Rational Term Equality, Functionally

Tom Schrijvers[1] and Bruno C. d. S. Oliveira[2]

[1]Universiteit Gent, Belgium
tom.schrijvers@ugent.be
[2] National University of Singapore
oliveira@comp.nus.edu.sg

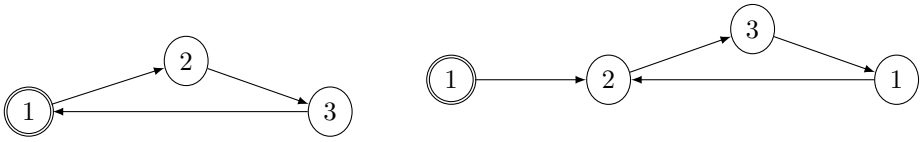
Abstract. This paper presents an elegant purely functional algorithm for deciding the equality of rational terms. The algorithm adapts Hopcroft and Karp's classic algorithm for equality of finite state automata to structured graph representation of rational terms.

1 Introduction

Rational terms are infinite terms that are finitely representable. They consist of a finite number of distinct (possibly infinite) subterms. The finite representation of choice are *graphs*, where infinity is captured in cycles.

Rational terms have many applications in logic and functional programming languages without explicit pointers: graphics [1], parser generation and grammar manipulation [2, 3], finite-state automata [2], natural language processing [4–6], interpreters for control constructs [7], (equi-)recursive types [8], ...

The following two cyclic graphs both denote a rational term, a stream:



The infinite rational terms are recovered by unfolding the graphs from their root, marked with a double circle. When doing so, despite not being structurally equal, the two graphs yield the same infinite stream $[1, 2, 3, 1, 2, 3, ...$

However, it is clearly not pratical to fully compare two infinite terms to decide equality. As far as we can tell, Hopcroft and Karp [9] were the first to propose an efficient algorithm to decide the equality of two rational terms directly from their finite representations. Their algorithm is formulated in terms of a slightly different finite guise of rational terms, finite state automata. More recent imperative incarnations of their algorithm deal with pointer-based representations of graphs and rely on pointer equality.

While graphs are pervasive in computer science, the efficient representation of graphs and elegant implementation of graph algorithms are notoriously hard in pure Functional Programming. A primary reason is that explicit pointer-based

representations and pointer equality cannot be used, as they violate *referential transparency*.

This paper presents an elegant purely functional implementation of Hopcroft and Karp's algorithm for computing the equaltiy of rational terms from their graph representations. It is based on Oliveira and Cook's purely functional representation of structured graphs [10]. The key step in our approach is to infinitely unfold the finite representation of the structured graph, but to tag each unique node with a pointer-like identifier.

2 Generic Structured Graphs

We base ourselves on the new functional representation of rational terms as structured graphs by Oliveira and Cook [10]. This representation employs lightweight *datatype-generic programming* [11] techniques to be generic in the particular structure of the graph; this genericity of representation lends a generality to our approach.

The representation is based on binders: the cycles in the graph are denoted by greatest fixpoints of functions, where the binder denotes one end of the cycle and an occurrence of the bound variable is the other end. The particular binder representation used is parametric higher-order abstract syntax (PHOAS) [12].

$$\begin{aligned}&\textbf{data}\ \mathit{Rec}\ f\ a = \\ &\quad \mathit{Var}\ a \\ &\mid \mathit{Mu}\ ([a] \to [f\ (\mathit{Rec}\ f\ a)]) \\ &\mid \mathit{In}\ (f\ (\mathit{Rec}\ f\ a))\end{aligned}$$

$$\textbf{newtype}\ \mathit{Graph}\ f = \downarrow \{\uparrow :: \forall\ a.\mathit{Rec}\ f\ a\}$$

The representation separates the datatype-specific parts of structured graphs (the *In* constructor) from the generic binding infrastructure (the constructors *Var* and *Mu*).

Generic Structure Support The datatype-specific parts are captured in the f type parameter: f is the *pattern* functor of the recursive datatype. The *In* constructor takes the fixpoint of this functor. Note that, ignoring *Var* and *Mu* constructors, we get the traditional formulation of fixpoints of functors used in various simple datatype-generic programming approaches [11, 13]:

$$\textbf{newtype}\ \mathit{Fix}\ f = \mathit{In}'\ \{\mathit{out}' :: f\ (\mathit{Fix}\ f)\}$$

The following are pattern functors for lists and internally labelled trees

$$\begin{aligned}&\textbf{data}\ \mathit{List}_F\ e\ a = []_F \mid e :_F a \\ &\textbf{data}\ \mathit{Tree}_F\ e\ a = \mathit{Leaf} \mid \mathit{Fork}\ e\ a\ a\end{aligned}$$

and values like:

```
alist :: Rec (List_F Int) a
alist = In (1 :_F In (2 :_F In []_F))
atree :: Rec (Tree_F Int) a
atree = In (Fork 3 (In Leaf) (In Leaf))
```

encode a list $[1, 2]$ and a tree *Fork* 3 *Leaf Leaf* respectively.

Binder Support The constructors *Var* and *Mu* provide the PHOAS binder structure. The type variable a of *Rec* f a is the PHOAS type parameter that denotes the type of variables. In the wrapper type *Graph* this type parameter is universally quantified to enable different possible instantiations from the same graph.

The constructor *Var* is conventional for PHOAS; it denotes an occurrence of a variable. The *Mu* constructor is a generalization from a simple binder to one that binds a group of potentially mutually recursive definitions: $[a]$ is a list of names that can be used in all of $[f\ (Rec\ f\ a)]$.

Note that the cycles must be *productive*. A non-productive loop like $Mu\ (\lambda[x] \to [\,Var\ x\,])$ is not allowed. Productivity is enforced because every cycle starts with an occurrence f of the pattern functor.

The following represents a cyclic list that starts with $[1, 2]$ and then loops back to the beginning:

```
l_1 :: Graph (List_F Int)
l_1 = ↓ (Mu (λ[x] → [1 :_F In (2 :_F Var x)]))
```

3 Structured Graph Equality

The *geq* function defines *structural* equality of structured graphs generically.

```
geq :: ∀ f.Eq_F f ⇒ Graph f → Graph f → Bool
geq g_1 g_2 = runReader (go (↑ g_1) (↑ g_2)) 0 where
  go :: Rec f Int → Rec f Int → Reader Int Bool
  go (Var x) (Var y) = return (x ≡ y)
  go (Mu g)  (Mu h)  =
    do n ← ask
       let a = g (iterate succ n)
           b = h (iterate succ n)
       and ⟨$⟩ local (+length a) (zipWithM (eq_F go) a b)
  go (In x)  (In y)  = eq_F go x y
  go _       _       = return False
```

The auxiliary function *go* deals with the generic binding structure, while the type class Eq_F provides equality for the structure-specific parts of the rational tree:

```
class Functor f ⇒ Eq_F f where
  eq_F :: Applicative m ⇒ (r → r → m Bool) → f r → f r → m Bool
```

Both the type r of recursive occurrences and the type m of applicative effects [14] are treated as abstract types. The recursive call, which knows how to deal with concrete r and m types is explicitly provided. The eq_F function is used by the function *geq*, which calls eq_F with the auxiliary function *go* as the recursive call argument. This technique avoids leaking implementation details of *geq* (such as dealing with fresh variables) into the code users write at the instances of Eq_F.

An example instance of Eq_F is that for $List_F$:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ e \Rightarrow Eq_F\ (List_F\ e)\ \textbf{where} \\
\quad eq_F\ (\sim)\ []_F\ []_F = pure\ True \\
\quad eq_F\ (\sim)\ (x_1 :_F p_1)\ (x_2 :_F p_2) \\
\qquad \mid x_1 \equiv x_2 \quad\ = p_1 \sim p_2 \\
\quad eq_F\ (\sim)\ \_\ \_ \quad\ = pure\ False
\end{array}
$$

The *go* function numbers all variables with the "nesting depth" of their binder; this depth is maintained in the environment of the *Reader* monad.

3.1 The Problem with Structured Graph Equality

While structured graph equality is fairly simple and straightforward, it is overly restrictive. In particular, it is highly sensitive to incidental encoding differences of structured graphs. For instance, the following cyclic list is a one-step unfolding of l_1.

$$
\begin{array}{l}
l_2 :: Graph\ (List_F\ Int) \\
l_2 = \downarrow (In\ (1 :_F In\ (2 :_F Mu\ (\lambda[x] \rightarrow [1 :_F In\ (2 :_F Var\ x)]))))
\end{array}
$$

Yet, it is structurally different from l_1 according to *geq*. To ignore the encoding differences, we need to consider proper *rational term equality*.

4 Rational Term Equality

Definition 1 (Rational Term Equality). *Two rational terms are equal if the infinite unfoldings of their finite representations are equal:*

$$
\begin{array}{l}
(\sim_{RT}) :: Eq_F\ f \Rightarrow Graph\ f \rightarrow Graph\ f \rightarrow Bool \\
g_1 \sim_{RT} g_2 = unfold\ g_1 \equiv_{RT} unfold\ g_2
\end{array}
$$

where unfolding is defined as:

$$
\begin{array}{l}
unfold :: \forall f.Functor\ f \Rightarrow Graph\ f \rightarrow Fix\ f \\
unfold\ g = go\ (\uparrow g)\ \textbf{where} \\
\quad go\ (Var\ x) = x \\
\quad go\ (Mu\ g)\ = head\ \$\ fix\ (map\ (In' \circ fmap\ go) \circ g)
\end{array}
$$

$$
\begin{aligned}
&\quad go\ (In\ fr) \;\; = In'\ \$\ fmap\ go\ fr \\
&fix\ f = \mathbf{let}\ r = f\ r\ \mathbf{in}\ r
\end{aligned}
$$

and equality is purely structural:

$$
\begin{aligned}
&(\equiv_{RT}) :: \forall f.Eq_F\ f \Rightarrow Fix\ f \to Fix\ f \to Bool \\
&x_1 \equiv_{RT} x_2 = runIdentity\ (go\ x_1\ x_2)\ \mathbf{where} \\
&\quad go\ (In'\ y_1)\ (In'\ y_2) = eq_F\ go\ y_1\ y_2
\end{aligned}
$$

The above definition only serves as a specification. It is not practically executable, because $\equiv_{RT}$ does not terminate for infinite structures. What we need is a different approach that properly takes into account the finitely representable nature of rational trees.

5 Pointed Structures

Pointed f augments *Fix* f with node identifiers, called *pointers*. This exposes sharing in the infinite unfolding of the rational term, which is needed to compute equality in finite time.

$$
\begin{aligned}
&\mathbf{type}\ Ptr = Int \\
&\mathbf{data}\ Pointed\ f = P\ Ptr\ (f\ (Pointed\ f))
\end{aligned}
$$

Since the graph has a finite number of nodes, the infinite unfolding also contains only a finite number of distinct pointers.

The two cyclic lists above are both represented as follows in this encoding:

$$
\begin{aligned}
&\mathbf{let}\ cl_0 = P\ 0\ (1 :_F cl_1) \\
&\quad\ \ cl_1 = P\ 1\ (2 :_F cl_0) \\
&\mathbf{in}\ cl_0 :: Pointed\ (List_F\ Int)
\end{aligned}
$$

5.1 From *Graph* to *Pointed*

Every *Graph* can be unfolded into a *Pointed* structure, provided that its pattern functor f is an instance of *Traversable* [14, 15].

$$
\begin{aligned}
&toPointed :: \forall\, f.Traversable\ f \Rightarrow Graph\ f \to Pointed\ f \\
&toPointed\ g = evalState\ (go\$ \uparrow g)\ 0 \\
&\mathbf{where} \\
&\quad go :: Rec\ f\ (Pointed\ f) \to State\ Ptr\ (Pointed\ f) \\
&\quad go\ (Var\ x) = return\ x \\
&\quad go\ (Mu\ g) \;\; = head\ \langle\$\rangle\ mfix\ (traverse\ go_F \circ g) \\
&\quad go\ (In\ r) \;\;\; = go_F\ r \\
&\quad go_F :: f\ (Rec\ f\ (Pointed\ f)) \to State\ Ptr\ (Pointed\ f)
\end{aligned}
$$

```
go_F r      = P ⟨$⟩ fresh ⊛ traverse go r
fresh :: State Ptr Ptr
fresh       = do n ← get
                 put (n + 1)
                 return n
```

This conversion function instantiates the PHOAS variable type to the result type *Pointed f*, in correspondence with the unfolding that replaces *Var* constructors by recursive occurrences of terms.

In order to tag every f constructor with a unique pointer, the state monad threads the unique name supply through the conversion. Moreover, the *Traversable f* constraint enables the recursive application of monadic conversion through f constructors.

The most complex case of the conversion is the *Mu* constructor: The (greatest) fixpoint of the binder function g yields the unfolding. This fixpoint is the monadic *mfix*[1] because the unfolding is interleaved with the monadic tagging of constructors. In the process, the multi-binder is turned into a tree, which is assumed to be defined by the first binder.

6 Hopcroft and Karp, Functionally

The *Pointed* structure removes structural differences from *Graph* that are due to the degree of freedom in encoding cycles. However, it does introduce its own incidental variation: the pointers. Moreover, the co-inductive structure makes it hard to determine how much of two graphs must be compared to be certain of their equality. It is with these two challenges that we deal in the last step, using Hopcroft and Karp's algorithm.

The solution to both problems is to build *equivalence classes* of pointers.

```
hk :: ∀ f.Eq_F f ⇒ Pointed f → Pointed f → Bool
hk p_1 p_2 = runUF_M $ p_1 ∼ p_2 where
  (∼) :: Pointed f → Pointed f → UF_M Bool
  P ptr_1 s_1 ∼ P ptr_2 s_2 = (ptr_1 ≡_Ptr ptr_2) ∨_M (s_1 ≡_Stream s_2)
  (≡_Stream) :: f (Pointed f) → f (Pointed f) → UF_M Bool
  s_1 ≡_Stream s_2 = eq_F (∼) s_1 s_2
```

The *hk* function is defined in terms of the auxiliary function $\sim$ that runs in a monad UF_M encapsulating the management of equivalence classes. Two pointed structures are equivalent if either their pointers are equivalent or their structures are.

The $\equiv_{Ptr}$ operator checks whether ptr_1 and ptr_2 are in the same equivalence class. As a side-effect, if they are not, $\equiv_{Ptr}$ merges their equivalence classes.

[1] Note that g must be sufficiently lazy to tie the knot, e.g., use lazy pattern matching as in $\lambda(\sim[x]) \to 1 :_F (Var\ x)$.

Moreover, if they are not equal, the monadic variant of lazy disjunction $\vee_M$, defined as

$$(\vee_M) :: Monad\ m \Rightarrow m\ Bool \to m\ Bool \to m\ Bool$$
$$x \vee_M y = x \gg\!= \lambda b \to \textbf{if}\ b\ \textbf{then}\ return\ b\ \textbf{else}\ y$$

checks for strucural equivalence with $(\equiv_{Stream})$. This operator performs the structural check eq_F for matching constructors and recursively applies the $\sim$ check on subterms.

The equivalence classes solve the two problems:

1. It overcomes the incidental difference in pointer identity by merging different pointers into the same equivalence class.
2. It ensures termination because if no structural difference is found. The algorithm continues as long as two pointers are discovered that do not belong to the same equivalence class. Given that there are only a finite number of pointers, this means that the algorithm runs out of distinct pointers after a finite number of steps.

6.1 Union-Find

A *union-find* structure manages the pointer equivalence classes. We opt for a simple map-based representation:

$$\textbf{type}\ TPtr = Either\ Ptr\ Ptr$$
$$\textbf{type}\ UF = Map\ TPtr\ TPtr$$

Note that we tag the pointers of the two graphs with *Left* and *Right* respectively in order to tell them apart – this prevents confusion when the same identifier is reused in the two graphs. Then the UF_M monad is just an alias for the *State* monad:

$$\textbf{type}\ UF_M\ a = State\ UF\ a$$
$$runUF_M\ m = evalState\ m\ empty$$

The main pointer equality operator $\equiv_{Ptr}$ is defined in terms of the two core union-find operations, *find* and *union*:

$$(\equiv_{Ptr}) :: Ptr \to Ptr \to UF_M\ Bool$$
$$x \equiv_{Ptr} y = \textbf{do}\ rx \leftarrow find\ (Left\ x)$$
$$\qquad\qquad\qquad\quad ry \leftarrow find\ (Right\ y)$$
$$\qquad\qquad\qquad\quad when\ (rx \not\equiv ry)\ (union\ rx\ ry)$$
$$\qquad\qquad\qquad\quad return\ (rx \equiv ry)$$

where *find* returns the representative or root of an equivalence class:

$$find :: TPtr \to UF_M\ TPtr$$
$$find\ x = \textbf{do}\ uf \leftarrow get$$

```
                return (find' uf x)
find' :: UF → TPtr → TPtr
find' uf x = go x (lookup x uf) where
  go x Nothing = x
  go x (Just y) = go y (lookup y uf)
```

and *union* merges the equivalence classes:

```
union :: TPtr → TPtr → UF_M ()
union x y = modify (λuf → union' uf x y)
union' :: UF → TPtr → TPtr → UF
union' uf x y = let rx = find' uf x
                    ry = find' uf y
                in insert rx ry uf
```

Note that the runtime complexity can be further improved at both the algorithmic level, by adding path compression and performing union-by-rank [16], and the implementation level, by replacing the *Map* in the *State* monad with an *STArray* in the *ST* monad.

7 Related Work

There is much work related to the equality of rational trees. This section only covers a small, but closely related fragment.

Finite State Automata and Bisimulation There is an obvious connection between the equality of finite state automata (FSAs) and rational terms. It would be interesting to investigate whether other FSA operations, like minimizations [17], can be implemented elegantly for the *Graph* representation.

Equality of FSAs is a special case of *bisimulation* [18], the mutual simulation of two state transition systems. *Weak bisimulation* is an extension of bisimulation where the state transition systems can have *silent* (or *internal*) transitions that are ignored. This notion is not relevant for *Graph* where the *Mu* constructor is explicitly productive, but is useful for, e.g., comparing grammars.

Equi-Recursive Types Gauthier and Pottier [19] study the problem of efficient type-checking in an extension of System F with equi-recursive types, System F_μ. In this setting, equi-recursive types are rational terms, extended with binders denoting universal quantifiers. Gauthier and Pottier provide an encoding that eliminates these binders and reduces the problem of deciding equality to traditional rational term equality. As equi-recursive types are often used in the context of object-oriented programming, another important operation is testing *subtyping* [20].

Rational Term Unification Rational (Herbrand) terms arose naturally in Prolog systems that did not implement the occurs check (by default). Currently, they are considered a desirable feature and actively supported by most major Prolog systems (e.g., SWI-Prolog [21]). In those systems not equality but unification is the central operation. Rational tree unification [22] attempts to make two two rational trees by suitably substituting logical variables. It can be implemented as an extension to Hopcroft and Karp's algorithm.

8 Conclusion

Dealing with graphs and graph algorithms in functional programming languages has always been challenging. Structured graphs show that call-by-need languages like Haskell can conveniently represent certain types of graph structures. However it is still unknown how many classic graph algorithms can be conveniently encoded in this representation.

This paper shows that a classic algorithm for computing the equivalence of finite state automata can be nicely adapted to the setting of structured graphs. Interestingly, the purely functional algorithm mimics a traditional imperative pointer-based algorithm using *pointed structures*. We believe that pointed structures can be useful to similarly adapt other imperative pointer-based algorithms.

References

1. Eggert, P.R., Chow, K.P.: Logic Programming, Graphics and Infinite Terms. Technical report, Department of Computer Science, University of California at Santa Barbara (1983)
2. Colmerauer, A.: Prolog and Infinite Trees. In Clark, K.L., Tärnlund, S.A., eds.: Logic Programming. Academic Press, London (1982) pp. 231–251
3. Giannesini, F., Cohen, J.: Parser Generation and Grammar Manipulation Using Prologs Infinite Trees. Journal of Logic Programming **3**, pp. 253–265 (1984)
4. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1984)
5. Carpenter, B.: The Logic of Typed Feature Structures with Applications to Unication-Based Grammars, Logic Programming and Constraint Resolution. In: Cambridge Tracts in Theoretical Computer Science. Volume 32. Cambridge University Press (1992)
6. Erbach, G.: ProFIT: Prolog with Features, Inheritance and Templates. In: 7th Conference of the European Chapter of the Association for Computational Linguistics. pp. 180–187. (1995)
7. Carro, M.: An application of rational trees in a logic programming interpreter for a procedural language. CoRR **cs.DS/0403028**, (2004)
8. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. PLDI '99, New York, NY, USA, pp. 50–63. ACM (1999)
9. Hopcroft, J., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California (1971)

10. Oliveira, B.C.d.S., Cook, W.R.: Functional programming with structured graphs. In: ICFP'12. (2012)
11. Gibbons, J.: Datatype-generic programming. In: Spring School on Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science. Springer-Verlag (2007)
12. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP'08. (2008)
13. Jansson, P., Jeuring, J.: Polyp – a polytypic programming language extension. In: POPL'97. (1997)
14. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming **18**(01), pp. 1–13 (2008)
15. Gibbons, J., Oliveira, B.: The essence of the Iterator pattern. In: Journal of Functional Programming. Volume 19. pp. 377–402. (2009)
16. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), pp. 245–281 (March 1984)
17. McCluskey, E.J.: Introduction to the Theory of Switching Circuits. McGraw-Hill Book Company, Inc., New York
18. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
19. Gauthier, N., Pottier, F.: Numbering matters: first-order canonical forms for second-order recursive types. In: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming. ICFP '04. (2004)
20. Colazzo, D., Ghelli, G.: Subtyping, Recursion and Parametric Polymorphism in Kernel Fun. Information and Computation **198**(2), pp. 71–179
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), pp. 67–96 (2012)
22. Jaffar, J.: Efficient unification over infinite terms. New Generation Computing **2**, pp. 207–219 (1984)