In-place Graph Rewriting with Interaction Nets

Ian Mackie

LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Shinya Sato

University Education Center, Ibaraki University, 2-1-1 Bunkyo, Mito, Ibaraki 310-8512, Japan

An algorithm is in-place, or runs in-situ, when it does not need any additional memory to execute beyond a small constant amount. There are many algorithms that are efficient because of this feature, therefore it is an important aspect of an algorithm. In most programming languages, it is not obvious when an algorithm can run in-place, and moreover it is often not clear that the implementation respects that idea. In this paper we study interaction nets as a formalism where we can see directly, visually, that an algorithm is in-place, and moreover the implementation will respect that it is in-place. Not all algorithms can run in-place however. We can nevertheless still use the same language, but now we can annotate parts of the algorithm that can run in-place. We suggest an annotation for rules, and give an algorithm to find this automatically through analysis of the interaction rules.

1 Introduction

An algorithm runs in-place, or in-situ, if it needs a constant amount of extra space to run. For an algorithm to be in-place, the input is usually overwritten, so mutable data-structures need to be supported by the programming language. There are many well-known in-place algorithms, in particular from the domain of sorting. One example is bubble sort, that we can write in Java for instance:

```
static void bubble() {
  int t;
  for (int i = n-1; i >= 0; --i)
    for (int j = 1; j <= i; ++j)
      if (a[j-1] > a[j])
        { t = a[j-1]; a[j-1] = a[j]; a[j] = t; }
}
```

With some knowledge of what the above instructions do, and tracing a few steps of the execution, we can soon realise that it runs in-place: one additional memory location (`t`) is all that is needed to sort the array `a` of integers. In many programs however, it is not obvious that an algorithm can run in-place, and moreover it is often not clear that the underlying implementation respects that idea. This issue becomes more pertinent when we examine different programming paradigms and different programming styles, especially when we have dynamic data-structures.

In Figure 1 we give four fragments of programs for inserting an element into a sorted list (so part of the insertion sort algorithm). These programs are written in Prolog, Haskell and Java, with the latter written using two different programming styles. Insertion can be written so that it runs in-place, but it is not easy to see which of these examples are (or can be) in-place unless we start to examine the compiler, the run-time system, and the definition of functions like `cons` in the case of Java. Declarative languages (functional and logic based in this example) are designed to be referentially transparent, so the data-structures are updated in a non-destructive way. Moreover, it is not the programmer who decides how memory is allocated and organised in these languages. On the other-hand, languages like C and Java

(the imperative fragment) the programmer does the memory allocation (and de-allocation also in some languages) explicitly, and therefore has a better idea of resource usage. These examples illustrate some of the difficulties in knowing if the program will run in-place or not.

Program 1: Prolog

```
insert([Y|Ys], X, [Y|Zs]):-
  Y < X, !, insert(Ys, X, Zs).
insert(Ys, X, [X|Ys]).
```

Program 2: Haskell

```
insert e [] = [e]
insert e (x:xs) = if e < x then e:x:xs
  else x:(insert e xs)
```

Program 3: Java (functional style)

```
static List insert(int x, List l) {
  if (isEmpty(l) || x < l.head)
    return cons(x, l);
  else
    return cons(l.head, insert(x, l.tail));
}
```

Program 4: Java (destructive style)

```
static List insert(int x, List l) {
  if (isEmpty(l) || x < l.head)
    return cons(x, l);
  else {
    l.tail = insert(x, l.tail);
    return l;
  }
}
```

Figure 1: Example programs

In-place algorithms are important because they can lead to more efficient algorithms, and even change time complexity. If the data-structure supports it, concatenation of two lists can be constant time if done in-place, but linear if not. Memory allocation is also expensive, so minimising it also makes it more efficient. Although less important in some ways, there are devices that have limited space (embedded systems, hand-held devises, etc.), so limiting the space usage if there is no run-time impact is always advantageous.

In this paper we use a formalism where we can see directly, in fact visualise, that an algorithm is in-place, and moreover the implementation respects that it is in-place. We use the graphical rewriting system of interaction nets [5] as our programming paradigm. This visual language has many similarities with term rewriting systems in that they are user-defined systems. For this reason, they can be considered as specification languages. However, they are also a model of computation that requires all aspects of the computation to be explained, including copying and garbage collection. For this reason they are like an implementation model, or low-level language. It is the mixture of these features that allows us to see directly how the program can be implemented, and thus see how the memory is allocated.

Not all algorithms can run in-place however. The formalism will still be of use though, and we identify three different uses of the information we can ascertain from interaction rules:

1. If the rewrite rules have a particular property then the algorithm is in-place (and will be implemented in-place).

2. If the rules can be applied in a given way, so a strategy is needed, then the algorithm can be implemented in-place.

3. If neither of the above hold, then we can still make use of the formalism by re-using as much data as possible in the computation.

For the final point, we can either ask the programmer to annotate the rules, or develop an algorithm to do this automatically. In this paper, we give examples to motivate the first and last points—more details including the second point will be given in a longer version of this paper.

The space usage of algorithms, as well as the time complexity, are fundamental in algorithm design and analysis, and well documented in many textbooks. There have also been a number of works that give a bound on the space usage through type systems, for example [3], and [4]. Our approach is more syntactical, and uses properties of the underlying run-time system.

Overview.   The rest of this paper is structured as follows. In the next section we recall the background, and give some examples to motivate the ideas. We then give some case studies of examples that are in-place in Section 3. In Section 4 we introduce an annotation for the rules which allows for node reuse. We then go on in Section 5 to show an algorithm to annotate a rule automatically in the case of using a fixed-size node representation for nodes. After a brief discussion on how we can use this information in a compiler in Section 6, we conclude in Section 7.

2   Background

In the graphical rewriting system of interaction nets [5], we have a set Σ of *symbols*, which are names of the nodes. 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* labelled with an arrow. Nodes are drawn as follows:
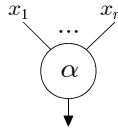
A *net* built on Σ is an undirected graph with nodes as the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*. Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction net analogue of a redex. A rule $((\alpha, \beta) \Rightarrow N)$ 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.
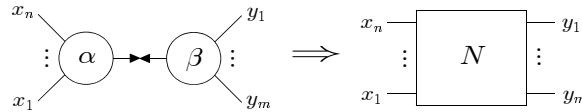
We refer to the rule $((\alpha, \beta) \Rightarrow N)$ as $\alpha \bowtie \beta$. The most powerful property of this graph rewriting system is that it is one-step confluent—all reduction sequences are permutation equivalent. We use an extension of these pure interaction nets: values can be stored in the nodes, and rules can test these values. This is done is such a way as to preserve the one-step confluence property. We use this extension in the insertion sort example below.

It is possible to reason about the graphical representation of nets, but it is convenient to have a textual calculus for compact representation. There are several calculi in the literature, and here we review one calculus [2], which is a refined version of [1].

Agents: Let Σ be a set of symbols, ranged over by $\alpha, \beta, \ldots$, each with a given *arity* ar $: \Sigma \to \mathbb{N}$. An occurrence of a symbol is called an *agent*, and the arity is the number of auxiliary ports.

Names: Let $\mathcal{N}$ be a set of names, ranged over by x, y, z, etc. N and Σ are assumed disjoint. Names correspond to wires in the graph system.

Terms: A term is built on Σ and $\mathcal{N}$ by the grammar: $t ::= x \mid \alpha(t_1,\ldots,t_n) \mid \$t$, where $x \in \mathcal{N}$, $\alpha \in \Sigma$, $\text{ar}(\alpha) = n$ and $t_1,\ldots,t_n,t$ are terms, with the restriction that each name can appear at most twice. If $n = 0$, then we omit the parentheses. If a name occurs twice in a term, we say that it is *bound*, otherwise it is *free*. We write s,t,u to range over terms, and $\vec{s},\vec{t},\vec{u}$ to range over sequences of terms. A term of the form $\alpha(t_1,\ldots,t_n)$ can be seen as a tree with the principal port of α at the root, and the terms $t_1,\ldots,t_n$ are the subtrees connected to the auxiliary ports of α. The term $\$t$ represents an indirection node which is created by reduction, and is not normally part of an initial term. Intuitively, $\$t$ corresponds to a variable bounded with t (or a state such that an environment captures t).

Equations: If t, u are terms, then the unordered pair $t = u$ is an *equation*. Θ will be used to range over sequences of equations.

Rules: Rules are pairs of terms written: $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Theta$, where $(\alpha,\beta) \in \Sigma \times \Sigma$ is the active pair, and Θ is the right-hand side of the rule. We will abbreviate in the following the left- and right-hand sides of the rule by LHS and RHS respectively. All names occur exactly twice in a rule, and there is at most one rule for each pair of agents.

3 In-place algorithms: case studies

Here we give some example interaction net systems that demonstrate the ideas we have discussed previously. The first one is unary numbers with addition. We represent the following term rewriting system: `add(Z,y)=y, add(S(x),y)=add(x,S(y))` as a system of nets with nodes Z, S, Add, and two rewrite rules:
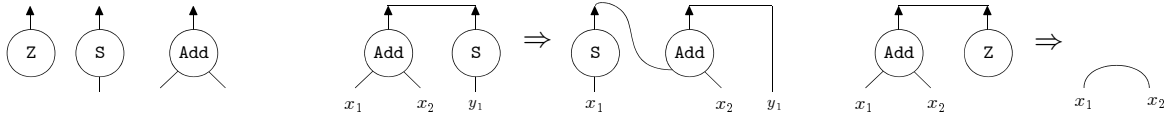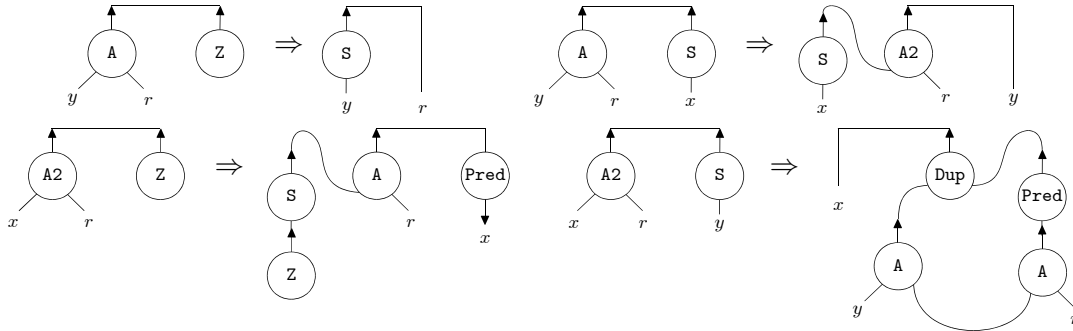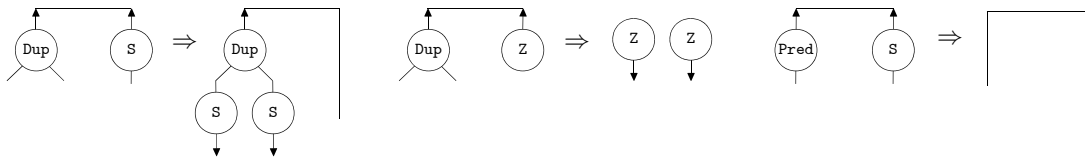

The following is an example of `add(S(Z),Z)`:
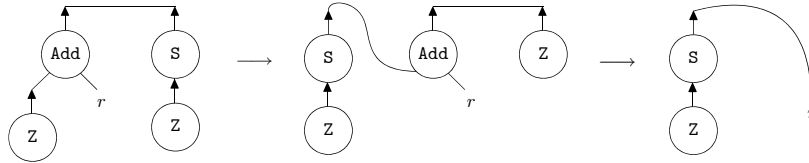
Next, we introduce an example for the Ackermann function defined by:

`ack 0 n = n+1,  ack m 0 = ack (m-1) 1,  ack m n = ack (m-1) (ack m (n-1)).`
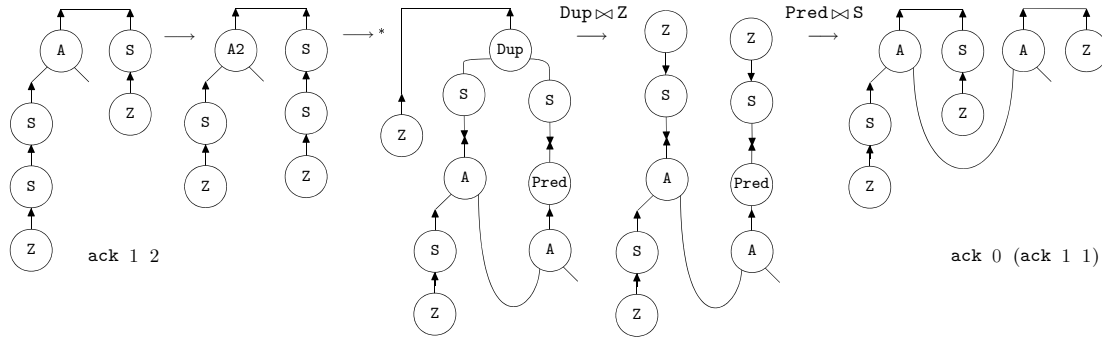
We can build the interaction net system on the unary natural numbers that corresponds to the term rewriting system as follows:

where the node Dup duplicates S and Z nodes, and the node Pred erases the S node:


The following is an example of rewriting:

Observation for in-place running.   Interaction nets are quite unique as a programming paradigm because we are basically writing programs using the internal data-structure. We characterise three kinds of rewriting rule:

- Case 1: there are two nodes in the right-hand side (RHS). The two nodes of the active pair can be reused. Thus, no matter which way we evaluate, the algorithm for these rules can run in constant space. The rules Add ⋈ S, A ⋈ S and Dup ⋈ Z are classified in this case.

- Case 2: there are less than two nodes in the RHS. The active pair nodes can be reused as nodes that occur in the RHS, so in terms of the memory space, it can run in constant space as well. For instance, the rules Add ⋈ Z, A ⋈ Z and Pred ⋈ S are classified in this case.

- Case 3: there are more than two nodes in the RHS. Here, active pair nodes can be reused as nodes that occur in the RHS, but additional memory space is required for other nodes. We divide this into two very different categories:

  1. An active pair creates another active pair that is Case 2 above. These two reductions together make the algorithm in-place. For instance, in the last two-step reductions of ack 1 2 to ack 0 (ack 1 1) in the example, we can save memory space for two nodes when we take Pred ⋈ S and Dup ⋈ Z in this order, in comparison with the order Dup ⋈ Z and Pred ⋈ S.
  2. For instance, the rules A2 ⋈ Z, A2 ⋈ S and Dup ⋈ S are classified in this case. These rules are not in-place, but the total cost can be reduced by choosing the two reused nodes well. For instance, in the rule A2 ⋈ S, it is better to reuse the A2 in the LHS as the right side A (not the left side) because the information of the right auxiliary port (denoted as "*r*") can be reused. Thus we try to reuse the memory in the best way possible.

Insertion sort is a well-known in-place algorithm. The first three interaction rules below encode insertion of an item into a sorted list, and the final two rules encode the insertion sort algorithm.
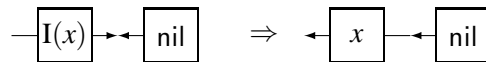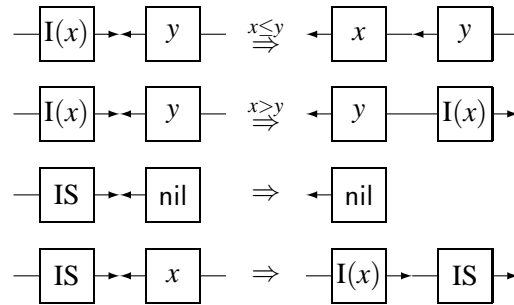
These five rules encode the whole program—there is nothing else. Our real point however, is that in this case a trace of the execution (an animation of this algorithm) is showing no more and no less than what is needed to explain this algorithm. It is in-place because to begin with we need to put an additional IS node, and the final rule for IS erases this. We invite the reader to trace the following example net:
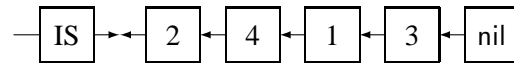
This example is the full system of interaction nets for the insertion sort algorithm, and it runs in-place. There are other examples, for instance reversing a list. In this case, we need to start with adding a rev node, and the final rule deletes it. The following two rules implement reversing any list:
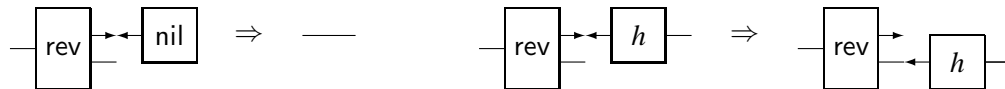

The starting configuration is shown in the following example, which will reverse the list in-place with five interactions:
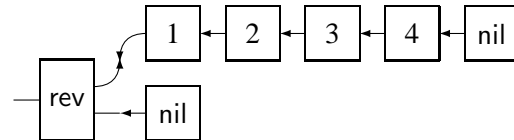

4   Annotating Rules

In the previous section we saw that there are interaction systems that can run in-place. To ensure that this is actually achieved at the implementation level, we need to make sure that for these examples the nodes in the rule are reused when building the new net. This idea can be also used even when there are more than two nodes in the right-hand side of the rule, and in this case there is a choice of how to reuse the nodes. We next introduce an annotation to show which nodes in the RHS of the rule are reused. This helps the compiler to analyse information so that it can improve on the in-place execution of parts of the program.

Annotation: ∗L and ∗R.   We introduce ∗L and ∗R to denote where the left-hand side and the right-hand side nodes in the LHS of a rule are used for in-place computation, respectively. For instance, the rule Add ⋈ S is written as follows:
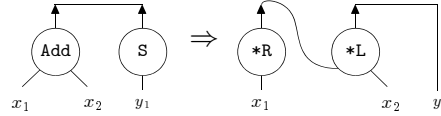
The advantage is that the compiler is easily able to know, traversing the net, not only where the active pair nodes are used, but also which information about the connections should be preserved. For instance, in the above example, the information denoted as "x_1" and "y_1" in the LHS of the rule should be preserved, say as "$\_x_1$" and "$\_y_1$", because these are overwritten in the RHS of the rule, and the "x_1" and "y_1" in the RHS should be replaced by the "$\_x_1$" and "$\_y_1$".

Annotation for changing of the node name. To change the node name, we introduce a name cast, such as the type cast of the C programming language, with the ∗L and ∗R. For instance, the rule A2 ⋈ S is written as follows:
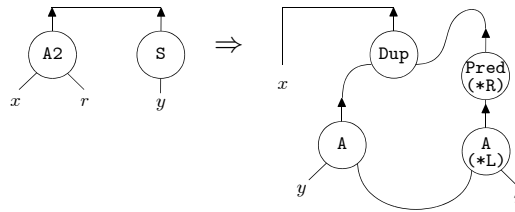
In this example, the compiler can also know that the information denoted as "x" and "y" in the LHS of the rule should be preserved, by checking the connection of ∗L and ∗R.

Advantage of using a fixed-size node representation for nodes. We represent nodes as a fixed-size node, thus fixed-size auxiliary ports. For this we need to use more space than necessary, but we can manage and reuse nodes in a simpler way [6]. Here, we assume that auxiliary ports are assigned by the order from the left-hand side to the right-hand side, the rule A ⋈ Z is written as follows:
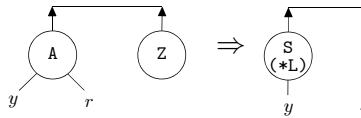
So we can reuse not only the ∗L node, but also the pointer information (denoted as "y").

5 Deriving Annotations

In the previous section we showed how rules can be annotated with information about the reuse of nodes. Here, in the case of using a fixed-size node representation, we define a function to calculate how a given term is similar to others. We first introduce some notation. For strings we write double quotes (" and "). We use the notation $\{x\}$ in a string as the result of replacing the occurrence $\{x\}$ with its actual value. For instance, if $x =$ "abc" and $y = 89$ then "1$\{x\}$2$\{y\}$" = "1abc289". We use $+$ as an infix binary operation to concatenate strings. We also use a symbol $-$ to show the empty sequence. In order to show where in a term is in a sequence of equations, we use the following term path notation: *nth* (L | R) : *arg₁ arg₂*.... For instance, the term t in $x = y, \alpha(\beta(s,t),z) = w$ is denoted as 2L:12 because the t occurs in $\alpha(\beta(s,t),z)$, which is the left-hand side term of the second equation, occurs in $\beta(s,t)$, which is the first argument of the term, and in the second argument of the term $\beta(s,t)$.

Using this, we can now give an important definition for this paper:

Definition 1 (Node matching) *The function* Match *below takes a term and a sequence of equations, and returns a list of a pair* $(score, term\ path)$ *where the score contains the number of the matched agent and matched arguments. We use Standard ML notation for lists, thus, in the following the operator "@" is list concatenation.*

$$\mathsf{Match}(t, (e_1,\ldots,e_n)) \quad = \quad \mathsf{Match}_e(t, e_1, 1)@\cdots@\mathsf{Match}_e(t, e_n, n)$$

$$\mathsf{Match}_e(t, s = u, pos) \quad = \quad \mathsf{Match}_t(t, s, \text{``}\{pos\}\mathtt{L}:\text{''})@\mathsf{Match}_t(t, u, \text{``}\{pos\}\mathtt{R}:\text{''})$$

$$
\begin{aligned}
\mathsf{Match}_t(\alpha(\vec{x}), x, tpath) \quad &= \quad [((0,0), tpath)] \\
|\quad \mathsf{Match}_t(\alpha(\vec{x}), \beta(t_1,\ldots,t_n), tpath) \quad &= \quad [((agentPts, namePts), tpath)] \\
&\qquad @\mathsf{Match}_t(\alpha(\vec{x}), t_1, \text{``}\{tpath\}\mathtt{1}\text{''})@\cdots \\
&\qquad @\mathsf{Match}_t(\alpha(\vec{x}), t_n, \text{``}\{tpath\}\mathtt{n}\text{''}) \\
&\quad \text{where} \quad agentPts = \text{if } \alpha = \beta \text{ then } 1 \text{ else } 0 \\
&\qquad\qquad\quad namePts = \mathsf{Match}_{ns}(\vec{x}, (t_1,\ldots,t_n))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Match}_{ns}(-, \vec{t}) \quad &= \quad 0 \\
|\quad \mathsf{Match}_{ns}(\vec{x}, -) \quad &= \quad 0 \\
|\quad \mathsf{Match}_{ns}((x,\vec{x}), (t,\vec{t})) \quad &= \quad (\text{if } x = t \text{ then } 1 \text{ else } 0) + \mathsf{Match}_{ns}(\vec{x}, \vec{t})
\end{aligned}
$$

Thus Match will return a list of the matching metrics for each node, and give the location in the net for each. We can then easily extract the best one from this information. The following examples illustrate how this information is calculated.

Example 2 *The rule* Add $\bowtie$ S *given previously is written textually as*

$$\mathtt{Add}(x_1,x_2) = \mathtt{S}(y_1) \Rightarrow \mathtt{Add}(\mathtt{S}(x_1,x_2)) = y_1$$

First we take the left hand side term Add(x_1,x_2) *of the active pair.*

$$\mathsf{Match}(\mathtt{Add}(x_1,x_2), \mathtt{Add}(\mathtt{S}(x_1),x_2) = y_1)$$

returns the following list:

```
[((1,1),1L:), ((0,1),1L:1), ((0,0),1L:11), ((0,0),1L:2), ((0,0),1R:)]
```

The first element of this result $((1,1),\mathtt{1L}:)$ *shows that the highest score* $(1,1)$ *is obtained when we annotate the term* $\mathtt{1L}:$, *which is* Add$(\mathtt{S}(x_1),x_2)$, *such as* $(*\mathtt{L})(\mathtt{S}(x_1),x_2)$. *Next we take the right hand side term* S(y_1). *The following is the result of the function* Match *for* S(y_1):

```
[((0,0),1L:), ((1,0),1L:1), ((0,0),1L:11), ((0,0),1L:2), ((0,0),1R:)]
```

This result shows that there is one agent term $\mathtt{1L}:1$ *(thus* S(x_1)*) that has the same id with* S(y_1), *and no agent terms that have the same occurrence of the argument. Therefore,* S(x_1) *should be annotated as* $(*\mathtt{R})(x_1)$. *These annotations correspond to the first graph in Section 4.*

Example 3 *The rule* A2 $\bowtie$ S *given previously is written textually as*

$$\mathtt{A2}(x,r) = \mathtt{S}(y) \Rightarrow x = \mathtt{Dup}(\mathtt{A}(y,w), \mathtt{Pred}(\mathtt{A}(w,r)))$$

The result of Match$(\mathtt{A2}(x,r), x = \mathtt{Dup}(\mathtt{A}(y,w), \mathtt{Pred}(\mathtt{A}(w,r))))$ *is as follows:*

```
[((0,0),1L:), ((0,0),1R:), ((0,0),1R:1), ((0,0),1R:11), ((0,0),1R:12),
 ((0,0),1R:2), ((0,1),1R:21)), ((0,0),1R:211), ((0,0),1R:212)].
```

This shows that the term 1R : 21*, which is* A(w,r)*, should be annotated by* (∗L) *because it has the highest score* $(0,1)$.

In the case of S(y), the result of the Match is as follows:

```
[((0,0),1L:), ((0,0),1R:), ((0,1),1R:1), ((0,0),1R:11), ((0,0),1R:12),
  ((0,0),1R:2), ((0,0),1R:21)), ((0,0),1R:211), ((0,0),1R:212)].
```

Thus, taking account of using a fixed-size node representation, we find that we should annotate the term 1R : 1*, which is* A(y,w)*, by* (∗R)*, though we annotated* Pred *in the graph in Section 4. Of course, the evaluation of the score depends on the implementation method. However, in these cases,* $(0,1)$ *must be highest because the others are* $(0,0)$.

6 Discussion

The algorithm given above can be put to use in the compilation of rules. There is some choice for the compiler if several nodes get the same score. We briefly summarise here some details about the low-level language, and a description of how data-structures are used. Our contribution here is to extend those ideas with reuse. In the longer version of this paper we will give the algorithm for compilation in detail and focus on the data-structures that are used. An important result that we get about compilation is that we do the least amount of work in implementing the interaction rules.

The concrete representation of an interaction net can be summarised by the following diagram, where Γ represents a net, EQ a stack of equations, and I an interface of the net:
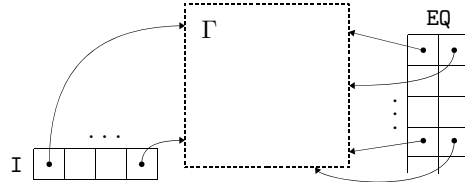
The following is the representation of the net Add(Z, r) = S(w), Add(Z, w) = S(Z), where N are nodes representing variables:
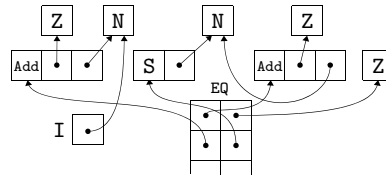
The interface of the net is where we collect all the free edges of a net. We have a set of instructions that can manipulate this data-structure, and the compilation of a rewrite rule needs to generate code to manipulate this structure. What we have achieved in this paper is an annotation of a rule so that we can generate code to manipulate data-structures of the kind shown above with the least amount of work. Specifically, we limit the allocation of memory on the heap, and also avoid unnecessary garbage collection. We also find the best way to build the RHS of the rule, by re-using the memory cells and the existing pointers.

Returning to the ideas presented in the introduction, we have obtained the following:

- Case 1: if there are two nodes in the RHS of the rule, then the algorithm for executing these rules will run in-place, and moreover we will perform the fewest updates of pointers to build the RHS. This is because the algorithm above will find the best way of reusing both nodes and connections.

- Cases 2 and 3: Even when the algorithm is not in-place because we have more than two nodes or fewer than two nodes in the RHS, the ideas of this paper still gives the optimal implementation of the rule. We note however that if we change the data-structure, there may be other solutions to this problem, so the result we obtain is with respect to the chosen data-structure.

One aspect of in-place interaction nets reduction that we have not examined in the current paper is pairing up interaction rules by using a specific reduction strategy. For example, if an interaction system has a rule with one node in the RHS, and another rule with three, then we might be able to schedule these two rules to be performed together, this maintaining the algorithm in-place. There are a number of ways we can do this at the implementation level, and we will include details of this aspect in the long version of the paper.

7   Conclusion

We have introduced a notion to rewrite graphs, specifically interaction nets, in-place. We have identified a number of in-place algorithms and this lead us to an annotation to facilitate the implementation of in-place rewriting by re-using nodes of active pairs. The main feature of this annotation is that is will give the best possible reuse, so will allow the least amount of work to be done when rewriting the graph. We are currently working to incorporate the ideas of this paper into an implementation, using the model previously developed in [2]. We hope to present details of that, and benchmark results at a future time.

References

[1] Maribel Fernández & Ian Mackie (1999): *A Calculus for Interaction Nets*. In G. Nadathur, editor: *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, *Lecture Notes in Computer Science* 1702, Springer-Verlag, pp. 170–187, doi:10.1007/10704567_10.

[2] Abubakar Hassan, Ian Mackie & Shinya Sato (2015): *An Implementation Model for Interaction Nets*. In: *Proceedings 8th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2014, Vienna, Austria, July 13, 2014.*, pp. 66–80, doi:10.4204/EPTCS.183.5.

[3] Martin Hofmann (2000): *A Type System for Bounded Space and Functional In-Place Update–Extended Abstract*. In Gert Smolka, editor: *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Proceedings*, *Lecture Notes in Computer Science* 1782, Springer, pp. 165–179, doi:10.1007/3-540-46425-5_11.

[4] John Hughes & Lars Pareto (1999): *Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming*. In Didier Rémi & Peter Lee, editors: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, 1999*, ACM, pp. 70–81, doi:10.1145/317636.317785.

[5] Yves Lafont (1990): *Interaction Nets*. In: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[6] Simon L. Peyton Jones (1987): *The Implementation of Functional Programming Languages*. Prentice Hall International.