A Flexible Categorial Formalisation of Term Graphs as Directed Hypergraphs

Wolfram Kahl, Yuhang Zhao

A Flexible Categorial Formalisation of Term Graphs as Directed Hypergraphs

Wolfram Kahl and Yuhang Zhao

McMaster University, Canada
{kahl,zhaoy36}@mcmaster.ca

Abstract. Term graphs are the concept at the core of important implementation techniques for functional programming languages, and are also used as internal data structures in many other symbolic computation setting, including in code generation back-ends for example in compilers. To our knowledge, there are no formally verified term graph manipulation systems so far; we present an approach to formalising term graphs, as a relatively complex example of graph structures, in the dependently-typed programming language and proof system Agda in a way that both the mathematical theory and useful executable implementations can be obtained as instances of the same abstract definition.

1 Introduction

Terms (or expressions) are the conceptual data structure at the heart of almost all symbol manipulation for mathematical reasoning and programming language implementation. Terms as a data structure are a kind of trees, and in many applications, intermediate or result terms arise that contain multiple copies of equal subterms. To save space in software implementations of such applications, all these copies are frequently represented by references to a single copy: Conceptually, the tree is replaced by a (directed, and for the purposes of the current paper always acyclic) graph, a *term graph*. Nowadays, term graphs are typically considered as *jungles*, a kind of directed hypergraphs introduced for this purpose by Hoffmann and Plump (1991) and Corradini and Rossi (1993).

For the purpose of creating a toolset for term graph manipulation supported by machine-checked correctness proofs, we develop a flexible formalisation of term graphs in a categorial setting, with the following goals:

– We want to use the formalisation to develop mathematical theories of term graph transformation and how it can be used in particular for correct-by-construction compiler optimisation passes.
– We want to use **that same** formalisation as basis for executable implementations of these compiler optimisation passes.

As our formalisation setting, we use the dependently-typed programming language and proof assistant Agda (Norell, 2007). Agda permits us to write definitions essentially in the way they are written for mathematical purposes, and prove properties about them, but all function definitions are also executable, making this a good environment for correct-by-construction tool development.

The body of this paper will start from a sequence of mathematical definitions (expressed in Agda) of datatypes for somewhat simplified term graphs, then consider also an implementation-oriented definition, and proceed to abstract both to a common generalisation. The full complexity of term graphs is the recovered in a few more refinements.

The result is a simple language for defining not only term graphs, but any of a large class of different kind of graph datastructures, when recognising these as coalgebras possibly including dependently-typed operations, as far as dependencies are used with a certain discipline.

2 Jungle Representation of Term Graphs

We think of term graphs as a kind of data-flow graphs, and we draw the flow from inputs (labelled by their positions in triangles) at the top to output positions at the bottom. We use the jungle approach of Hoffmann and Plump (1991); Corradini and Rossi (1993): We define term graphs as hypergraphs, where each (hyper-)edge is labelled with an operation name, and connected via "input tentacles" (drawn as arrows to the box representing the hyperedge) to the edge's input nodes, and via a single "output tentacle" (pointing away from the edge) to its output node. Term graph inputs correspond to variables in terms; so if we map each input position i to the variable x_i, then the term graph in the following drawing to the left represents the term $(x_1 + x_2) * x_2$:

The term graph drawn above to the right has two output positions, and therefore should be interpreted as a pair of terms; in this case the two output positions are "fed" from the same node, so this is just the pair $\langle (x_1 + x_2) * x_2,\ (x_1 + x_2) * x_2 \rangle$. (We could easily switch to considering multi-output edges, but for the purposes of the current paper this would only result in some duplication, without introducing any additional interesting aspects, so we stick with single-output edges.)

The pure "directed hypergraph" aspect of term graph structure, without considering inputs and outputs, and without restricting the edge output assignment to be bijective onto non-input nodes, can be captured via the following signature:

$$
\begin{aligned}
\mathsf{sigDHG} := \langle\ & \textbf{sorts: } \mathsf{N}, \mathsf{E} \\
& \textbf{ops: } \mathsf{src} : \mathsf{E} \to \mathsf{List}\ \mathsf{N} \\
& \qquad \mathsf{trg} : \mathsf{E} \to \mathsf{N} \\
& \qquad \mathsf{eLab} : \mathsf{E} \to L \qquad \rangle
\end{aligned}
$$

This is a coalgebraic signature in the sense used in (Kahl, 2014, 2015): the argument type of each operation symbol is a single sort, and the result type is a term in a language of functor symbols (here including the constant symbol L and the unary List functor symbol) over the sorts (as variables).

3 Directed Hypergraphs — Simplified

To proceed towards capturing the full term graph structure and reduce ad-hoc notations, we now switch to using Agda Norell (2007) as our mathematical notation. The following Agda record type definition defines the type DHG_{00} to be the type of tuples containing the four sets[1] Input, Output, Inner, and Edge, together with the five functions gOut, eOut, eArity, eLabel, and eIn. The choice to have separate functions for assigning each edge its arity (number of edge input positions), its label, and its actual input node sequence has been made to introduce the right kind of problems for discussion in the current paper.

Different input positions need to be associated with different nodes — for simplicity, we identify input positions and input nodes, and introduce a separate carrier set for "inner" nodes, that is, nodes that are not input nodes. Both input nodes and inner nodes can be used as edge inputs, so we introduce the abbreviation[2] "Node" for the set of all nodes, constructed as the disjoint sum of the input node set and the inner node set:

```
record DHG₀₀ : Set₁ where
  field Input : Set                  -- set of input nodes/positions
        Output : Set                 -- set of output interface positions
        Inner : Set                  -- set of inner (non-input) nodes
  Node = Input ⊎ Inner               -- derived set of all nodes
  field gOut   : Output → Node       -- graph output assignment
        Edge   : Set                 -- set of edges
        eOut   : Edge → Inner        -- edge output node
        eArity : Edge → ℕ            -- edge arity
        eLabel : Edge → Label₀₀      -- edge label
        eIn    : Edge → List Node    -- edge input nodes
```

This models directed hypergraphs with input and output interfaces, but not yet term graphs where eOut needs to be bijective — we will come back to that only in Sect. 11. Dealing with directed hypergraphs is motivated by the fact that we use them as setting for double-pushout (DPO) rewriting of term graphs — directed hypergraphs include the "term graphs with holes" that occur as gluing and host graphs in DPO rewriting steps.

[1] We gloss over the fact that in any real Agda development of these mathematical definitions, Setoid types will normally be used instead of Set. A variant using setoids of the definitions of the current section can be found in (Kahl, 2011, Sect. 3).

[2] Agda **record** declarations simultaneously define modules, and as such can contain other definitions besides field declarations.

The DHG_{00} record type declaration corresponds again to a coalgebraic signature in the sense explained above, after expanding the Node abbreviation, as can be seen in the following reformulation:

```
record DHG₀₁ : Set₁ where
  field Input  : Set                          -- set of input nodes
        Output : Set                          -- set of output interface positions
        Inner  : Set                          -- set of inner (non-input) nodes
        Edge   : Set                          -- set of edges
  field gOut   : Output → Input ⊎ Inner       -- graph output assignment
        eOut   : Edge → Inner                 -- edge output node
        eArity : Edge → ℕ                     -- edge arity
        eLabel : Edge → Label₀₀               -- edge label
        eIn    : Edge → List (Input ⊎ Inner)  -- edge input nodes
```

Since the presence of the local definition for Node may perhaps be confusing for readers unfamiliar with Agda, we will stick with continue our development from this expanded version.

4 Interface-Parameterised Directed Hypergraphs

We will need to implement several operations on our directed hypergraphs, in particular sequential composition: If the set of output positions of G_1 coincides with the set of input positions of G_2, then their sequential composition $G_1 \fatsemi G_2$ results from "gluing them together" along this common interface.

Since we want type-checking to guarantee well-definedness of any applications of $\_\fatsemi\_$ we use in programs manipulating term graphs, the input and output interfaces need to be part of the type of G_1 and G_2. In Agda, this is achieved by making the record type parameterised:

```
record DHG₀₂ (Input : Set) (Output : Set) : Set₁ where
  field Inner  : Set                          -- set of inner (non-input) nodes
        Edge   : Set                          -- set of edges
  field gOut   : Output → Input ⊎ Inner       -- graph output assignment
        eOut   : Edge → Inner                 -- edge output node
        eArity : Edge → ℕ                     -- edge arity
        eLabel : Edge → Label₀₀                  -- edge label
        eIn    : Edge → List (Input ⊎ Inner)  -- edge input nodes
```

(Mathematically, this corresponds to defining a functor from trivial two-sort coalgebras to coalgebras of shape DHG_{01}.)

5 Implemented Directed Hypergraphs

So far, the record types we defined are mathematical datatypes, with sets as components, exactly in the way used for mathematical studies of term graphs. Since we used Agda as our mathematical language, and Agda can be used as a

proof checker, we can build a mathematical theory of directed hypergraphs and term graphs on top of these definitions.

However, since Agda is also a programming language, we would like to also use our definitions for data structures used in programs that manipulate term graphs. However, records containing Set fields are hard to use — how do you save one of those to a file? The field Edge could be a set of functions...

To come from the opposite perspective, consider now what a plausible implementation datatype for directed hypergraphs might look like. We present a "proof-of-concept" implementation based on arrays, using the Vec datatype constructor for dependently-typed vectors from the Agda standard library (Danielsson et al., 2018) — the type "Vec A n" is the type of n-element vectors with elements of type A. (A "production" implementation might for example use some kind of binary trees, or a type of arrays with constant-time access.)

A plausible design is then to use as carrier sets only sets constructed by Fin; for a natural number n, the type "Fin n" is the type of natural numbers less than n. The elements of "Fin n" are precisely the indices that can be used with vectors of type "Vec A n".

However, where the mathamatical data structure contains Sets of size n, the implementation data structure will contain only the index n:

```
record VecDHG₁ (input : ℕ) (output : ℕ) : Set₁ where
  field inner  : ℕ
        edge   : ℕ
  field gOut   : Vec (Fin input ⊎ Fin inner) output
        eOut   : Vec (Fin inner) edge
        eArity : Vec ℕ edge
        eLabel : Vec Label₀₀ edge
        eIn    : Vec (List (Fin input ⊎ Fin inner)) edge
```

It is straight-forward to write a function that maps each element of "VecDHG_1 m n" to the mathematical representation of that graph in the type "DHG_{02} (Fin m) (Fin n)", and this would populate also the Inner and Edge fields with Fin types. However, it is quite combersome to attempt to define even a partial inverse to that, which makes it essentially infeasible to use operations defined on the "mathematical implementation" DHG_{02} to induce operations on the "executable implementation" VecDHG_1.

Perhaps more importantly, there is no good way to "obtain" the definition of VecDHG_1 "from" that of DHG_{02}, or even more generally, to adapt DHG_{02} to finite node and edge sets — one could do this via an extension that adds finiteness proofs. But using this approach to restrict DHG_{02}s to those having node and edge sets of shape "Fin n" would involve a type-level propositional equality that would be extremely awkward to use.

The solution to this problem is to obtain both as instances of a generalised, *abstract* definition, with essentially the goal of being able to

– instantiate with Set and → to obtain the mathematical theory, and
– instantiate with ℕ and flip Vec to obtain the desired implementation.

After putting it this way, the natural option is to use a category as parameter.

6 Abstract Directed Hypergraphs — First Attempt

We now assume that we are in a setting where $\mathcal{C}$ is an arbitrary but fixed category with coproducts — the Agda way of expressing this is to locate the development in a parameterised module (with additional parameters for ListF etc.):

```
module _ (𝒞 : Category) (hasCoproduct : HasCoproducts 𝒞) [...] where
  [...]  -- bring _⊞_ and other useful material into scope...
```

Then occurrences of Set in DHG_{02} are replaced with the type of objects of category $\mathcal{C}$, and operations become morphisms instead of functions:

```
record ADHG₀ (Input : 𝒞.Obj) (Output : 𝒞.Obj) : Set _ where
  field Inner  : 𝒞.Obj                          -- inner (non-input) nodes
        Edge   : 𝒞.Obj                          -- edges
  field gOut   : 𝒞.Mor Output (Input ⊞ Inner)   -- graph output assignment
        eOut   : 𝒞.Mor Edge Inner               -- edge output node
        eArity : 𝒞.Mor Edge objℕ                -- edge arity
        eLabel : 𝒞.Mor Edge Label₁₀             -- edge label
        eIn    : 𝒞.Mor Edge (ListF (Input ⊞ Inner))   -- edge input nodes
```

We shall use the name vecCategory for the category with natural numbers as objects, and where the type of morphisms from m to n is "Vec (Fin n) m"; the coproduct there is just addition.

Trying to instantiate $\mathcal{C}$ with vecCategory presents the problem that that even if objℕ and ListF are supplied as module parameters in [...], we will not find any n such that Fin n represents ℕ respectively List (Input ⊞ Inner). (For the sake of the argument, we will ignore the option to restrict to some maximal arity that might be sufficient for some particular application.)

7 Abstract Directed Hypergraphs — Second Attempt

The solution to this problem is to make use of the type discipline of a coalgebra: Only sorts occur as argument types; infinite types like ℕ and List (Input ⊞ Inner) only occur in the result types. We translate this into a setting where we do not need morphisms starting from all types — we embed the parameter $\mathcal{C}$ (that we plan to instantiate with vecCategory), used for the morphisms between all relevant finite sets, including the carrier sets, in a *semigroupoid*[3] $\mathcal{S}$ that provides objects also for ℕ and List (Input ⊞ Inner).

The semigroupoid $\mathcal{S}$ will need to have morphisms from objects of $\mathcal{C}$ *to* the object objℕ implementing ℕ, and in the context of our implementation, these can all be implemented as vectors of the types "Vec ℕ k" for natural numbers k. However, $\mathcal{S}$ does not need any morphisms starting at objℕ, so we can characterise $\mathcal{S}$ in a way that precisely fits this vector-based implementation: Vectors can contain elements of infinite types, but vectors cannot be infinite.

[3] A semigroupoid is a "category without identity morphisms", analogous to how a semigroup is a "monoid without identity element".

The (full and faithful, coproduct-preserving, ...) semigroupoid functor $\mathcal{F}$ embedding $\mathcal{C}$ in $\mathcal{S}$ becomes another important part of the setting we now adopt:

```
module _ (𝒞 : Category) (hasCoproduct : HasCoproducts 𝒞)
         (𝒮 : Semigroupoid) (ℱ : SGFunctor′ 𝒞 𝒮)
         (objℕ : 𝒮.Obj) (ListF : SGFunctor 𝒮 𝒮) [...] where
```

Functions "between sorts", here gOut and eOut, are now morphisms in the parameter category $\mathcal{C}$, while functions from a sort to an "arbitrary" (potentially infinite) type are morphisms in the parameter semigroupoid $\mathcal{S}$, starting from the $\mathcal{F}$-image of the sort.

```
record ADHG₁ (Input : 𝒞.Obj) (Output : 𝒞.Obj) : Set _ where
  field Inner  : 𝒞.Obj                                        -- inner nodes
        Edge   : 𝒞.Obj                                        -- edges
  field gOut   : 𝒞.Mor Output (Input ⊞ Inner)                 -- graph output
        eOut   : 𝒞.Mor Edge Inner                             -- edge output
        eArity : 𝒮.Mor (ℱ Edge) objℕ                          -- edge arity
        eLabel : 𝒮.Mor (ℱ Edge) Label₁₁                       -- edge label
        eIn    : 𝒮.Mor (ℱ Edge) (ListF (ℱ (Input ⊞ Inner)))   -- edge input
```

Instantiating $\mathcal{C}$ with the category *Set* and $\mathcal{S}$ with the underlying semigroupoid makes the resulting ADHG_1 directly equivalent with DHG_{02}.

Instantiating $\mathcal{C}$ with vecCategory and $\mathcal{S}$ with a carefully constructed semigroupoid ($\mathcal{S}^{\mathcal{F}}$ in appendix B) with arbitrary vectors as morphisms resulting ADHG_1 directly equivalent with VecDHG_1.

Other easy instantiations are useful, too: For example, instantiating $\mathcal{C}$ with the category of all finite sets and $\mathcal{S}$ with the semigroupoid of all sets gives us the variant of DHG_{02} restricted to finite carrier sets.

8 Directed Hypergraphs — Dependently Typed

A different issue with DHG_{02} is the fact that the types do not enforce that the length of an edge's input node list corresponds to its arity: In terms of DHG_{02}, we want to add the following restriction:

```
∀ (e : Edge) → eArity e ≡ length (eIn e)
```

It would be possible to add this in the spirit of datatype invariants as the type of an additional **field** to the record, which then induces a proof obligation at every record construction site. Therefore it is far more attractive to move this invariant into the type system, which is possible in Agda due to its support for dependent types: A **dependent function type** "(e : Edge) → R e" contains functions mapping each e : Edge to an element of type "R e", where R : Edge → Set is assumed to be some "result" type constructor depending on an Edge argument.

We use the additional expressivity provided by dependent types to move from List to Vec in the result type of eIn, and for each result vector we supply the arity of the edge in question as length:

```
record DHG₁ (Input : Set) (Output : Set) : Set₁ where
  field Inner  : Set                                -- set of inner (non-input) nodes
        Edge   : Set                                -- set of edges
  Node = Input ⊎ Inner                              -- set of all nodes
  field gOut   : Output → Input ⊎ Inner             -- graph output assignment
        eOut   : Edge → Inner                       -- edge output node
        eArity : Edge → ℕ                           -- edge arity
        eLabel : (e : Edge) → Label₁ (eArity e)     -- edge label
        eIn    : (e : Edge) → Vec Node (eArity e)   -- edge input nodes
```

At the same time, we also switched the type of edge labels to come from an arity-indexed label set $\mathsf{Label}_1 : \mathbb{N} \to \mathsf{Set}$.

Although this is not anymore of the shape of a coalgebra signature as described in Sect. 2, this is still a type of coalgebras mathematically, due to the fact that the dependent arguments are used only as arguments to other operations.

9 Implementation of Dependently-Typed Fields

The implementation type VecDHG_1 is easily adapted to such dependent fields, exploiting the presence of dependent pair types (Σ-types): The type "Σ a : A • B a" is inhabited by pairs "a , b" where a : A and b : B a (where B : A → Set is a type constructor taking an argument of type A).

Straight-forwardly embedding the type constructors for labels and input vectors in Σ-types yields the following refined implementation type:

```
record VecDHG₂ (input : ℕ) (output : ℕ) : Set₁ where
  field inner  : ℕ
        edge   : ℕ
  field gOut   : Vec (Fin input ⊎ Fin inner) output
        eOut   : Vec (Fin inner) edge
        eArity : Vec ℕ edge
        eLabel : Vec (Σ n : ℕ • Label₀₁ n) edge
        eIn    : Vec (Σ n : ℕ • Vec (Fin input ⊎ Fin inner) n) edge
```

Such structures will then be subject to the following datatype invariants:

```
∀ (e : Edge) → fst (lookup e eLabel) ≡ eArity e
∀ (e : Edge) → fst (lookup e eIn)    ≡ eArity e
```

A more rational implementation (which can easily be obtained by a systematic transformation from VecDHG_2) would store these three equal values only once, and at the same time also be closer to directly representing the functor underlying the coalgebra type here:

```
record VecDHG₃ (input : ℕ) (output : ℕ) : Set₁ where
  field inner : ℕ
        edge  : ℕ
  field gOut  : Vec (Fin input ⊎ Fin inner) output
        eOut  : Vec (Fin inner) edge
        eInfo : Vec (Σ n : ℕ • Label₀₁ n × Vec (Fin input ⊎ Fin inner) n) edge
```

10 Dependently-Typed Abstract Directed Hypergraphs

For abstracting dependently-typed operations into the category-semigroupoid setting of Sect. 7, we introduce an minimal interface to *dependent objects* that can be seen as individual building blocks of a *type-category* as described by Pitts (2001), adapted so that it "does not demand existence of too many morphisms" for our semigroupoid:

Definition 10.1 *For an object I of $\mathcal{S}$, an object D of $\mathcal{S}$ is a* dependent object indexed over I *iff for every object $Y : \mathcal{C}.Obj$ and every morphism f from $\mathcal{F}\ Y$ to D in $\mathcal{S}$ there is a morphism $ind_D\ f$ from $\mathcal{F}\ Y$ to I in $\mathcal{S}$ such that the operation ind_D commutes with $\mathcal{C}$-pre-composition, that is, for every object X of $\mathcal{C}$ and every morphism g from X to Y in $\mathcal{C}$, the following holds:*

$$ind_D\ (\mathcal{F}\ g \fatsemi f) \quad = \quad \mathcal{F}\ g \fatsemi ind_D\ f \qquad \square$$

The Σ-types of Sect. 9 are an instance of dependent objects by virtue of implementing $\mathsf{ind}_D\ \mathsf{f}$ as $(\mathsf{Vec.map}\ \mathsf{proj}_1\ \mathsf{f})$, extracting the index from dependent pairs. The "trick" of dependent objects is that the dependent-pair-projection proj_1 used here does not need to be a morphism of the semigroupoid $\mathcal{S}$, making it possible to define $\mathcal{S}$ in a way that all its morphisms can be implemented based on vectors.

For the abstract variant, we assume a dependent objects Label and a "dependent functor" VecF; the latter needs to map any object A of $\mathcal{S}$ to a dependent object with the common index $\mathsf{obj}\mathbb{N}$. (The dependent functor image of a morphism f can be implemented as f itself tagged with a name of the functor, see appendix B.)

We introduce two new abbreviations, so that operation types now can be of the following three kinds (due to the coalgebra nature, all have to "conceptually start" at sorts, which are objects of $\mathcal{C}$):

$\mathcal{C}.\mathsf{Mor}\ \mathsf{X}\ \mathsf{Y}$	(straight morphisms in $\mathcal{C}$)	
$\mathsf{X} \mapsto \mathsf{A}$	abbreviates	$\mathcal{S}.\mathsf{Mor}\ (\mathcal{F}\ \mathsf{X})\ \mathsf{A}$, where $\mathsf{X} : \mathcal{C}.\mathsf{Obj}$ and $\mathsf{A} : \mathcal{S}.\mathsf{Obj}$
$\mathsf{f} \nearrow \mathsf{D}$	abbreviates	$\Sigma\ \mathsf{g} : \mathsf{X} \mapsto \mathsf{D} \bullet (\mathsf{ind}_D\ \mathsf{g} = \mathsf{f})$, where $\mathsf{f} : \mathsf{X} \mapsto \mathsf{I}$

That is, $\mathsf{f} \nearrow \mathsf{D}$ contains pairs of shape $(\mathsf{g}\ ,\ \mathsf{p})$ where $\mathsf{g} : \mathsf{X} \mapsto \mathsf{D}$ and p is a proof for the morphism equality of $\mathsf{ind}_D\ \mathsf{g}$ with f.

For the instance VecDHG_2, these proofs are exactly proofs for the datatype invariants mentioned there. The final abstract version of our directed hypergraph type therefore also starts closer to VecDHG_2 than to VecDHG_3:

```
module _ [...] (objℕ : 𝒮.Obj) (Label : DepObj objℕ)
                               (VecF : DepFunctor objℕ) [...] where
  [...]
  record ADHG₂ (Input : 𝒞.Obj) (Output : 𝒞.Obj) : Set _ where
    field Inner Edge : 𝒞.Obj
    field gOut   : 𝒞.Mor Output (Input ⊞ Inner)
          eOut   : 𝒞.Mor Edge Inner
          eArity : Edge ↦ objℕ
          ELabel : eArity ↗ Label
          EIn    : eArity ↗ VecF (ℱ (Input ⊞ Inner))
```

While $\mapsto$ is essentially just a kind of "casting" that emphasises the "starting at a sort" intention, the type constructor $\nearrow$ is the real innovation here; thanks to $\nearrow$, the presentation of ADHG_2 does not require local variable binders; $\nearrow$ therefore introduces the possibility of result type dependencies on the result of other operations into coalgebraic signatures while preserving the overall character of traditional signatures. (Technically, $\mapsto$ and $\nearrow$ can be considered as parts of a shallowly-embedded DSL for a novel kind of coalgebra signatures.)

Expanding definitions, we see that $\mathsf{ELabel} : \mathsf{eArity} \nearrow \mathsf{Label}$ from above is a dependent pair of type $\Sigma\ \mathsf{g} : \mathsf{Edge} \mapsto \mathsf{Label} \bullet (\mathsf{ind}_{\mathsf{Label}}\ \mathsf{g} = \mathsf{eArity})$; for convenience, we give individual names to the two constitutents of this pair, which then have the following types, the second of which corresponds to the first datatype invariant in Sect. 9 (where fst implements $\mathsf{ind}_{\mathsf{Label}}$).

```
eLabel     : Edge ↦ Label
eLabel-ind : indLabel eLabel = eArity
```

11 GS-Monoidal Categories of Abstract Term Graphs

The definition of abstract directed hypergraphs we actually use also has the Node definition again, and therefore is even more readable:

```
module _ [...] (objℕ : 𝒮.Obj) (Label : DepObj objℕ)
                               (VecF : DepFunctor objℕ) [...] where
  [...]
  record ADHG₃ (Input : 𝒞.Obj) (Output : 𝒞.Obj) : Set _ where
    field Inner Edge : 𝒞.Obj
    Node = Input ⊞ Inner
    field gOut   : 𝒞.Mor Output Node
          eOut   : 𝒞.Mor Edge Inner
          eArity : Edge ↦ objℕ
          ELabel : eArity ↗ Label
          EIn    : eArity ↗ VecF (ℱ Node)
```

As mentioned in Sect. 2, we are really interested in jungles, which are directed hypergraphs with a one-to-one correspondence between edges and inner nodes established by eOut. Since we need directed hypergraphs as common substrate for an adapted kind of double-pushout term graph rewriting, we define jungles separately as "ADHG_3s where eOut is an isomorphism in $\mathcal{C}$", in Agda:

```
record AJungle (m n : 𝒞.Obj) : Set _ where
  field dhg       : ADHG₃ m n
  open ADHG₃ dhg              -- bringing Inner, Edge, eOut, etc. into scope
  field eOutIsIso : 𝒞.IsIso eOut
  eOut⁻¹ : 𝒞.Mor Inner Edge   -- providing a nice name for the inverse
  eOut⁻¹ = eOutIsIso 𝒞.IsIso.⁻¹
```

The full setting used as context for this includes a few properties not yet mentioned in Sect. 7; it consists of the following items:

– A category $\mathcal{C}$ intended to have (representations of) all possible carrier sets as objects, and (representations of) functions between these as morphisms.
  $\mathcal{C}$ needs to have coproducts, a terminal object, and a strict initial object.
– A semigroupoid $\mathcal{S}$ intended to have (representations of) all possible value sets (including label sets, $\mathbb{N}$, vector sets) as objects.
  $\mathcal{S}$ is only required to contain the morphisms associated with the additional structure below; it can be quite "sparse".
– A full and faithful *semigroupoid functor* $\mathcal{F}$ from the semigroupoid underlying $\mathcal{C}$ to $\mathcal{S}$ that preserves identity morphisms, coproducts, and initial objects.
  This functor is understood as embedding $\mathcal{C}$ into $\mathcal{S}$.
– Specifically as setting for the ADHG definitions, a natural number object objℕ, an objℕ-indexed dependent object Label, and an objℕ-indexed dependent functor VecF for vectors satisfying an appropriate vector specification.

In this setting, we have implemented large parts of the theory of gs-monoidal categories introduced by Corradini and Gadducci (1999): For term graphs, monoidal composition $\otimes$ is "parallel composition" that "concatenates" (via coproduct) the input and output interfaces; gs-monoidal categories are monoidal categories with additional transformations ! and ∇:

– $!_A : A \to \mathbb{1}$ is the *terminator* and introduces **g***arbage*, and
– $\nabla_A : A \to (A \otimes A)$ is the *duplicator* and introduces **s***haring*.

These are present also in cartesian categories such as Lawvere theories, and there they are natural transformations. In **gs**-monoidal categories they do not need to be natural, which is important for term graphs, where **g**arbage and **s**haring make a difference.

We have implemented (Zhao, 2018a,b) Agda-verified gs-monoidal categories with ADHGs respectively Jungles as morphisms fully at the abstract level in the category-semigroupoid setting described above. We also implemented Jungle decomposition and proved it correct, which is the core of the result of Corradini and Gadducci (1999) that that term graphs (i.e., jungles) form a free gs-monoidal category. For this part, we followed Corradini and Gadducci's set-up, which specialises $\mathcal{C}$.obj to $\mathbb{N}$, interpreting n : $\mathbb{N}$ as the type Fin n — this is justified by the fact that there will be a forgetful functor from every practically useful gs-monoidal category mapping the object monoid to $\mathbb{N}$, and this functor will reflect decomposition. We used this specialisation for decomposition of wiring graphs (which have no edges); apart from that, we elaborated the proofs at the abstract category-semigroupoid level as far as we found feasible. An improved library of dependent functors will make fully abstract proofs possible in the future. We also started to develop a rewriting mechanism for these Jungles via constrained DPO rewriting steps in the category of ADHG matchings, see (Kahl and Zhao, 2019).

12 Conclusion

An important observation arising from the development of our ADHG formalisations is that categorial abstraction is frequently enhanced by embedding a "nice"

category in a "big" semigroupoid. Careful choices then allow us to develop theory and implementations at the abstract level, and obtain the conventional *Set*-based mathematical theory as one instantiation, while correct-by-construction executables can be generated via instantiations with concrete datatypes. In this way, we achieve re-usability of theoretical developments as implementations that are tunable for efficiency.

A Representation Contexts

We now provide a more fine-grained abstraction for the category-semigroupoid setting of Sections 7 and 11. Recall that the key idea is to provide a separate interface, the category $\mathcal{C}$, for objects that can be used as carriers of coalgebra sorts, and "extend" this category to an encompassing semigroupoid $\mathcal{S}$ that can contain also other objects that may be used to interpret the result type expressions of coalgebra function symbols. For example, the (conceptually) infinite datatype String will never be used as node set of a graph, but it may well be used for node labels. In addition, a type of "representations" for $\mathcal{S}$-morphisms that "start at $\mathcal{C}$-objects" is assumed — these are the morphisms that may serve as interpretations of coalgebra function symbols. The "upwards arrows" are motivated by visualising the the semigroupoid $\mathcal{S}$ *above* the category $\mathcal{C}$, which is "embedded" into $\mathcal{S}$ via $\mathcal{R}$.

Definition A.1 *A* representation context $\mathcal{X} = (\mathcal{C}, \mathcal{S}, \mathcal{R}, \nearrow, \mathbb{S}, \mathbb{R}, \nearrow)$ *consists of*

- *a category* $\mathcal{C}$
- *a semigroupoid* $\mathcal{S}$
- *a full and faithful semigroupoid functor* $\mathcal{R} : \mathcal{C} \to \mathcal{S}$ *that preserves identities*
- *for each object* k *of* $\mathcal{C}$ *and each object* A *of* $\mathcal{S}$*, a collection* $k \nearrow A$ *of* representations*, together with a bijection* $\mathbb{S}_{k,A}$ *between* $k \nearrow A$ *and the* $\mathcal{S}$*-homset* $\mathcal{R}\ k \to A$*,*
- *for any two objects* k *and* m *of* $\mathcal{C}$*, a bijection* $\mathbb{R}_{k,m}$ *between the* $\mathcal{C}$*-homset* $k \to m$ *and* $k \nearrow \mathcal{R}\ m$*, and*
- *for each representation* $U : k \nearrow A$ *and each* $\mathcal{S}$*-morphism* $g : A \to B$ *a* composition $U \nearrow g$ *in* $k \nearrow B$

such that the following are satisfied:

$$
\begin{array}{lll}
\mathbb{S}_{k,B}\ (U \nearrow g) & = (\mathbb{S}_{k,A}\ U) \fatsemi g & \text{for } U : k \nearrow A, \text{ and } g : A \to B \text{ in } \mathcal{S} \\
\mathbb{S}^{-1}_{k,B}\ (f \fatsemi g) & = (\mathbb{S}^{-1}_{k,A}\ f) \nearrow g & \text{for } f : \mathcal{R}\ k \to A \text{ and } g : A \to B \text{ in } \mathcal{S} \\
\mathcal{R}\ f & = \mathbb{S}_{k,\mathcal{R}\ m}\ (\mathbb{R}_{k,m}\ f) & \text{for } f : k \to m \text{ in } \mathcal{C}
\end{array}
$$

The implementation setting described in Sect. 7 for obtaining VecDHG_1 from ADHG_1 can be explained as a representation context where

- $\mathcal{C}$ has $\mathbb{N}$ as object collection, and $\mathsf{Fin}\ m \to \mathsf{Fin}\ n$ as homset from m to n;
- $\mathcal{S}$ is the semigroupoid underlying *Set*;
- $\mathcal{R} : \mathcal{C} \to \mathcal{S}$ maps $n : \mathbb{N}$ to the set $\mathsf{Fin}\ n$, and is the identity on morphisms;

– for each $k : \mathbb{N}$ and each set A, the type of representations is $k \nrightarrow A = \mathsf{Vec}\ A\ k$, and $\mathbb{S}_{k,A}$ is the canonical isomorphism between $\mathsf{Vec}\ A\ k$ and $\mathsf{Fin}\ k \to A$;
– for each $k, m : \mathbb{N}$, the canonical isomorphism between $\mathsf{Fin}\ k \to \mathsf{Fin}\ m$ and $\mathsf{Vec}\ (\mathsf{Fin}\ m)\ k$ is used as $\mathbb{R}_{k,m}$;
– for each vector $U : \mathsf{Vec}\ A\ k$ and each *Set*-morphism $g : A \to B$, the composition is $U \nearrow g = \mathsf{mapVec}\ g\ U$.

Note that there are "more" vectors than there are morphisms in $\mathcal{C}$, and yet more set functions in $\mathcal{S}$ than there are vectors.

Theoretically, one could choose to identify $k \nrightarrow A$ with the $\mathcal{S}$-homset $\mathcal{R}\ k \to A$, but we consider it useful to keep the two separate: The point of having $\nrightarrow$ as a separate component of representation contexts is that it can be instantiated with *morphism implementations* for which $\mathbb{S}$ provides the semantics in terms of the semigroupoid $\mathcal{S}$, which in turn is intended to provide the connection to *Set*.

For interpretation of coalgebra signatures (as shown in Sect. 2), we assume a fixed interpretation function $\mathcal{F}$ that maps n-ary functor symbols to semigroupoid functors from $\mathcal{S}^n$ to $\mathcal{S}$ that preserve identities (and correspond to meet-preserving relators). If a structure A provides an interpretation of sort symbols as objects in $\mathcal{C}$, then let $[\![t]\!]_A$ be the resulting interpretation of the type expression t, where each sort s is interpreted as $[\![s]\!]_A = \mathcal{R}\ s_A$, and functor symbol applications are interpreted as the corresponding functor applications:

$$[\![F(t_1, \ldots, t_n)]\!]_A = (\mathcal{F}\ F)([\![t_1]\!]_A, \ldots, [\![t_n]\!]_A)$$

For each type expression T, this gives rise to an identity-preserving semigroupoid functor, written $[\![t]\!]$, from the sort-indexed product category $\mathcal{C}^{Sort_\Sigma}$ to $\mathcal{S}$.

Definition A.2 *Let a coalgebraic signature Σ and a representation context $\mathcal{X} = (\mathcal{C}, \mathcal{S}, \mathcal{R}, \nrightarrow, \mathbb{S}, \mathbb{R}, \nearrow)$ be given. A Σ-$\mathcal{X}$-*coalgebra *A consists of*
– *for each sort s an object s_A of $\mathcal{C}$*
– *for each function symbol $f : s \to t$ a representation $f_A : s_A \nrightarrow [\![t]\!]_A$*

*Given two such coalgebras A and B, a Σ-$\mathcal{X}$-*coalgebra homomorphism ϕ from A to B *consists of*
– *for each sort s a representation $\phi_S : s_A \nrightarrow (\mathcal{R}\ s_B)$ of the $\mathcal{C}$ morphism $(\mathbb{R}^{-1}_{s_A, s_B} \phi_S) : s_A \to s_B$,*
– *such that for each function symbol $f : s \to t$, the following* homomorphism property *holds:*

$$f_A \nearrow [\![t]\!]\ (\mathbb{R}^{-1}\ \phi) \quad = \quad \phi_s \nearrow (\mathbb{S}_{s_B, [\![t]\!]_B}\ f_B)$$

This homomorphism property is an equality of representations; in concrete applications this will be a decidable equivalence.

It is easy to see that the Σ-$\mathcal{X}$-coalgebra homomorphisms of Def. A.2 form a category; this is a "good implementation" of Σ-coalgebras in the following sense:

Theorem A.3 *Let a coalgebraic signature Σ, and a representation context $\mathcal{X} = (\mathcal{C}, \mathrm{Set}, \mathcal{R}, \nrightarrow, \mathbb{S}, \mathbb{R}, \nearrow)$ using the full category* Set *for $\mathcal{S}$ be given. For each Σ-$\mathcal{X}$-coalgebra A, applying $\mathcal{R}$ to each carrier object s_A, and applying $\mathbb{S}$ to each*

function symbol interpretation f_A maps the Σ-$\mathcal{X}$-coalgebra A to a conventional Set*-based coalgebra in a way that gives rise to a full and faithful functor.* □

In the setting of Sect. 7, the category of Σ-$\mathcal{X}$-coalgebras is therefore equivalent to the subcategory of Σ-coalgebras over *Set* which results from restriction to finite carrier sets.

B Concretised Representation Context

For an implementation based on, for example, the vectors of Sect. 5, the question arises how to represent not only the components of coalgebras and of morphisms, both of which are representations, but also the results of functor application to morphisms, which are used in the context of the dependent functors mentioned in Sect. 10.

We now assume a language $\mathcal{F}$ of functor symbols (with arity). Our goal is to move from an abstract semigroupoid $\mathcal{S}$, such as *Set*, to one that has a concrete representation amenable to implementation using finite datastructures. (Objects of *Set*, as far as relevant in this context, are considered to be implemented as datatype identifiers or type expressions.)

Given a representation context $\mathcal{X} = (\mathcal{C}, \mathcal{S}, \mathcal{R}, \nrightarrow, \mathbb{S}, \mathbb{R}, \nearrow)$ and a functor symbol semantics that maps each functor symbol $F : \mathcal{F}$ to a semigroupoid endofunctor $[\![F]\!]$ (of corresponding arity) on $\mathcal{S}$, we construct a new *concretised representation context* $\mathcal{X}^{\mathcal{F}} = (\mathcal{C}, \mathcal{S}^{\mathcal{F}}, \mathcal{R}^{\mathcal{F}}, \nrightarrow^{\mathcal{F}}, \mathbb{S}^{\mathcal{F}}, \mathbb{R}^{\mathcal{F}}, \nearrow^{\mathcal{F}})$ over the same base category $\mathcal{C}$, where:

– A $\mathcal{S}^{\mathcal{F}}$ object is
  - either $\mathsf{LIFT}\ k$ for a $\mathcal{C}$ object k,
  - or $\mathsf{EMBED}\ A$ for an $\mathcal{S}$ object A,
  - or $\mathsf{WRAP}\ F\ (Q_1, \ldots, Q_n)$ for an n-ary functor symbol F and $\mathcal{S}^{\mathcal{F}}$ objects $Q_1, \ldots, Q_n$.

  $\mathcal{S}^{\mathcal{F}}$ objects are assigned a straightforward "semantics" as $\mathcal{S}$ objects:

$$[\![\mathsf{LIFT}\ k]\!] = \mathcal{R}\ k$$
$$[\![\mathsf{EMBED}\ A]\!] = A$$
$$[\![\mathsf{WRAP}\ F\ (Q_1, \ldots, Q_n)]\!] = [\![F]\!]([\![Q_1]\!], \ldots, [\![Q_n]\!])$$

– $\mathcal{S}^{\mathcal{F}}$ morphisms are
  - either $\mathbb{S}^{\mathcal{F}}_{k,Q}\ U : \mathsf{LIFT}\ k \to Q$ for a representation $U : k \nrightarrow^{\mathcal{F}} Q$, which is a representation in $k \nrightarrow [\![Q]\!]$,
  - or $\mathsf{MAP}\ F\ (g_1, \ldots, g_n) : \mathsf{WRAP}\ F\ (Q_1, \ldots, Q_n) \to \mathsf{WRAP}\ F\ (P_1, \ldots, P_n)$ for an n-ary functor symbol F and morphisms $g_i : Q_i \to P_i$.

  (There are no morphisms starting from EMBED objects.)

  Morphisms are also given a straightforward semantics in $\mathcal{S}$.

– $\mathbb{R}^{\mathcal{F}}$, the composition $;^{\mathcal{F}}$ in $\mathcal{S}^{\mathcal{F}}$, and the composition $\nearrow^{\mathcal{F}}$ are determined by the semantics; $\mathcal{R}^{\mathcal{F}}$ is induced by LIFT.

$\mathcal{X}^{\mathcal{F}}$ is a well-defined representation context, and if $\mathcal{R}$ preserves finite colimits, so does $\mathcal{R}^{\mathcal{F}}$. Note that $\mathcal{S}^{\mathcal{F}}$ is only a semigroupoid, and cannot be a category, since there are no morphisms starting at EMBED objects, not even identity morphisms. This fact is the motivation for asking only for a semigroupoid in this place in a representation context.

For signatures Σ over the functors in $\mathcal{F}$, the concretised representation context $\mathcal{X}^{\mathcal{F}}$ generates the same Σ-coalgebras as the (possibly abstract) context $\mathcal{X}$, but extends the concretely implementable morphisms to "exactly all morphisms ever required while reasoning about Σ-coalgebra transformation".

Theorem B.1 *The category of Σ-$\mathcal{X}^{\mathcal{F}}$-coalgebras is equivalent to the subcategory of the corresponding category of Σ-coalgebras in $\mathcal{S}$ restricted to carriers in $\mathcal{C}$.* □

References

A. Corradini, F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–331, 1999. ISSN 1572-9095. https://doi.org/10.1023/A:1008647417502.

A. Corradini, F. Rossi. Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming. *Theoret. Comput. Sci.*, 109(1–2): 7–48, 1993. https://doi.org/10.1016/0304-3975(93)90063-Y.

N. A. Danielsson, M. Daggit et al. Agda standard library, version 0.17. `http://tinyurl.com/AgdaStdlib`, 2018.

B. Hoffmann, D. Plump. Implementing term rewriting by jungle evaluation. *Informatique théorique et applications/Theoretical Informatics and Applications*, 25(5):445–472, 1991. https://doi.org/10.1051/ita/1991250504451.

W. Kahl. Dependently-typed formalisation of typed term graphs. In R. Echahed (ed.), *Proc. of 6th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2011*, EPTCS, vol. 48, pp. 38–53, 2011. https://doi.org/10.4204/EPTCS.48.6.

W. Kahl. Categories of coalgebras with monadic homomorphisms. In M. Bonsangue (ed.) CMCS 2014, LNCS, vol. 8446, pp. 151–167. Springer, 2014. https://doi.org/10.1007/978-3-662-44124-4_9. Agda theories at `http://RelMiCS.McMaster.ca/RATH-Agda/`.

W. Kahl. Graph transformation with symbolic attributes via monadic coalgebra homomorphisms. *ECEASST*, 71:5.1–5.17, 2015. https://doi.org/10.14279/tuj.eceasst.71.999.

W. Kahl, Y. Zhao. Dependently-typed formalisation of typed term graphs. In M. Fernández, I. Mackie (eds.), *Proc. Tenth International Workshop on Computing with Terms and Graphs, TERMGRAPH 2018*, EPTCS, vol. 288, pp. 26–37, 2019. https://doi.org/10.4204/EPTCS.288.3.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, 2007. See also http://wiki.portal.chalmers.se/agda/pmwiki.php.

A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. 5, pp. 39–128. Oxford University Press, 2001.

Y. Zhao. *A Machine-checked Categorial Formalisation of Term Graph Rewriting with Semantics Preservation.* PhD thesis, McMaster University, 2018a.

Y. Zhao. A Formalisation of Term Graph Rewriting in Agda — TGR1. Mechanically checked Agda development, with 283 pages literate document output. http://relmics.mcmaster.ca/RATH-Agda/TGR1/, 2018b.