraint and types of agent attributes. Using a GADT we can express both the value type and polarity of each port so that port compatibility is already checked at compile-time. Furthermore, it allows the OCaml type checker to prove the exhaustiveness of the rewriting step (i.e., that all possible active-pair patterns are accounted for). Since the encoding is purely done on the level of types, no additional runtime cost is incurred.

Experiments indicate that the encoding of certain algorithms can make use of the inherent parallelism. Others seem to be constrained by the scheduling of individual threads on the available cores. More investigation is needed regarding different performance counters and optimal scheduling of the rewriting process.

Another direction of future work relates to the choice of library to provide the primitives. As described in the introduction, the idea behind OCaml's low-level primitives in the standard library is that different (opinionated) higher-level libraries can be developed on top of it. The library we have used in this paper is only one possible option, and it would be interesting to see, if the choice of library has a significant impact on the runtime behaviour of the net evaluation.

In this paper, we demonstrated how to encode interaction nets in OCaml manually. To allow for better comparisons with other tools, a compiler could be developed, taking as input a language similar to that of previous interaction net evaluators. OCaml offers an inbuilt system for language extension, therefore by developing an extension for interaction nets, interaction net algorithms could be embedded directly into a surrounding program, while automatically using parallel evaluation if run on a multicore platform.

Finally, we want to remark, that while our focus here was on an encoding in OCaml, our method is general enough to be used for any language that offers GADTs and concurrency.

Acknowledgements This work is partially supported by the ERC CUSTOMER project.

References

[1] Yves Bertot & Pierre Castran (2010): *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st edition. Springer Publishing Company, Incorporated, doi:10.1007/978-3-662-07964-5.

[2] Simon Cruanes (2024): *Moonpool*. Available at `https://github.com/c-cube/moonpool`. Version 0.6.

[3] Georges Gonthier, Martín Abadi & Jean-Jacques Lévy (1992): *The geometry of optimal lambda reduction*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, Association for Computing Machinery, New York, NY, USA, p. 15–26, doi:10.1145/143165.143172.

[4] Abubakar Hassan, Ian Mackie & Shinya Sato (2008): *Interaction nets: programming language design and implementation*. *ECEASST* 10, doi:10.14279/tuj.eceasst.10.156.

[5] Abubakar Hassan, Ian Mackie & Shinya Sato (2009): *Compilation of Interaction Nets*. *Electronic Notes in Theoretical Computer Science* 253(4), pp. 73–90, doi:10.1016/j.entcs.2009.10.018. Proceedings of the Fifth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009).

[6] Nikolaus Huber (2024): *An Encoding of Interaction Nets in OCaml - Software Artifact*, doi:10.5281/zenodo.12633477. Available at `https://doi.org/10.5281/zenodo.12633477`.

[7] Eugen Jiresch (2011): *ingpu*. Available at `https://github.com/euschn/ingpu`.

[8] Eugen Jiresch (2014): *Towards a GPU-based implementation of interaction nets*. *Electronic Proceedings in Theoretical Computer Science* 143, p. 41–53, doi:10.4204/eptcs.143.4.

[9] Wolfram Kahl (2015): *HINet - Interaction Nets in Haskell*. Available at `http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/`.

[10] Wolfram Kahl (2015): *A Simple Parallel Implementation of Interaction Nets in Haskell*. *Electronic Proceedings in Theoretical Computer Science* 179, p. 33–47, doi:10.4204/eptcs.179.3.

[11] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2015): *Frama-C: A software analysis perspective*. *Formal aspects of computing* 27(3), pp. 573–609, doi:10.1007/s00165-014-0326-7.

[12] Yves Lafont (1989): *Interaction nets*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 95–108, doi:10.1145/96709.96718.

[13] John Lamping (1989): *An algorithm for optimal lambda calculus reduction*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 16–30, doi:10.1145/96709.96711.

[14] Ian Mackie (1998): *YALE: yet another lambda evaluator based on interaction nets*. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, Association for Computing Machinery, New York, NY, USA, p. 117–128, doi:10.1145/289423.289434.

[15] Ian Mackie & Shinya Sato (2010): *A lightweight abstract machine for interaction nets*. *ECEASST* 29, doi:10.14279/tuj.eceasst.29.416.

[16] Simon Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, Association for Computing Machinery, New York, NY, USA, p. 295–308, doi:10.1145/237721.237794.

[17] Jorge Sousa Pinto (2000): *Sequential and Concurrent Abstract Machines for Interaction Nets*. In Jerzy Tiuryn, editor: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 267–282, doi:10.1007/3-540-46432-8_18.

[18] Shinya Sato (2015): *Design and implementation of a low-level language for interaction nets*. Ph.D. thesis, University of Sussex. Available at `https://sussex.figshare.com/articles/thesis/Design_and_implementation_of_a_low-level_language_for_interaction_nets/23417312`.

[19] Shinya Sato (2023): *inpla*. Available at `https://github.com/inpla/inpla`. Version 0.11.0.

[20] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman & Anil Madhavapeddy (2020): *Retrofitting parallelism onto OCaml*. *Proc. ACM Program. Lang.* 4(ICFP), doi:10.1145/3408995.

A Overview of OCaml

The basic units of any OCaml program are *expressions*, such as `10 + 25`. Each expression has a *type*, such as int, float, bool, string, etc., and can be bound to a name via a `let` directive. The OCaml compiler uses type inference, so that for most ordinary cases no type annotations are needed. However, a programmer can always express the types of symbols explicitly if desired. The following definitions are equivalent:

```
let x = 10 + 25
let x : int = 10 + 25
```

Anonymous functions can be implemented with the `fun` keyword. As functions are first class values, they can also be bound to a name. Function types are indicated with the `->` symbol. The following definitions are all semantically equivalent:

```
let square = fun x -> x * x
let square : int -> int = fun x -> x * x
let square x = x * x
let square (x : int) : int = x * x
```

Functions are curried, so partial application is possible:

```
let add x y = x + y
let add2 = add 2
let sum = add2 4
```

The core of OCaml's expressive power comes from the ability to define rich data types. The one most used in this paper is called *variant data type* (a generalisation of enumeration and union types):

```
type myType =
    | A of int
    | B of bool
```

The above expression introduces a type `myType`. Values of `myType` are *either* an `A` carrying an integer as an attribute, *or* a `B` carrying a boolean. Variants are constructed by applying their constructor name (e.g., `A` or `B`) to the appropriate number of argument expressions according to the type definitions. Variants can be deconstructed by *pattern matching*:

```
let x : myType = A 1
let output = match x with
    | A n -> if n = 1 then true else false
    | B b -> not b
```

We can see that the value carried by a constructor can be bound to a name on the left-hand side of a case, and then be referred to in the expression on the right-hand side. All cases of a pattern match must return the same type. Matches are also checked for exhaustiveness, so if we change the above example to

```
let x : myType = A 1
let output = match x with
    | A n -> if n = 1 then true else false
```

the compiler will complain:

```
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
B _
```

Variants can be parameterized in terms of other types. For example, the OCaml standard library defines the list type in the following way:

```
type 'a list =
    | Nil
    | Cons of 'a * 'a list
```

The *type parameter* `'a` can be instantiated to any already defined type. Type synonyms can be introduced to make the code more readable:

```
type myList = myType list
let l : myList = Cons (A 1, Cons (B false, Nil))
```

Type constructors are not functions, so their application cannot be partial.