Port Graph Rewriting in Haskell

Jan Rochel

February 6, 2011

Abstract

Introduction to a Haskell library for port graph rewriting with a guide on how to use it in order to define a graph rewriting system and build a graphical, interactive application around it. Also some aspects of the implementation are explained.

1 Introduction

This document gives an overview over a collection of Haskell packages (available on Hackage, see below) that offer a monadic EDSL (embedded domain-specific language) to define a port graph rewriting system together with a frontend that can be used to build a graphical, interactive application around it.

The need for an interactive port graph rewriting library arose from investigations into different λ-calculi and several variants of combinatory logic in order to find more efficient implementations of functional programming languages. These calculi are (implementable as) graph rewriting system, amongst others:

Combinatory logic

Supercombinator calculus

Lambdascope [van Oostrom 2004]

λ-calculus with Wadsworth's notion of sharing

While all of these implement the λ-calculus each have unique characteristics and show different run-time behaviour. It can be very hard to get a good understanding of such systems just by looking at their definition. To this end one would usually evaluate examples by hand or implement an evaluator for the calculus. Both have their problems. The pen-and-paper approach involves an immense amount of manual labour again and again, but at least provides a visualisation of the reductions, which is usually very helpful. An evaluator relieves you from having to perform reductions by hand, but automation does not always make things clearer. Furthermore it forces you to decide for a reduction strategy, which takes away the flexibility of manual reduction.[1]

This is the gap that this library is to fill by making it possible to specify with minimal effort a graph rewriting system in Haskell along with an interactive frontend, that

[1] You might not be sure which reduction strategy you might want apply after all.

allows you to apply individual rewriting rules at specific positions and gives you a visual impression of the evaluation.

1.1 Port Graph Rewriting

The presented framework is for *rewrite systems* that operate on *hypergraphs* with *ports*. It is assumed that the reader has a basic notion of graph rewriting, otherwise refer to [Heckel 2006], for more in-depth study of port graph rewriting, see [Stewart 2001]. However there is one thing to clarify about this document:

DISCLAIMER. This is not a scientific result on graph rewriting. Neither does it give any theoretical insight nor is the library based on theoretical results in graph rewriting. It is not more than a purely pragmatic approach to attack a concrete problem. Also do not complain about improper use or introduction of terminology. This is meant to be informal!

That said, we can concentrate on the ingredients of our framework and describe quickly at what sort of port graph rewriting we are looking: Each of the graph's nodes have a user-defined type. Each type has a signature that specifies a fixed[2] number of distinguishable ports. Nodes are linked via edges, which can only be attached to their ports. While a port can not bind not more than one edge, edges may be hyperedges, i.e one edge may connect not only two ports but any subset of the graph's ports.[3] On this structure we may define a set of *rewrite rules* each expressing a substitution of some part of the graph. They are usually of the form $L \to R$ where L is called the *left-hand side* or *pattern* of the rule and identifies the subgraph that may be substituted by R, the *right-hand side* of the rule. With a set of such rewrite rules we can encode complex graph transformation procedures. A rewrite rule can be applied to any position of the graph that *matches* the pattern L.

2 Case Study: SKI Combinators

As an illustration of such a system, let us look at the SKI combinator calculus, a very simple (yet Turing complete) *term rewrite system*.[4] SKI terms have the form $expr ::= var \mid \mathtt{S} \mid \mathtt{K} \mid \mathtt{I} \mid expr\ expr$ thus an applicative structure over the constants `S`, `K`, and `I` and some set of variables *var*.

$$\begin{array}{lcl} \mathtt{S}\ f\ g\ x & \to & f\ x\ (g\ x) \\ \mathtt{K}\ x\ y & \to & x \\ \mathtt{I}\ x & \to & x \end{array}$$

(Rewrite rules of the SKI calculus)

The first of the above rules, which duplicates x gives us a good motivation to reformulate this term rewrite system as a graph rewrite system. That allows us to avoid

[2] It is actually possible to change the port assignment of a node dynamically while rewriting, but for now it is easier to assume a static assignment.

[3] While for most cases edges connecting nor more than two ports are sufficient, such a restriction would not make the library any simpler, but rather involve additional checks, hence hyperedges are supported.

[4] For some fundamentals about term rewriting, see Wikipedia.

duplication by keeping a single instance of the subterm matched by x but maintaining two pointers to it. That way subsequent reductions inside that subterm are *shared*. *Fig. 1* shows a straightforward translation of the above rules into a graph format with sharing. The @ symbol stands for an *applicator* node, indicating an application of its left subgraph to the one on its right.

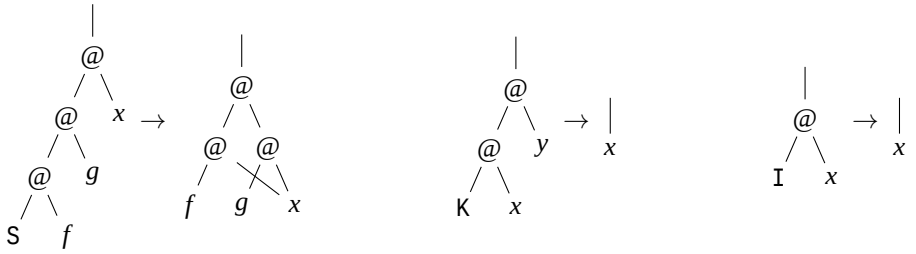

Figure 1: Naive translation of the SKI rules into graph form

However when attempting to implement this system one will notice, that these rules do not yet specify the reduction precisely enough. First, in the K rule, the whole subgraph bound to y is eliminated, a procedure that is by no means atomic. Second, it is not clear how to handle shared subgraphs, e.g. take the term IIx with the I being shared. In order to reduce the I it first has to be *unshared* (see *Fig. 2*).

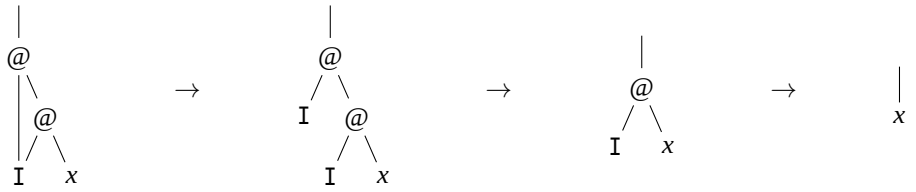

Figure 2: Unsharing of a subgraph

wn order to handle sharing/unsharing and the erasure of subgraphs we make these concepts explicit by introducing two more node types, *duplicators* and *erasers*. A summary of the node types necessary to implement the SKI combinator calculus is shown in *Table 1*.

Based on these node types in *Fig. 3* we redefine the combinator rules from *Fig. 1*. The differences are in the first rule, where the sharing of x is made explicit by a duplicator, and in the second rule where the erasure of the y subterm is made explicit by an eraser in the second rule.

The duplication of subterms (*Fig. 4*) is the same for all three combinators, therefore we represent it as a single rewrite rule where we abstract over the different combinators by C. In case of an application the duplication has to split up and proceed in both of the subterms below the applicator.

The same goes for erasure (*Fig. 5*), where in case of an application after erasing the applicator both of the applied subterms remain to be erased. The last two of the

symbol	name	ports
S	S combinator	input (north)
K	K combinator	input (north)
I	I combinator	input (north)
@	applicator	1 input (north), 2 outputs (south-west, south-east)
◯	eraser	input (south)
▽	duplicator	2 inputs (north-west and north-east), 1 output (north)

Table 1: Node types for the SKI graph rewriting calculus

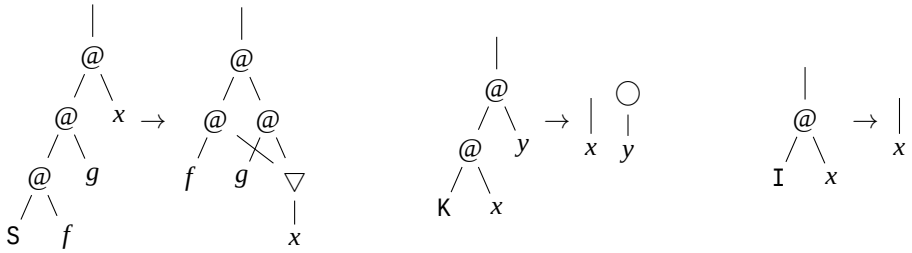

Figure 3: Combinator rules

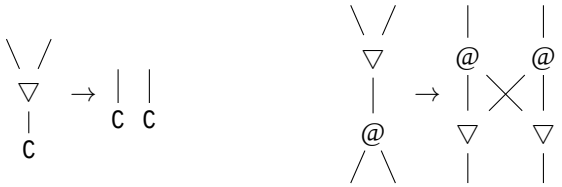

Figure 4: Duplication rules

erasure rules are an optimisation that prevents a subgraph to be duplicated just to be erased afterwards.

Figure 5: Erasure and elimination rules

Exercise. Evaluate the term $SK(KIS)K(SSKIK(SIKSK)S)K$ once by term rewriting and once by graph rewriting using pen and paper and the above rules. Have fun!

3 SKI Combinators in Haskell, a Tutorial

Now on to formulating this system in Haskell! We will dive right in and pick up learning more detailed aspects about the library as we go. Basically, there are two things we need to specify: 1. the data type of our nodes, 2. the rewrite rules that operate on a graph with nodes of that type.

3.1 The Node Type

For the former we choose a straightforward translation of *Table 1* into a Haskell data type using record syntax to give names to the individual ports. In *Fig. 6* each port is of type `GraphRewriting.Port`. We define two additional kind of nodes, `Variable` and `Root`, the first denoting an atom that is not a combinator, the latter indicating the position in the graph that is to be read as the root of the term.

```
data SKI
  = S          {inp :: Port}
  | K          {inp :: Port}
  | I          {inp :: Port}
  | Applicator {inp, out1, out2 :: Port}
  | Duplicator {inp1, inp2, out :: Port}
  | Eraser     {inp :: Port}
  | Variable   {inp :: Port, name :: String}
  | Root       {out :: Port}
```

Figure 6: Node type

While this declaration is very readable to us, the `graph-rewriting` library cannot without further ado, recognise the nodes' ports by their record fields. That is why we need to expose the ports to the library which we do by means of an instance declaration `View [Port] SKI`.[5]

[5]Using Template Haskell, it would be possible to extract all record fields of type `Port` automatically and

```
class View v n where
  inspect :: n → v
  update :: v → n → n
```

Figure 7: `View` abstraction (module `Data.View`)

The `View` abstraction plays an important role in the presented library. It is a multi-parameter type-class that by an instance declaration `View v n` permits to abstract from a type `n` to expose a value of type `v`. It allows both to 'inspect' and 'update' the value, while hiding the internal representation of `n`. By that we can specify our rewrite system to operate on a graph with a polymorphic node type `n` as long as it exposes the node type `v` for which our rewrite rules are defined. The instance declaration (*Fig. 8*) is settled with some boilerplate code.

```
instance View [Port] SKI where
  inspect ski = case ski of
    S          {inp = i}                        → [i]
    K          {inp = i}                        → [i]
    I          {inp = i}                        → [i]
    Applicator {inp = i, out1 = o1, out2 = o2}  → [i, o1, o2]
    Duplicator {inp1 = i1, inp2 = i2, out = o}  → [i1, i2, o]
    Eraser     {inp = i}                        → [i]
    Variable   {inp = i}                        → [i]
    Root       {out = o}                        → [o]
  update ports ski = case ski of
    S          {} → ski {inp = i}                        where [i]      = ports
    K          {} → ski {inp = i}                        where [i]      = ports
    I          {} → ski {inp = i}                        where [i]      = ports
    Applicator {} → ski {inp = i, out1 = o1, out2 = o2}  where [i, o1, o2] = ports
    Duplicator {} → ski {inp1 = i1, inp2 = i2, out = o}  where [i1, i2, o] = ports
    Eraser     {} → ski {inp = i}                        where [i]      = ports
    Variable   {} → ski {inp = i}                        where [i]      = ports
    Root       {} → ski {out = o}                        where [o]      = ports
```

Figure 8: Exposing the nodes' ports to the library

3.2 One Rule

With the node type defined we now ought to specify our rewriting rules. These consist of two components, a left-hand side and a right-hand side. In this library they are represented in a very different way. While the left-hand side expressed as a pattern matching statement within the `Pattern` monad, there are several possibilities to express the right-hand side. But before going into the details let us just have a look at an

derive the corresponding instance declaration.

example to see how such a rule definition looks like. *Fig. 9* shows an implementation of the second of the combinator rules from *Fig. 3* (the `K` rule).

```
ruleK :: (View [Port] n, View SKI n) ⇒ Rule n
ruleK = do
  K {inp = si} ← node
  Applicator {inp = i1, out1 = o1, out2 = x} ← nodeAt si
  require (si ≡ o1)
  Applicator {inp = i2, out1 = o2, out2 = y} ← nodeAt i1
  require (i1 ≡ o2)
  replace0 [Wire x i2, Node $ Eraser {inp = y}]
```

Figure 9: The K rule

We easily recognise a pattern that comprises one `K` combinator node and two applicator nodes that must meet some additional requirements. In the last line these nodes are then replaced by a wire connecting *x* with some other port, and an eraser node that is connected to *y*. If one labels the ports in *Fig. 3* with their port names (with labels named as in *Fig. 6*), the code in *Fig. 9* should be easy enough to understand.

Apparently, the code in *Fig. 9* is monadic. Indeed, the `Rule` type is an alias for a `Pattern` returning a `Rewrite`, both of which are monads. The latter again is an alias for a `State` that operates on a graph as a state variable (see *Fig. 10*). Hence, a `Rewrite` is nothing but a graph transformation expressed in an imperative manner.

```
type Rule n = Pattern n (Rewrite n ())
type Rewrite n = State (Graph n)
```

Figure 10: `Rule` type (module `GraphRewriting.Rule`)

3.2.1 Pattern Matching

All but the last line inside of the `do` block in *Fig. 9* encode the left-hand side of the rule. They are expressions of the `Pattern` monad, which unsuprisingly encodes pattern matching of a subgraph. Its implementation is hidden. The code makes use of three functions, `node`, `nodeAt`, and `require` (see *Fig. 11*). While `node` matches any node occuring anywhere in the graph, `nodeAt` only matches nodes that are directly connected to the edge that is supplied as an argument.

Monadic failure in the `Pattern` monad is interpreted as the pattern not being matched, in which case the rule is not applicable. To express the occurrence of specific node types in our pattern, we exploit a very convenient feauture of how in Haskell patterns matching failures within monads are handled, namely that they result in a monadic failure instead of an `error`. Therefore we can simply denote, which node type to expect in a monadic assignment.

The `require` function is a kind of guard, that returns `()` (aka does nothing) if the supplied parameter is `True` and `fail`s otherwise. In the example it is used to ensure

$node :: View\ v\ n \Rightarrow Pattern\ n\ v$
$nodeAt :: View\ v\ n \Rightarrow Edge \rightarrow Pattern\ n\ v$
$require :: Monad\ m \Rightarrow Bool \rightarrow m\ ()$

Figure 11: `Pattern` construction (module `GraphRewriting.Pattern`)

that the node chain spans along the spine of the matched subexpression. Without these guards the pattern would also match the graph shown in *Fig. 12*.

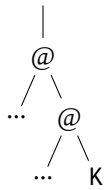

Figure 12: Graph matched by pattern in *Fig. 9* without the `require` guards.

3.2.2 The Right-Hand Side

There are several ways to represent the right-hand side of the rewrite rule. Here we do that in a single line (the last one) using a function of the `replaceN` family that has the effect that all the nodes that have been matched in the left-hand side are replaced by what is given as an argument. In the example, besides adding a new `Eraser` node we also want to connect the edges (or ports)[6] `x` and `i2`, indicated by the `Wire` constructor.

3.2.3 Efficiency

Choosing a good order for pattern construction. This library is not about efficiency

4 Making Patterns into Rules

5 Implementation

Not comprehensive just a glance at various key aspects.

[6] In this library there is no distinction between ports and edges. Edges are really defined as `type Edge = Port`.

```
rewrite :: (Match → Rewrite n ()) → Rule n
erase :: View [Port] n ⇒ Rule n
rewire :: View [Port] n ⇒ [[Edge]] → Rule n
data RHS v = Node v | Wire Edge Edge | Merge [Edge]
replace :: (View [Port] n, View v n) ⇒ Int → ([Edge] → [RHS v]) → Rule n
replace0 vs = replace 0 $ λ[]      → vs
replace1 vs = replace 1 $ λ[e1]    → vs e1
replace2 vs = replace 2 $ λ[e1,e2] → vs e1 e2
```

Figure 13: `Rule` construction

5.1 Graph representation

5.2 Reading and writing

5.2.1 Safe vs. unsafe

5.3 The Pattern Monad

6 Conclusion

applications of graph rewriting: language optimisations, educational purposes

References

[van Oostrom 2004] VINCENT VAN OOSTROM, KEES-JAN VAN DE LOOIJ, MARIJN ZWITSERLOOD, *Lambdascope – Another optimal implementation of the lambda-calculus*, Department of Philosophy, Universiteit Utrecht, April 10th 2004.

[Heckel 2006] REIKO HECKEL, *Graph Transformation in a Nutshell*. *Proceedings of the School on Foundations of Visual Modelling Techniques*, Elsevier, 2006, pages 187-198. *Electronic Notes in TCS*, Volume 148.

[Stewart 2001] CHARLES STEWART, *Reducibility between classes of port graph grammar*, Dept. of Computer Science, Boston University, 24th March 2001.