A MACHINE-CHECKED CATEG.

FORMALISATION OF TERM GRAPH REW.
A MACHINE-CHECKED CATEGORIAL FORMALISATION OF
TERM GRAPH REWRITING WITH SEMANTICS PRESERVATION

By

YUHANG ZHAO, M.Sc.

A Thesis

Submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

McMaster University © Copyright by Yuhang Zhao, September 2018
Title: A Machine-checked Categorial Formalisation of Term Graph Rewriting with Semantics Preservation

Author: Yuhang Zhao

M.Sc.

McMaster University, Hamilton, Ontario, Canada

Supervisor: Dr. Wolfram Kahl

Number of Pages: x, 104
To my father
Abstract

Term graph rewriting is important as “conceptual implementation” of the execution of functional programs, and of data-flow optimisations in compilers. Since term graphs were introduced into the literature, various flavours of term graph rewriting have been investigated mainly as implementation of term rewriting. One way to define term graph rewriting rule application is via the well-established and intuitively accessible double-pushout (DPO) approach. It uses the abstraction of category theory to define matching and replacement on a black-box level through basic categorial theoretic concepts like pushouts. However, the semantics preservation of DPO term graph rewriting, to our knowledge, has never been formalised before. In this thesis, we show the gs-monoidal categories proposed by Andrea Corradini and Fabio Gadducci serves not only as a category-theoretic interface for programming “on top of” term graphs with sequential and parallel composition, but also as the necessary link relating our formalisation of DPO term graph rewriting to the categorial description for program semantics.

One achievement of our work is the representation of term graphs employed by the dependently-typed programming language Agda on a suitable level of abstraction from the concrete choice of set representation for graph nodes and edges through the novel category-theoretic abstraction of dependent objects.

Another result is the formalisation of the functor from gs-monoidal category of term graphs to any gs-monoidal categories which enables us to obtain the semantics of term graphs.

Finally, we present a new result proving the semantics preservation for such DPO-based term graph rewriting.
Acknowledgements

First and foremost, I would like to express my gratitude to my academic supervisor
Dr. Wolfram Kahl, for his insight guidance and continuous encouragement throughout
my research studies.

I wish to thank the members of my Supervisor Committee: Dr. Christopher Anand
and Dr. Emil Sekerinski, for many valuable advices and comments. Thanks to many
friends and colleagues for discussions and friendship, help me through these years.

Many thanks to the Natural Sciences and Engineering Research Council of Canada
(NSERC) and McMaster University graduate scholarship which have provided the
financial support during my doctoral studies.

Last but not least, I would like to thank my father Suian Zhao and my mother Xiaobi
He, for their understanding, my son Eric and my daughter Sophia, for the happiness
they brought to me, and my wife Jinfang Xu, for her support and encouragement.
# Contents

1 Introduction .................................................. 1
   1.1 Term Graph Rewriting .................................. 1
   1.2 Related Work and Problem Statement .................. 4
   1.3 Agda .................................................... 5
   1.4 Main Contributions ...................................... 5
   1.5 Outline of the Thesis .................................. 6

2 Background Knowledge ....................................... 8
   2.1 Term Graphs ............................................. 8
   2.2 Categories ............................................. 10
      2.2.1 Category ........................................... 11
      2.2.2 Monoidal Category ................................ 12
      2.2.3 GS-Monoidal Category ............................ 15
   2.3 Algebraic Approach to Graph Transformation .......... 17

3 Formal Presentation of Categories ......................... 19
   3.1 Category ................................................. 19
   3.2 Monoidal Category ..................................... 24
   3.3 GS-Monoidal Category .................................. 26

4 Overview of the Formalisation ................................ 28

5 Abstract Directed HyperGraphs and Categories .......... 30
   5.1 Directed HyperGraph Implemented with Sets .......... 31
   5.2 Category Embedded in Semigroupoid .................... 33
   5.3 Categoric Abstraction of Dependent Objects .......... 35
   5.4 Abstract Directed HyperGraph (ADHG) ................ 37
   5.5 Homomorphism and Category ............................ 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.1</td>
<td>Homomorphism</td>
<td>42</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Homomorphism Category</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Matching and Category</td>
<td>43</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Matching</td>
<td>43</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Matching Category</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>Term Graph Category</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>GS-Monoidal Category</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>GS-Monoidal Expressions</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>GSME</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>GS-Monoidal Category of GSMEs</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>GSME0</td>
<td>53</td>
</tr>
<tr>
<td>7</td>
<td>Decomposition of Acyclic Term Graph</td>
<td>54</td>
</tr>
<tr>
<td>7.1</td>
<td>Purpose</td>
<td>55</td>
</tr>
<tr>
<td>7.2</td>
<td>Function Type and Working Mechanism</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation</td>
<td>56</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Discrete Term Graphs</td>
<td>58</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Non-discrete Acyclic Term Graphs</td>
<td>61</td>
</tr>
<tr>
<td>7.4</td>
<td>Object Structuring Transformation</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>Correctness of Acyclic Term Graph Decomposition</td>
<td>68</td>
</tr>
<tr>
<td>8.1</td>
<td>Formalisation of the Correctness</td>
<td>68</td>
</tr>
<tr>
<td>8.2</td>
<td>Non-discrete Acyclic Term Graphs</td>
<td>69</td>
</tr>
<tr>
<td>8.3</td>
<td>Discrete Term Graphs</td>
<td>72</td>
</tr>
<tr>
<td>8.4</td>
<td>Core Lemmas</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>Term Graph Rewriting</td>
<td>76</td>
</tr>
<tr>
<td>9.1</td>
<td>Term Graph Rewriting</td>
<td>77</td>
</tr>
<tr>
<td>9.1.1</td>
<td>The Variant of DPO Approach</td>
<td>77</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Gluing Condition</td>
<td>79</td>
</tr>
<tr>
<td>9.1.3</td>
<td>The Pushout Complement Construction</td>
<td>83</td>
</tr>
<tr>
<td>9.1.4</td>
<td>The Right Pushout Construction</td>
<td>84</td>
</tr>
<tr>
<td>9.2</td>
<td>Semantics Preservation of Term Graph Rewriting</td>
<td>87</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Semantics of Term Graphs</td>
<td>88</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Context</td>
<td>88</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Tree-structure for term $(2 + x) * (2 + x)$ .............................. 1
1.2 Term graph for $(2 + x) * (2 + x)$ ............................................ 2
1.3 Example for term graph rewriting derivation .......................... 3
2.1 Examples for term graph and jungle ........................................ 9
2.2 Natural transformation .......................................................... 11
2.3 Pushout of $(b, c)$ ................................................................. 12
2.4 Monoidal category — coherence axiom1 ................................ 13
2.5 Monoidal category — coherence axiom2 ................................ 13
2.6 Symmetric monoidal category — associativity coherence axiom ... 14
2.7 Symmetric monoidal category — unit coherence ....................... 14
2.8 Symmetric monoidal category — inverse law ............................ 14
2.9 gs-monoidal category — coherence axiom-1 ............................ 15
2.10 gs-monoidal category — coherence axiom-2 ............................ 15
2.11 gs-monoidal category — coherence axiom-3 ............................ 15
2.12 gs-monoidal category — monoidality axiom-1 ......................... 16
2.13 gs-monoidal category — monoidality axiom-2 ......................... 16
2.14 $G$ and $G_{\downarrow}$ .............................................................. 17
2.15 Example for direct derivation of unlabelled graphs .................. 18
2.16 The double pushout characterisation of a direct derivation ........ 18
3.1 Illustration for coproduct ..................................................... 22
3.2 Illustration for CoCone2Univ ................................................ 22
3.3 Choice of coproducts ............................................................ 23
5.1 The vertical composition of 2-cells ....................................... 45
5.2 The horizontal composition of 1-cells ..................................... 46
5.3 exchADHG ................................................................. 47
5.4 termADHG ......................................................... 47
5.5 dupADHG ......................................................... 48
5.6 Illustration for the violation of naturality ................. 48
6.1 The isomorphic jungles ......................................... 51
6.2 An ADHG not expressible as GSME ...................... 51
7.1 A non-discrete term graph $G$ ................................ 56
7.2 A decomposition for $G$ ......................................... 57
7.3 Illustration for $\psi$, $\phi$ and $\psi$ .......................... 64
9.1 A term graph rewriting derivation in DPO approach .... 78
9.2 A double pushout characterising a direct derivation .... 79
9.3 Dangling conflict in term graph rewriting ............... 80
9.4 Identification conflict in term graph rewriting .......... 81
9.5 Shape of Fig. 9.3 and Fig. 9.4 .............................. 82
9.6 The pushout complement construction ..................... 83
9.7 The right pushout construction ............................. 85
9.8 A conflict for right Pushout construction ............... 85
9.9 Non-monic $X$ result in $B$ not being a jungle ........... 86
9.10 The context view of the application graph .......... ... 89
Chapter 1

Introduction

We start this introductory chapter with an overview of term graph rewriting and the algebraic approach to term graph rewriting in Sect. 1.1 which serves as introduction to the problem area of our work. It is followed by a review of related work and the problem statement in Sect. 1.2. Then we give a brief introduction to the programming language Agda in Sect. 1.3 which we use as both the implementation tool for term graph rewriting and the proof assistant for proving the correctness of our implementations. In section Sect. 1.4, we sketch our main contributions. Finally, in Sect. 1.5, we present an outline of the remainder of this thesis.

1.1 Term Graph Rewriting

Terms are an inductively defined tree-structured abstraction for representing the abstract syntax of non-trivial formal languages, in particular logical languages and programming languages. Fig. 1.1 is the tree-structure for the term \((2 + x) \ast (2 + x)\):

![Tree-structure for term (2 + x) * (2 + x)](image-url)

Figure 1.1: Tree-structure for term \((2 + x) \ast (2 + x)\)
Evaluation of expressions can be understood as a procedure of repeatedly applying the term rewrite rules until the normal form of an expression is obtained, if it exists. For example, in functional programming, computation is treated as evaluation of expressions, which is also a form of term rewriting. Because of the efficiency of rewriting in space and time, term graph rewriting is nowadays an efficient implementation technique for term rewriting.

**Term graphs** are “terms with sharing”; they can be understood as a generalisation of the graph view of “terms as (directed) trees” that allows nodes to have multiple “parents”. Fig. 1.2 is one term graph representation for the term \((2 + x) \ast (2 + x)\):

![Term graph for \((2 + x) \ast (2 + x)\)](image)

Figure 1.2: Term graph for \((2 + x) \ast (2 + x)\)

Term graphs can be defined in various ways. They can be defined as directed graphs with constraints (Barendregt et al., 1987), as labelled hyper-graphs called jungles (Hoffmann and Plump, 1991; Corradini and Rossi, 1993), or as sets of recursive equations (Ariola and Klop, 1994).

**Term graph rewriting** is the field of studying the representation of term graphs and their transformations. A simple example can be found in Fig. 1.3 illustrating a term graph rewrite step: The result in term graph \(B\) of the rewriting step is obtained by replacing the occurrence of \(L\) in \(A\) by \(R\). Term graphs were introduced into the literature by Wadsworth (1971) as an efficient implementation mechanism for the \(\lambda\)-calculus. Since then, term graph rewriting has been used as an efficient implementation of term rewriting. Sharing of common subexpressions in term graphs improves the efficiency of rewriting in space and time, because it does not repeat rewriting for each copy of the shared common subexpressions. Term graph rewriting has been applied in program transformation, e.g., the “Higher Object Programming System” HOPS by Kahl (1999).

For a general introduction to applications and transformation of term graphs, see the collection by Sleep, Plasmeijer, and Van Eekelen (Sleep et al., 1993). More detailed presentation of the notions will be supplied in Sect. 2.1.
Figure 1.3: Example for term graph rewriting derivation
1.2 Related Work and Problem Statement

The aim of our work is to supply a verified tool for program transformation as term graph rewriting with semantics preservation. The implementation of the tool requires: The representation of terms as graphs, the sufficiently intuitive description of transformation concept, and the characterisation of semantics preservation.

When attempting to employ traditional categorial approaches to graph rewriting, two main problems arise: First, categories of “standard” term graph homomorphisms typically do not have all pushouts, since unification translates into pushouts; Second, the interface graphs needed for the double-pushout (DPO) approach are typically not term graphs, but some kind of “term graphs with holes”. Term graph rewriting is therefore a niche of graph transformation that has pioneered exploration of formalisms where pushout squares are generalised in some way, in particular by using different morphisms in the horizontal and vertical directions of the standard DPO drawing. For example, Duval et al. (2009) proposed an approach using separate classes of horizontal and vertical morphism for term graph rewriting, who are using a specific rule concept as morphisms in the horizontal direction in their “heterogeneous pushout approach”. More recently, motivated by attributed graphs, which share some characteristics with term graphs, Habel and Plump (2012) propose “\(M,N\)-adhesive transformation systems” as one general framework to accommodate different classes of morphisms in the horizontal and vertical directions of the double-pushout setting.

To our knowledge, the correctness of DPO term graph rewriting still remains elusive. I.e., the preservation of function semantics during the transformation is demanded as the correctness of the program transformation. The main challenge is lack of a means to connect the semantics of term graphs with the DPO transformation mechanism.

Corradini and Gadducci (1999a, 2002) opened up a new way of investigating term graphs by defining gs-monoidal categories which parallels the categorical description of term rewriting (Lawvere, 1963); they showed that taking natural numbers as objects and term graphs with \(m\) inputs and \(n\) outputs as morphisms from object \(m\) to object \(n\) produces a free gs-monoidal category, and thus automatically obtained a functorial semantics for term graphs in arbitrary gs-monoidal categories, which include all Cartesian categories, and so in particular also \(\text{Set}\). Continuing this line of work, Corradini and Gadducci (1997, 1999b) obtain semantics preservation for a low-level definition of “ranked dag rewriting”, and involving “contexts” analogous to the contexts of term rewriting. Finally, Corradini and Gadducci (2005) show a quasi-adhesive category of term graphs, but emphasis that adhesive categorial rewriting in that category does not quite match term graph rewriting. They mention in their conclusion that a possible alternative is to perform the DPO on a super-category of hypergraphs; this is essentially the approach we are elaborating here.
1.3 Agda

Agda ([Norell, 2007; Bove et al., 2009]) is a proof assistant which is based on Martin-Löf’s intuitionistic type theory. By the Curry-Howard correspondence, logical propositions are presented by types, and a proposition is proved by writing a term of the type. The Emacs-based interface allows programming by gradual refinement of incomplete type-correct terms; programmers can get useful information on how to fill the unfinished parts. Moreover, Agda has full support for Unicode identifiers and keywords, which allows us to write mathematics in a natural way.

Agda is also a functional programming language with dependent types. Dependent types are types that depend on elements of other types. They are different to parameterised types which are usually families of types indexed by other types. As soon as the signature of term constructors is a parameter, term graphs are naturally considered as dependently typed, since the length of the argument list of a term constructor application depends on that constructor. Agda also supports a wide range of inductive data types.

Our project is to implement a tool for term graph rewriting with the proof for correctness provided. Agda permits us to not only write definitions essentially in the way they are written for mathematical purposes, and prove properties about them, but also to compile these definitions into executable programs. It is a good environment for correct-by-construction tool development. Moreover, we have available RATH-Agda ([Kahl, 2011]) which is an Agda library for fundamental formalisations of categories and relational categories ([Kahl, 2001]). Throughout this thesis, we shall use Agda as our main mathematical notation. Fundamental language mechanisms and vocabulary will be explained when they are encountered in the remaining sections.

We use Agda version 2.5.2 with Andrea Vezzosi’s patch ([Vezzosi, 2015]) and the standard library version 0.13. Without Andrea Vezzosi’s patch, typechecking of theorems in (nested) instances of large theories, e.g., evalGSMEv-EQ in Sect. 8.4, does not terminate even within a week. (A side-effect of using this patch is that occasionally we can make fewer implicit arguments explicit than necessary for standard Agda.)

1.4 Main Contributions

In this thesis, we will present the following achievements:

The first result is the representation of term graphs as a restricted version of abstract directed hypergraphs “(ADHG)” on a suitable level of abstraction from the concrete choice of set representation for graph nodes and edges. The dependently-typed internal aspects of term graphs leads us to the novel category-theoretic abstraction of
dependent objects. On the top of them, we formalise various categories for term graphs. In the term graphs in the spirit of “jungles” (Hoffmann and Plump 1991; Corradini and Rossi 1993), the function for edge output needs to be a bijection. Our initial representation does not have this restriction, because it is allowed to have some “empty area” in the presentation of the internal structure. In the formalisation of term graph rewriting (see Sect. 9.1.1), such “empty area” is used to accommodate the difference between the internal structures of two interface preserved jungles.

Second, we implement an algorithm for term graph decomposition with the results encoded as expressions, and prove the correctness of this algorithm. This essentially achieves a machine checked formalisation of the freeness result of Corradini and Gadducci (1999a). The algorithm for term graph decomposition is the essential part of the functor from term graphs to their semantics which is a main component of the study of the term graph rewriting with gs-monoidal categorial representation.

Third, we formalise the term graph rewriting through the variant of the standard DPO approach where the morphisms in the horizontal and vertical direction are different. We also present a new result formalising the semantics preservation of the DPO approach to term graph rewriting which is listed as an open problem in Corradini and Gadducci (1999a).

For the sake of conciseness, this thesis only supplies the essential code fragments. However, the complete Agda modules are available on-line (Zhao 2018).

1.5 Outline of the Thesis

The thesis is organised as below:

In Chapter 2 we start with the recall of the basic definitions of term graphs. It is followed by a brief introduction to category and gs-monoidal category, with the algebraic approach to graph transformation.

In Chapter 3 we supply the formalisation of category and gs-monoidal category in Agda.

Using the background and technical terminology established in Chapter 2 and Chapter 3 we explain the contribution of this thesis as first sketched in Sect. 1.4 more precisely in Chapter 4.

In the remaining thesis, we present the incremental implementations for the requirements specified for our goal in Sect. 1.2 in sequence:

In Chapter 5, we start from the directed hypergraphs implemented with Sets in Sect. 5.1 but which is not a good choice for formalising finite term graphs. We
then switch to explore the abstraction over the collection of edges and inner nodes of term graphs through an underlying category in Sect. 5.2. The need to coordinate the dependently-typed internal aspects of term graphs with the abstraction of term graph components leads us to use the novel category-theoretic abstraction of dependent objects in Sect. 5.3. It is followed by the appropriate formalisation of the dependent-aspects of term graphs as ADHG in Sect. 5.4. Next we proceed to supply the formalisation of the category of term graphs and homomorphisms in Sect. 5.5 and also the gs-monoidal category of ADHG in Sect. 5.8.

In Chapter 6, we provide gs-monoidal category expressions according to the specification of gs-monoidal category. These are used for encoding the result of the decomposition of acyclic term graphs. Then we proceed to introduce the algorithm for acyclic term graph decomposition in Chapter 7 followed by the correctness of this algorithm in Chapter 8. They are the essential ingredients for obtaining the semantics of term graphs.

Finally, in Chapter 9, we explore how to use these tools to formalise term graph rewriting in Sect. 9.1 and semantics preservation in Sect. 9.2.

In Chapter 10, we summarise the achievements of our work, and also the limitations which outline future research.
Chapter 2

Background Knowledge

In this chapter, we recall some basic notions that will be useful in the remainder of this thesis. In section Sect. 2.1 we recall the concepts of term graphs and term graph rewriting. In section Sect. 2.2 we recall the fundamental notions of category, monoidal category and gs-monoidal category. In section Sect. 2.3 we recall the algebraic approach to graph transformations. More details and examples about the notions can be found in the referenced literature.

2.1 Term Graphs

Let signature $\Sigma$ be a countable set of function symbols where each $f \in \Sigma$ comes with a natural number $\text{arity}(f)$. Function symbols with arity 0 are called constant symbols. Let further $X$ be a countable set of variables such that $X \cap \Sigma = \emptyset$.

A term over $\Sigma$ and $X$ is a variable, a constant, or a function application $f(t_1,t_2,\ldots,t_n)$ where $f$ is a function symbol of arity $n \geq 1$ and $t_1,\ldots,t_n$ are terms. Rewriting systems (also called reduction or replacement systems) are means to compute by a stepwise transformation of objects. In term rewriting systems these objects are terms.

Term graphs may be used to represent terms. A term may be read as a syntax tree, which is already a form of graph. However instead of retaining the variable names which may occur in a term, we instead use variable nodes, in such a way that repeated occurrences of the same variable in a term are represented by multiple edges pointing to the same variable node of the graph.

In early definitions, e.g., [Sleep et al., 1993], term graph over $\Sigma$ consists of a set of nodes $N$, a tuple of members of $N$ (the “roots” of the graph), a partial function $\text{lab}$ from a subset of $N$ to $\Sigma$, and a partial function $\text{succ}$ from the same subset of $N$ to
$N^*$, where for each $n \in \text{dom}(\text{lab})$, we have $\text{arity}(\text{lab}(n)) = \text{length}(\text{succ}(n))$. Nodes outside the domain of $\text{lab}$ and $\text{succ}$ are called variable nodes. The first drawing in Fig. 2.1 shows an example. A homomorphism from a graph $G = (N_G, \text{lab}_G, \text{succ}_G)$ to a graph $H = (N_H, \text{lab}_H, \text{succ}_H)$ is a function $f$ from $N_G$ to $N_H$, such that for every non-variable node $n$ of $G$, $\text{lab}_G(n) = \text{lab}_H(f(n))$ and $\text{f(succ}_G(n)) = \text{succ}_H(f(n))$. These graphs and homomorphisms form the category of term graphs over $\Sigma$.

Figure 2.1: Examples for term graph and jungle

HyperGraphs \cite{Drewesetal2007} are an alternative formulation of essentially the same type of term graphs. Each node in a hypergraph represents a well-formed expression, or term. A hyperedge (hyperarc) has a list of source nodes and a list of target nodes. (Although edges with multiple outputs have uses for example in the code graphs of \cite{Kahletal2006, AnandandKahl2009}, most of the literature including all the cited work by Corradini and Gadducci, only considers single-output operations (edges), so we also do this here.) Hyperedges are labelled with function symbols. A directed hypergraph $G$ over $\Sigma$ is a tuple $G = (V, E, s, t, m)$ where $V$ is a set of nodes (or vertices), $E$ is a set of hyperedges (or hyperarcs), $s : E \to V^*$ is the source function, and $t : E \to V$ is the target function, and $m : E \to \Sigma$ maps a hyperedge to an operator. See the two examples to the right in Fig. 2.1. Nowadays, term graphs are typically considered as jungles, a kind of directed hypergraphs introduced by \cite{HoffmannandPlump1991}. Directed hypergraphs differ from jungles in that $t$ does not have to be a bijection: The possibility that $t$ is not surjective allows for “undefined” inner nodes,
and the possibility that $t$ is not injective allows for “join nodes” in the sense of Kahl et al. (2006).

In Fig. 2.1, we show to the left a drawing of a term graph, with variable nodes drawn as “$x$" and inner nodes drawn as their labels “+”, “∗”, “2”, “3”, etc. Argument sequence is implicitly understood to be indicated by the left-to-right ordering of the outgoing argument edges.

In the middle is a jungle drawing in the same orientation, with nodes drawn as small bullets, and hyperedges drawn as boxes with their labels inside. Position in the argument sequence is indicated by labels “0” and “1” attached to the argument tentacles of the hyperedges. The arrows here follow the customary direction of the term graph drawing to the left.

To the right is the kind of drawing we will use in the remainder of this thesis: Nodes and hyperedges are drawn in the same way as in the middle, but the arrows follow the data flow; graph input nodes are on the top of the drawing and flagged by numbered triangles with arrows going into the input nodes; Graph output nodes are on the bottom side of the drawing and flagged by numbered triangles with arrows coming out of the output nodes.

Term graph rewriting is concerned with representation of functional expressions as graphs, and with rule-based transformation of these graphs. Representing expressions as graphs allows to share common subexpressions, improving the efficiency of rewriting in space and time. Since term graphs were introduced, term graph rewriting has been investigated mainly as implementation of term rewriting. Besides the operational style, there is also the categorical description of term graph rewriting (Corradini and Gadducci, 1999a). Best suited for proofs, the categorial presentation for the term graph rewriting provides the definition of rewriting over term graphs and lays the foundation for the development of proof and analysis.

2.2 Categories

This section recalls the notions of category in Sect. 2.2.1, more details could be found in the book by Barr and Wells (1990). It is followed by the notions of monoidal category in Sect. 2.2.2 where more details could be found in the book by Mac Lane (1971); Kelly (1982). Note what we present in this chapter are the non-strict versions of monoidal category and symmetric monoidal category, as well as the gs-monoidal category in Sect. 2.2.3 because they request the equivalence up to “isomorphism” in the formalisation (see Chapter 3), whereas the strict versions would request adaptations for propositional equivalence.
2.2.1 Category

In this subsection we recall the definition of category.

A category \( C = (\text{Obj}, \text{Arw}, \text{source}, \text{target}, \circ, \text{id}) \) comprises:

- A collection \( \text{Obj} \) of entities called \textit{objects};
- A collection \( \text{Arw} \) of entities called \textit{arrows} (also called \textit{morphisms});
- Two assignments for each arrow \( f \), one is object \textit{source} \( f \) which is the source object of \( f \); the other is object \textit{target} \( f \) which is the target object of \( f \); they are also denoted together as \( A \xrightarrow{f} B \) or \( f : A \to B \); the collection of all arrows with source \( A \) and target \( B \) is written \( C(A, B) \);
- A composition operator \( \circ \) for each pair of arrows \( f \) and \( g \) with \textit{target} \( f = \text{source} \) \( g \), where the composite arrow \( f \circ g : \text{source} \ f \to \text{target} \ g \) satisfying the \textit{associative law}: For any arrows \( f : A \to B, g : B \to C, \) and \( h : C \to D \) (where \( A, B, C, \) and \( D \) not necessarily distinct), the following holds: \( f \circ (g \circ h) = (f \circ g) \circ h \);
- For each object \( A \), an identity arrow \( \text{id}_A : A \to A \) satisfying the following \textit{identity law}: For any arrow \( f : A \to B \), the following holds: \( \text{id}_A \circ f = f \) and \( f \circ \text{id}_B = f \).

For categories \( C \) and \( D \), a \textit{functor} \( F : C \to D \) is a pair of functions \( F_0 \) and \( F_1 \) mapping objects and arrows of \( C \) to \( D \) where:

- If \( f : A \to B \) is an arrow from object \( A \) to \( B \) in category \( C \), then \( F_1(f) : F_0(A) \to F_0(B) \) is in \( D \).
- For any object \( A \) of \( C \), \( F_1(\text{id}_A) = \text{id}_{F_0(A)} \).
- If \( f \circ g \) is defined in \( C \), then \( F_1(f) \circ F_1(g) \) is defined in \( D \), and \( F_1(f \circ g) = F_1(f) \circ F_1(g) \).

Following common usage, we omit the subscripts “\( 0 \)” and “\( 1 \)”. Given two functors \( F \) and \( G \) from category \( C \) to \( D \), a \textit{natural transformation} \( T : F \to G \) is a family of arrows \( T(A) : F(A) \to G(A) \) of \( D \) indexed by the object \( A \) of \( C \), such that for each arrow \( f : A \to B \) of \( C \) the diagram of \( D \) in Fig. 2.2 commutes.

\[
\begin{array}{ccc}
F(A) & \xrightarrow{T(A)} & G(A) \\
\downarrow F(f) & \quad & \downarrow G(f) \\
F(B) & \xrightarrow{T(B)} & G(B)
\end{array}
\]

Figure 2.2: Natural transformation
A natural isomorphism \cite{Simmons2011} is a natural transformation \( T : F \to G \) of two functors \( F \) and \( G \), such that for each object \( A \) in \( C \), \( T(A) : F(A) \to G(A) \) is an isomorphism in \( D \).

Given a category and two arrows \( b : A \to B, c : A \to C \), a triple \( (D, d : B \to D, f : C \to D) \) as in Fig. 2.3 is called a pushout of \( (b, c) \) and \( D \) is called a pushout object of \( (b, c) \) if the following hold:

\[
\begin{array}{cc}
A & b \\
\downarrow c & \downarrow d \\
C & \downarrow f \\
\end{array}
\begin{array}{cc}
B & \\
\downarrow & \downarrow \\
D & \downarrow d' \\
\end{array}
\begin{array}{ccc}
 & D' & \\
\downarrow h & \searrow f' & \\
 & & \\
\end{array}
\]

Figure 2.3: Pushout of \( (b, c) \)

- Commutativity: \( b \circ d = c \circ f \), and
- Universal Property: For all objects \( D' \) and arrows \( d' : B \to D' \), and \( f' : C \to D' \), with \( b \circ d' = c \circ f' \), there exists a unique arrow \( h : D \to D' \) such that \( d \circ h = d' \) and \( f \circ h = f' \).

Moreover, given arrows \( b : A \to B \) and \( d : B \to D \), a pushout complement of \( (b, d) \) is a triple \( (C, c : A \to C, f : C \to D) \) such that \( (D, d, f) \) is a pushout of \( (b, c) \). In this case \( C \) is called a pushout complement object of \( (b, d) \).

### 2.2.2 Monoidal Category

In this subsection, we recall the definitions of monoidal category and symmetric monoidal category.

**Monoidal Category**

A monoidal category \cite{Kelly1982} \( MC = (C, \otimes, I, \alpha, l, r) \) consists of a category \( C \), a bi-functor \( \otimes : C \times C \to C \), an object \( I \) of \( C \), natural isomorphisms \( \alpha_{A,B,C} : (A \otimes B) \otimes C \cong (A \otimes (B \otimes C)) \). ...
A\otimes (B \otimes C), l : I \otimes A \simeq A and r : A \otimes I \simeq A, subject to two coherence axioms expressing
the commutativity of the diagrams in Fig. 2.4 and Fig. 2.5.

\[ ((A \otimes B) \otimes C) \otimes D \xrightarrow{\alpha_{A,B,C} \otimes Id_D} (A \otimes (B \otimes C)) \otimes D \]

\[ A \otimes ((B \otimes C) \otimes D) \xrightarrow{Id_A \otimes \alpha_{B,C,D}} \]

\[ (A \otimes B) \otimes (C \otimes D) \xrightarrow{\alpha_{A,B,C} \otimes Id_D} A \otimes (B \otimes (C \otimes D)) \]

Figure 2.4: Monoidal category — coherence axiom1

\[ (A \otimes I) \otimes B \xrightarrow{\alpha_{A,I,B}} A \otimes (I \otimes B) \]

\[ r \otimes Id_B \quad Id_A \otimes l \]

\[ A \otimes B \]

Figure 2.5: Monoidal category — coherence axiom2

Symmetric Monoidal Category

A symmetric monoidal category \( \text{SMC} = (MC, S) \) consists of a monoidal category \( MC \) and natural transformation \( S : A \otimes B \rightarrow B \otimes A \) subject to the coherence axioms
that can be expressed as the commutativity of the diagrams in Fig. 2.6, Fig. 2.7 and Fig. 2.8.
Figure 2.6: Symmetric monoidal category — associativity coherence axiom

Figure 2.7: Symmetric monoidal category — unit coherence

Figure 2.8: Symmetric monoidal category — inverse law
2.2.3 GS-Monoidal Category

A gs-monoidal category \( GSM = (SMC, \nabla, !) \) consists of a symmetric monoidal category \( SMC = (MC, S) \) and two transformations (which do not need to be natural) \( \nabla : A \to A \otimes A \) and \( ! : A \to I \) satisfying the coherence axioms in Fig. 2.9, Fig. 2.10 and Fig. 2.11. The monoidality axioms are shown in Fig. 2.12 and Fig. 2.13.

\[
\begin{align*}
A & \xrightarrow{\nabla} A \otimes A \\
\nabla & \downarrow \\
A \otimes A & \xrightarrow{Id_A \otimes \nabla} A \otimes (A \otimes A) \xrightarrow{\alpha_{A,A,A}^{-1}} (A \otimes A) \otimes A
\end{align*}
\]

Figure 2.9: gs-monoidal category — coherence axiom-1

\[
\begin{align*}
A & \xrightarrow{\nabla} A \otimes A \\
\nabla & \downarrow \\
A \otimes I & \xrightarrow{Id_A \otimes !} A \otimes I
\end{align*}
\]

Figure 2.10: gs-monoidal category — coherence axiom-2

\[
\begin{align*}
A & \xrightarrow{\nabla} A \otimes A \\
\nabla & \downarrow \\
A \otimes A & \xrightarrow{S_{A,A}} A \otimes A
\end{align*}
\]

Figure 2.11: gs-monoidal category — coherence axiom-3

GS-monoidal categories, proposed by Corradini and Gadducci (1999a), are a category-theoretic interface for programming “on top of” term graphs with sequential and
parallel composition. The “s” of “gs-monoidal” stands for “sharing”: For example, every input of $\nabla_k (F \otimes G)$ is shared by $F : k \rightarrow m$ and $G : k \rightarrow n$. The “g” of “gs-monoidal” stands for “garbage”: For example, in the gs-monoidal category for term graphs, all edges of a term graph $G : m \rightarrow n$ are garbage in the term graph $G_\sharp !_n$. An example is shown in Fig. 2.14. To the left is a term graph $G : 1 \rightarrow 1$ with one output
using the result of the S-edge. To the right is the term graph \( G_\frac{1}{2} \), with "no" output so that the S-edge and its result can be considered unused, i.e., "garbage".

![Diagram](image)

Figure 2.14: \( G \) and \( G_\frac{1}{2} \)

### 2.3 Algebraic Approach to Graph Transformation

Among the various approaches to graph transformation, the algebraic approaches (Corradini et al., 1997) are based on the concept of gluing of graphs, characterised by the use of categorial notions for the very basic definitions of graph transformation rules (or production), of matches (i.e., of occurrences of the left-hand side of a production in a graph), and of rule application (called direct derivations).

In the originally-motivated instances of the algebraic approaches, a graph is considered as a two-sorted algebra where the sets of vertices \( V \) and edges \( E \) are the carriers, while source \( s : E \to V \) and target \( t : E \to V \) are two unary operations. Moreover we may have label functions \( l_v : V \to L_v \) and \( l_e : E \to L_e \), where \( L_v \) and \( L_e \) are fixed label alphabets for vertices and edges.

A production \( p : L \to R \) defines a correspondence between elements of its left-hand side graph \( L \) and right-hand side graph \( R \), determining which nodes and edges have to be preserved by an application of \( p \), which have to be deleted, and which have to be created.

A match \( m : L \to A \) for \( p \) is a graph homomorphism mapping nodes and edges of \( L \) to \( A \) in such a way that the graph structure and the labels are preserved.

A direct derivation (a graph transformation step) is applying \( p \) to \( A \) and leading to a derived graph \( B \), if the match \( m \) fixes an occurrence of \( L \) in a given graph \( A \). Intuitively, \( B \) is obtained by replacing the occurrence of \( L \) in \( A \) by \( R \). Fig. 2.15 shows an example.
A fundamental fact is that graphs and graph morphisms form a category. This allows us to formulate most of the definitions, constructions and results in pure categorial terms.

Among algebraic approaches, DPO (Double PushOut) formally characterises direct derivations as double pushout diagrams in the category, see Fig. 2.16.

A production in the DPO approach is given by a pair $L \leftarrow G \rightarrow R$ of graph homomorphisms from a “gluing graph” $G$, and a direct derivation consists of two attached pushout diagrams of graphs and total graph morphisms. The host graph $H$ can be thought of as obtained from the given graph $A$ by deleting all elements of $A$ which have a pre-image in $L$, but none in $G$. Via the left diagram this deletion is described as an inverse insertion operation, while the right diagram models the actual insertion into $H$ of all elements of $R$ that do not have a pre-image in $G$. 
Chapter 3

Formal Presentation of Categories

In this chapter, we present our formalisation for the fundamental concepts of categories. The formalisation translates the mathematical concepts in Sect. 2.2 into data types which Agda accepts, and therewith performs the role as the specification for our term graph instance. With the formalisations, we also illustrate the basic Agda vocabulary and language mechanisms. We start with the formalisation for category in Sect. 3.1 followed by monoidal category in Sect. 3.2. In Sect. 3.3 we present the formalisation for gs-monoidal category.

3.1 Category

In this section, we present the formalisation for concepts of category.

Setoid

In Agda, a set with equality is typically modelled as a *setoid*, that is, as a carrier type equipped with an equivalence. We use the formalisation in the Agda standard library (Danielsson et al. [2013]) which is a record consisting of a Carrier set, a relation \(\_ \approx \_\) on that carrier, and a proof that the relation \(\_ \approx \_\) is an equivalence relation:

\[
\text{record Setoid } (c \ell : \text{Level}) : \text{Set} (\text{succ} (c \cup \ell)) \text{ where}
\]

\[
\text{field}
\]

\[
\text{Carrier : Set } c
\]

\[
\_ \approx \_ : \text{Rel Carrier } \ell
\]

\[
isEquivalence : \text{IsEquivalence } \_ \approx \_
\]

\[
\text{open } \text{IsEquivalence isEquivalence public}
\]
In Agda there is hierarchy of increasingly large types: The type of Set is Set_1, whose type is Set_2 and so on. This record is parameterised by two explicit arguments c and ℓ which represent a “level of Set” (and therefore have Agda type Level).

An Agda record is much like a datatype. The fields of the record are indicated by the keyword field presenting the necessary components and properties. Since Agda records are also modules, they can contain additional materials besides fields. The “open” clause makes the fields of the equivalence proof available as if they were fields of Setoid. This Agda feature enables incremental extension of smaller theories to larger theories at a very low notational cost.

Category

We use the RATH-Agda formalisation of category [Kahl 2011, 2017], where a category (Sect. 2.2.1) is presented as a record.

This record is first parameterised by an implicit argument i of type Level, and two explicit arguments j and k of type Level. The next argument Obj is explicit and stands for the collections of objects of the category. Note that i has been made an implicit argument because it can be inferred from type of the explicit argument Obj.

The first field Hom stands for the “hom-setoid” between two objects. Mor is the collection of morphisms between two objects which is derived from “hom-set”, ≈ is the equivalence for morphisms from the object A to B, which is derived from field Hom A B which is a Setoid. The composition operator _◦_ and the associativity law are then presented as fields in sequence. They are followed by identity arrow Id and the identity law.

In this sample code, we see that mathematical symbols such as _◦_ and ≈ are used in identifiers because Agda has full support for Unicode identifiers and keywords.

```
record Category {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊕ suc (j ⊕ k)) where
  field Hom : Obj → Obj → Setoid j k
  Mor : Obj → Obj → Set j
  Mor = λ A B → Setoid.Carrier (Hom A B)
  _≈_ = λ {A} {B} → Setoid._≈_ (Hom A B)
  field _◦_ _≈_ : {A B C : Obj} → Mor A B → Mor B C → Mor A C
  _≈_-cong : {A B C : Obj} {f1 f2 : Mor A B} {g1 g2 : Mor B C}
    → f1 ≈ f2 → g1 ≈ g2 → (f1 _◦_ g1) ≈ (f2 _◦_ g2)
  _≈_-assoc : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
    → ((f _◦_ g) _◦_ h) ≈ (f _◦_ (g _◦_ h))
  Id : {A : Obj} → Mor A A
  leftId : {A B : Obj} → {f : Mor A B} → (Id _◦_ f) ≈ f
  rightId : {A B : Obj} → {f : Mor A B} → (f _◦_ Id) ≈ f
```
Above is a monolithic characterisation of categories. Strictly speaking, if \( C \) is of type \( \text{Category j k Obj} \), then this means that “\( C \) is a category with elements of Obj as objects”, since \( \text{Obj} \) is a parameter of the record \( \text{Category} \) instead of a \( \text{field} \).

**Isomorphism**

We formalise *isomorphisms* between two objects \( A \) and \( B \) in a structure \( \text{Iso A B} \), which consists of a morphism \( f \) from \( A \) to \( B \), a morphism \( f^{-1} \) from \( B \) to \( A \), and the proofs that the morphism composition \( f \circ f^{-1} \) is equivalent to the identity morphism, and so is \( f^{-1} \circ f \):

```plaintext
record IsIso {A B : Obj} (f : Mor A B) : Set (j \& k) where
  infix 20 _⁻¹
  field
    _⁻¹ : Mor B A
    rightInverse : f ⁻¹ \& f ≈ Id
    leftInverse : f \& f⁻¹ ≈ Id

record Iso (A B : Obj) : Set (j \& k) where
  field
    isoMor : Mor A B
    isIso : IsIso isoMor
  open IsIso isIso public
  open Iso public
```

Note that \( C.'.IsIso.'.⁻¹ \) refers to the field \( _⁻¹ \) of a \( C.'IsIso.' \). Also note, that if \( g : C.'IsIso.' \), then \( g C.'IsIso.'⁻¹ \) is shorthand for the application of field selector \( C.'IsIso.'._⁻¹ g \).

**Coproduct**

Below we supply the formalisation for a categorial construction — coproduct:

A *coproduct* of two objects \( A \) and \( B \) is an object \( S \), together with two “injection” arrows \( \iota : A \to S \) and \( \kappa : B \to S \), such that for any object \( C \) and pair of arrows \( F : A \to C \) and \( G : B \to C \) there is exactly one arrow \( F \triangle G : S \to C \) making the following diagram commute as shown in Fig. 3.1:

For the formalisation of *coproduct*, we first present the structure \( \text{CoCone2Univ} \) for two cospans \( A \to X \leftarrow B \) and \( A \to Z \leftarrow B \). It consists of:

- The morphism \( \text{univMor} \) from \( X \) to \( Z \)
- The commutativities \( \text{univMor-factors-left} \) and \( \text{univMor-factors-right} \)
The uniqueness of \( \text{univMor} \)

\[
\text{record CoCone2Univ} \{ A \ B \ X : \text{Obj} \} (R : \text{Mor} \ A X) (S : \text{Mor} \ B X) \\
\{ Z : \text{Obj} \} (R' : \text{Mor} \ A Z) (S' : \text{Mor} \ B Z) : \text{Set} \ (i \cup j \cup k) \text{ where} \\
\text{field} \\
\text{univMor} : \text{Mor} \ X Z \\
\text{univMor-factors-left} : R \cup \text{univMor} \approx R' \\
\text{univMor-factors-right} : S \cup \text{univMor} \approx S' \\
\text{univMor-unique} : \{ V : \text{Mor} \ X Z \} \to R \cup V \approx R' \to S \cup V \approx S' \to V \approx \text{univMor}
\]

The commutative diagram is shown in Fig. 3.2.

Given \( \iota : \text{Mor} \ A S \) and \( \kappa : \text{Mor} \ B S \), \( \text{IsCoproduct} \ \iota \ \kappa \) is a function type: It results in a \( \text{CoCone2Univ} \ \iota \ \kappa \ \{ Z \} \ F \ G \) from \( Z : \text{Obj} \), \( F : \text{Mor} \ A Z \) and \( G : \text{Mor} \ B Z \).

\[
\text{IsCoproduct} : \{ A \ B \ S : \text{Obj} \} (\iota : \text{Mor} \ A S) (\kappa : \text{Mor} \ B S) \to \text{Set} \ (i \cup j \cup k) \\
\text{IsCoproduct} \ \{ A \} \ \{ B \} \ \{ S \} \ \iota \ \kappa = \{ Z : \text{Obj} \} (\text{F} : \text{Mor} \ A Z) (\text{G} : \text{Mor} \ B Z) \\
\quad \to \text{CoCone2Univ} \ \iota \ \kappa \ \{ Z \} \ F \ G
\]
Next we present module \texttt{IsCoproduct} which renames the components of an instance \texttt{CoCone2Univ}:

\begin{verbatim}
module IsCoproduct \{ A B S : Obj \} \{ \iota : Mor A S \} \{ \kappa : Mor B S \}
(isCoproduct : IsCoproduct \iota \kappa) where

private
module Univ \{ C : Obj \} \{ F : Mor A C \} \{ G : Mor B C \}
= CoCone2Univ (isCoproduct \{ C \} F G)

open Univ public using () renaming

\end{verbatim}

Then coproduct is formalised as the record \texttt{Coproduct} consisting of components \( S, \iota \) and \( \kappa \) where \( \iota \) and \( \kappa \) composite a \texttt{CoCone2Univ} through \texttt{isCoproduct : IsCoproduct \iota \kappa}.

\begin{verbatim}
record Coproduct (A B : Obj) : Set \{ i j k \} where

field

S : Obj
\iota : Mor A S
\kappa : Mor B S
isCoproduct : IsCoproduct \iota \kappa

\end{verbatim}

Note that we also organise what in conventional category theory would be called a “choice of coproducts” as the record type \texttt{HasCoproducts} below, providing for two objects \( A \) and \( B \) a coproduct object \( A \sqcup B \), together with two injection arrows \( \iota : A \to A \sqcup B \) and \( \kappa : B \to A \sqcup B \), such that for any object \( C \) and pair of arrows \( F : A \to C \) and \( G : B \to C \) there is exactly one arrow \( F \triangleright G : A \sqcup B \to C \) making the following diagram commute as shown in Fig. 3.3.

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node (A) at (0,0) {A};
\node (B) at (2,0) {B};
\node (C) at (1,2) {C};
\node (D) at (1,0) {A \sqcup B};
\node (E) at (1,1) {F \triangleright G};
\draw[->] (A) to node[below] {$\iota$} (D);
\draw[->] (B) to node[below] {$\kappa$} (D);
\draw[->] (D) to node[right] {$F$} (C);
\draw[->] (D) to node[left] {$G$} (C);
\end{tikzpicture}
\caption{Choice of coproducts}
\end{figure}
We introduce the full definition here since we will be using in particular \( \odot \), \( \iota \), and \( \kappa \) in the further development. The function \texttt{coproduct} included below most directly corresponds to a “choice of coproducts”.

\[
\text{record HasCoproducts : Set (i \circ j \circ k) where}
\]
\[
\quad \text{infixr 3 \( \odot \)}
\]
\[
\quad \text{field}
\]
\[
\quad \quad \odot : \text{Obj} \to \text{Obj} \to \text{Obj}
\]
\[
\quad \quad \iota : \{A B : \text{Obj}\} \to \text{Mor A (A \oplus B)}
\]
\[
\quad \quad \kappa : \{A B : \text{Obj}\} \to \text{Mor B (A \oplus B)}
\]
\[
\quad \quad \text{isCoproduct} : \{A B : \text{Obj}\} \to \text{IsCoproduct \{A\} \{B\} \iota \kappa}
\]
\[
\text{module _ \{A B : \text{Obj}\} where}
\]
\[
\quad \text{open IsCoproduct (isCoproduct \{A\} \{B\}) public}
\]
\[
\quad \text{coproduct : (A B : Obj) \to Coproduct A B}
\]
\[
\quad \text{coproduct A B = record \{obj = A \oplus B; \iota = \iota; \kappa = \kappa; isCoproduct = isCoproduct\}}
\]

Then we supply a coproduct construction \texttt{Coproduct-\(\odot\)-Iso}. First we make the components of \texttt{Coproduct S\textsubscript{1}} available through:

\[
\text{private}
\]
\[
\text{module S\textsubscript{1} = Coproduct S\textsubscript{1}}
\]

Then \texttt{Coproduct-\(\odot\)-Iso} constructs a coproduct through \( \Phi : \text{Iso S\textsubscript{1}.obj S\textsubscript{2}} \) which is an isomorphism between \( S\textsubscript{1}.\text{Obj} \) (which is the field \texttt{Obj} of \texttt{S\textsubscript{1}}) and \( S\textsubscript{2} \).

\[
\text{module _ \{A B : \text{Obj}\} \{S\textsubscript{1} : \text{Coproduct A B}\} where}
\]
\[
\quad \text{private}
\]
\[
\quad \quad \text{module S\textsubscript{1} = Coproduct S\textsubscript{1}}
\]
\[
\quad \quad \text{Coproduct-\(\odot\)-Iso : \{S\textsubscript{2} : \text{Obj}\} \to (\Phi : \text{Iso S\textsubscript{1}.obj S\textsubscript{2}})
\]
\[
\quad \quad \quad \to \text{Coproduct A B}
\]
\[
\quad \quad \text{Coproduct-\(\odot\)-Iso \{S\textsubscript{2}\} \Phi = record}
\]
\[
\quad \quad \quad \{\text{obj = S\textsubscript{2}}
\]
\[
\quad \quad \quad \quad : \iota = S\textsubscript{1.}\iota \circ \text{isoMor } \Phi
\]
\[
\quad \quad \quad \quad : \kappa = S\textsubscript{1.}\kappa \circ \text{isoMor } \Phi
\]
\[
\quad \quad \quad \quad : \text{isCoproduct = IsCoproduct-\(\odot\)-Iso S\textsubscript{1}.isCoproduct } \Phi
\]
\[
\quad \}
\]

In the constructed coproduct, \texttt{Obj} is \( S\textsubscript{2} \), the injections \( \iota \) and \( \kappa \) are constructed as the injections of \( S\textsubscript{1} \) composed with \( \Phi \). Isomorphism is what we understand as “equivalence of objects” in category theory.

### 3.2 Monoidal Category

Our formalisation of monoidal categories follows the definition by [Mac Lane (1971)] which includes natural isomorphisms instead of object equations. We choose to for-
malise non-strict monoidal category because these are more friendly in Agda formalisation.

In the following formalisation of the materials in Sect. 2.2 we do not present the definitions of \texttt{Bifunctor} (bi-functor), \texttt{NatIso} (natural isomorphism), etc. in detail. The last line of the code makes in particular the object and morphism components of \( \otimes \) available as \( \otimes \circ \) and \( \otimes \circ m \) respectively.

\[
\text{record} \quad \text{MonoidalCategory} \{ i : \text{Level} \} (j k : \text{Level}) (\text{Obj} : \text{Set} i) : \text{Set} (i \circ \text{suc} j \circ \text{suc} k) \\
\text{where} \\
\text{field} \\
\quad \text{category} : \text{Category} j k \text{Obj} \\
\quad \otimes : \text{Bifunctor category category category} \\
\text{open} \text{ OTimes} \otimes
\]

Associativity is introduced as a natural isomorphism between the two functors that apply the object part of \( \otimes \circ \) to the triple product of the category nesting to the left respectively to the right (omitting the details of these functors and of \text{category}^3).

\[
\otimes \text{-NestL} : \text{Functor category}^3 \text{category} \\
\otimes \text{-NestL} = \text{record} \{ \text{obj} = \lambda A \rightarrow \text{obj} (A 0 3 \otimes A 1 3) (A 2 3); \ldots \} \\
\otimes \text{-NestR} : \text{Functor category}^3 \text{category} \\
\otimes \text{-NestR} = \text{record} \{ \text{obj} = \lambda A \rightarrow \text{obj} (A 0 3) (A 1 3 \otimes A 2 3); \ldots \} \\
\text{field} \otimes \text{-Assoc} : \text{NatIso} \otimes \text{-NestL} \otimes \text{-NestR}
\]

The monoidal left- and right-unit laws are also embodied in natural isomorphisms:

\[
\text{field} \\
\quad \text{1} : \text{Obj} \\
\text{LeftUnit} : \text{Functor category category} \\
\text{LeftUnit} = \text{record} \{ \text{obj} = \lambda A \rightarrow \text{1} \otimes A; \ldots \} \\
\text{RightUnit} : \text{Functor category category} \\
\text{RightUnit} = \text{record} \{ \text{obj} = \lambda A \rightarrow A \otimes \text{1}; \ldots \} \\
\text{field} \\
\quad \otimes \text{-LeftUnit} : \text{NatIso LeftUnit (Identity _)} \\
\quad \otimes \text{-RightUnit} : \text{NatIso RightUnit (Identity _)}
\]

The three coherence conditions are:

\[
\otimes \text{-assoc-pentagon} : \{ A B C D : \text{Obj} \} \\
\rightarrow \otimes \text{-assoc} \{ A \otimes B \} \{ C \} \{ D \}; \otimes \text{-assoc} \{ A \} \{ B \} \{ C \otimes D \} \\
\approx (\otimes \text{-assoc} \circ m \text{ ld} \{ D \}) \circ \otimes \text{-assoc} \circ m \circ \text{ ld} \{ A \} \circ (\otimes \text{-assoc}) \\
\otimes \text{-triangle} : \{ A B : \text{Obj} \} \rightarrow \otimes \text{-assoc} \{ A \} \{ 1 \} \{ B \}; \circ \text{ ld} \{ A \} \circ m \circ \text{ leftUnit} \\
\approx \otimes \text{-rightUnit} \circ m \text{ ld} \{ B \} \\
\otimes \text{-leftUnit-1} : \otimes \text{-leftUnit} \{ 1 \} \approx \otimes \text{-rightUnit} \{ 1 \}
\]
Here, $\otimes$-leftUnit : \{A : Obj\} $\rightarrow$ Mor (\(\mathbb{1}\) $\otimes$ A) A is the arrow from \(\mathbb{1}\) $\otimes$ A to A for any A : Obj, which is the forward direction of the natural isomorphism \(l : I \otimes A \approx A\) in Sect. 2.2.2. The other direction of \(l\) is formalised as $\otimes$-leftUnit\(^{-1}\) : \{A : Obj\} $\rightarrow$ Mor A (\(\mathbb{1}\) $\otimes$ A). Similarly for $\otimes$-rightUnit and $\otimes$-rightUnit\(^{-1}\).

For symmetric monoidal categories, we follow again Mac Lane (1971), which is non-strict:

\[
\text{record } \text{MonCatSym} \{i j k : \text{Level}\} \{\text{Obj} : \text{Set} i\} \\
(MC : \text{MonoidalCategory} j k \text{Obj}) : \text{Set} (i \mathbin{\text{uni228D}} j \mathbin{\text{uni228D}} k) \\
\text{field} \\
\text{swap} : \{A B : \text{Obj}\} \rightarrow \text{Mor} (A \mathbin{\otimes} B) (B \mathbin{\otimes} A) \\
\text{swap-natural} : \{A_1 A_2 B_1 B_2 : \text{Obj}\} \{F : \text{Mor} A_1 A_2\} \{G : \text{Mor} B_1 B_2\} \\
\rightarrow (F \mathbin{\otimes} m G) \triangleleft \text{swap} \{A_2\} \{B_2\} \approx \text{swap} \{A_1\} \{B_1\} \mathbin{\otimes} (G \mathbin{\otimes} m F) \\
\text{swap-cancel} : \{A B : \text{Obj}\} \rightarrow \text{swap} \{A\} \{B\} \triangleleft \text{swap} \{B\} \{A\} \approx \text{id} \\
\text{swap-unit} : \text{swap} (\mathbb{1}) (\mathbb{1}) \approx \text{id} (\mathbb{1} \mathbin{\otimes} \mathbb{1}) \\
\text{swap-monoidal} : \{A B C : \text{Obj}\} \\
\rightarrow (\text{id} \{A\} \mathbin{\otimes} m \text{swap} \{B\} \{C\}) \mathbin{\otimes} \text{assocL} \{A\} \{C\} \{B\} \\
\approx (\text{swap} \{A\} \{C\} \mathbin{\otimes} m \text{id} \{B\}) \\
\approx (\otimes \text{assocL} \{A\} \{B\} \{C\}) \approx \text{swap} (A \mathbin{\otimes} B) \{C\} \\
\approx (\otimes \text{assocL} \{C\} \{A\} \{B\}) \\
\text{swap-\otimes-leftUnit} : \{A : \text{Obj}\} \rightarrow \text{swap} \otimes \otimes\text{leftUnit} \approx \otimes\text{rightUnit} \{A\}
\]

3.3 GS-Monoidal Category

Then we proceed to the “gs-monoidal category” following Corradini and Gadducci (1999a) for the coherence conditions but adapting them to the non-strict monoidal setting. The “g” of “gs-monoidal” represented as \(!\) stands for “garbage” (see Sect. 2.2.3). Below is the monoidal category equipped with termination transformation (which does not need to be natural):

\[
\text{record } \text{MonCatG} \{i j k : \text{Level}\} \{\text{Obj} : \text{Set} i\} \\
(M \text{onCat} : \text{MonoidalCategory} j k \text{Obj}) : \text{Set} (i \mathbin{\text{uni228D}} j \mathbin{\text{uni228D}} k) \\
\text{field} \\
\text{l\text{-unit}} : \{A : \text{Obj}\} \rightarrow \text{Mor} A \mathbb{1} \\
\text{l\text{-unit}} : ! (\mathbb{1}) \approx \text{id} (\mathbb{1}) \\
\text{l\text{-monoidal}} : \{A B : \text{Obj}\} \rightarrow ! \{A \mathbin{\otimes} B\} \approx (! \{A\} \mathbin{\otimes} m \{B\}) \mathbin{\otimes} \text{leftUnit} \{\mathbb{1}\}
\]

We supply the symmetric monoidal category equipped with the duplication transformation \(\triangleleft\) which stands for “sharing” (see Sect. 2.2.3) (which does not need to be natural too):
record MonCatS {i j k : Level} {Obj : Set i}
  (monCat : MonoidalCategory j k Obj)
  (monCatSym : MonCatSym monCat) : Set (i \uplus j \uplus k) where

\[\n\begin{align*}
\n\n\triangledown & : \{A : Obj\} \rightarrow Mor A (A \otimes A) \\
\n\triangledown \text{-} unit & : \triangledown \{1\} \approx \otimes \text{-}leftUnit^{-1} \{1\} \\
\n\triangledown \text{-} assoc & : \{A : Obj\} \\
& \quad \rightarrow \triangledown \{A\} \uplus (\triangledown \{A\} \otimes m \text{ld} \{A\}) \\
& \quad \approx \triangledown \{A\} \uplus (\text{ld} \{A\} \otimes m \triangledown \{A\}) \uplus \otimes \text{-}assocL \{A\} \{A\} \{A\} \\
\n\triangledown \text{-}\text{swap} & : \{A : Obj\} \rightarrow \triangledown \{A\} \uplus \text{swap} \{A\} \{A\} \approx \triangledown \{A\} \\
\n\triangledown \text{-} monoidal & : \{A B : Obj\} \\
& \quad \rightarrow \triangledown \{A \otimes B\} \uplus \otimes \text{-}assoc \{A\} \{B\} \{A \otimes B\} \\
& \quad \approx (\triangledown \{A\} \otimes m \triangledown \{B\}) \uplus \otimes \text{-}assoc \{A\} \{B\} \{B\} \uplus \text{ld} \{A\} \otimes m \otimes \text{-}assocL \{A\} \{B\} \{B\}
\end{align*}\]

With the above formalisation, we offer the record MonCatGS to organise gs-monoidal categories as an extension of symmetric monoidal categories with a condition \(\triangledown \text{-} rightInv\) relating termination and duplication together:

record MonCatGS {i j k : Level} {Obj : Set i}
  (monCat : MonoidalCategory j k Obj)
  (monCatSym : MonCatSym monCat) : Set (i \uplus j \uplus k) where

\[\n\begin{align*}
\n\text{monCatG} & : \text{MonCatG} \text{monCat} \\
\text{monCatS} & : \text{MonCatS} \text{monCat} \text{monCatSym} \\
\triangledown \text{-}rightInv & : \{A : Obj\} \rightarrow \triangledown \{A\} \uplus (\text{ld} \{A\} \otimes m \{A\}) \approx \otimes \text{-}rightUnit^{-1}
\end{align*}\]

Finally, gs-monoidal category is formalised as below:

record GSMonoidalCategory {i : Level} {j k : Level} (Obj : Set i)
  : Set (i \uplus \text{suc} j \uplus \text{suc} k) where

\[\n\begin{align*}
\n\text{monCat} & : \text{MonoidalCategory} j k \text{Obj} \\
\text{monCatSym} & : \text{MonCatSym} \text{monCat} \\
\text{monCatGS} & : \text{MonCatGS} \text{monCat} \text{monCatSym}
\end{align*}\]

It contains a monoidal category \text{monCat}, a symmetric monoidal category \text{monCatSym} built from \text{monCat} and \text{monCatGS} which is built from \text{monCat} and \text{monCatSym} as a symmetric monoidal category equipped with “garbage collection” and “sharing”. 

Chapter 4

Overview of the Formalisation

Now that we have introduced most of the basic definitions of the setting we are building on, we outline the plan of the remaining chapters in more detail.

In Chapter 5, we will present term graphs as abstract directed hypergraphs (ADHGs) on a suitable level of abstraction from the concrete choice of representation of node and edge sets. Through such abstraction, what we obtain is generality, e.g., the underlying category can be instantiated to not only sets and functions, but also to settings other than that of (all) sets and functions. Then the dependently-typed internal aspects of term graphs will lead us to a novel category-theoretic abstraction of dependent objects, which forms an “internal interface” between term graphs and the implementations of their node and edge sets, and their labelling and connection functions. Our representation will be more general than the term graphs in the spirit of “jungles” (Hoffmann and Plump 1991; Corradini and Rossi 1993) (where the function for edge output needs to be a bijection). That is, our representation can have some “empty area” to accommodate the difference between the internal structures of the two sides of a rule for the formalisation of term graph rewriting (see Sect. 9.1.1). Also in that chapter, we will present our formalisation of ADHG homomorphism which preserves not only the internal structure of graphs, but also the graph input and output interfaces. Note that these ADHG homomorphisms are not the conventional concept of term graph homomorphisms (Corradini and Gadducci 1999a) (which we will formalise as Matching) which does not require preservation of graph input and output interfaces respectively. In the same chapter, we will present the formalisation of categories of ADHGs which lays the foundation for the development of term graph rewriting and correctness in Chapter 9 i.e., the category of ADHGs and homomorphisms, of ADHGs and Matchings, of term graphs where the morphisms are ADHGs, and the gs-monoidal category of ADHGs which provides combinators for term graph assembly and manipulation, and laws of reasoning.

After we define GSME (gs-monoidal category expressions) in Chapter 6 as an induc-
tively defined data type for denoting the result of term graph decomposition, we will proceed to develop an algorithm in Chapter 7 for acyclic term graph decomposition. In Sect. 7.1, we will see this algorithm is an essential part of the functor from the gs-monoidal category of term graphs to the gs-monoidal category chosen as semantics, which is the way to obtain the semantics of term graphs. We will present the function type for the algorithm and illustrate the working mechanism in Sect. 7.2. It will be followed by Sect. 7.3 where we present the implementation details for the algorithm.

Our ultimate goal is to supply a trusted (correctness guaranteed) tool for term graph rewriting (see Sect. 2.1). In Chapter 9 we will adopt a variant of the standard DPO approach (Kahl, 1997) to present a categorial formalisation of term graph rewriting. We are also interested in the semantics preservation of term graph rewriting. Corradini and Gadducci (1999a) shows us a road to the categorial formalisation of the semantics for term graphs, with that we will also investigate the semantics preservation of term graph rewriting in the DPO approach which is listed as an open problem in Corradini and Gadducci (1999a).

For the sake of conciseness, this thesis only supplies the essential code fragments. However, the complete Agda modules are available on-line (Zhao, 2018) with the following contents:

- The representation of a restricted version of term graphs as abstract directed hypergraphs "(ADHG)", with various categories for term graphs
- The gs-monoidal expressions "(GSME)" for denoting the result of the decomposition of term graphs
- The decomposition of term graphs with the proof of the correctness
- The formalisation of term graph rewriting through the variant of the standard DPO approach, and the semantics preservation of the rewriting
Chapter 5

Abstract Directed HyperGraphs and Categories

Our categorically formalised type of term graphs will be referred to as ADHG (Abstract Directed HyperGraph), because it is formalised within a parameterised underlying category instead of using Sets. With the abstract category (which will be understood to be some category “of sets and functions”), the development can focus on higher-level structures and the formalisation of properties, which are more concise this way. The abstract category can be instantiated to not only sets and functions, but also settings other than that of (all) sets and functions. The abstract category can be instantiated for the purpose of executing graph transformation on concrete graph structures. For example, for the “proof-of-concept” in Chapter 7 we use an instantiation with vecCategory (which is a category available in the library RATH-Agda). In vecCategory, the collection of objects are natural numbers and the morphisms from objects A to B are size-“A” vectors of finite numbers of Fin B in the shape of Vec (Fin B) A. The vector type constructor Vec is defined in the Agda standard library representing the functions between the denoted datatypes as certain container datastructures.

Our approach of arriving at the ADHG formalisation is implementation-oriented and incremental: We start with a formalisation of term graphs using sets, and discuss how it is not easily adapted for finite node and edge sets. Therefore, we propose to replace sets and functions by the objects and morphisms in an underlying category. However, this still does not work for having node and edge sets as the shape “Fin n” which is a type containing exactly the natural numbers less than n. Instead, we refine the setting from a single category to a category embedded in a semigroupoid. Then, in order to formalise the dependently-typed internal aspects of term graphs within the abstraction over a category, we adopt the novel category-theoretic abstraction of dependent objects. It can be implemented by concrete data structures, like the extension of vecCategory, and at the same time offers sufficient expressiveness for
elegant formalisations of and proofs about dependently-typed components.

While these categorial abstraction form the “internal interface” between the definition of term graphs and the implementations, the gs-monoidal categories of [Corradini and Gadducci (1999a)] form a useful “external interface” of term graphs, providing combinatorial for term graph assembly and manipulation, and laws of reasoning. We adopt the non-strict version of monoidal categories instead of the strict version, because we find it turns out to be much less burdensome to use for reasoning in the type system of Agda. (Therefore we already presented monoidal categories in the non-strict version in Chapter 2.)

In this chapter, we present our formalisation of ADHG$s$, as well as categories of ADHG$s$: For reference, we first present an attempt to formalise term graphs as directed hypergraphs in the conventional set-based setting. Then the abstraction from the *sets* of edges and inner nodes to *objects* of a parameter category, together with the arity-dependent term graph components, leads us to the categorial abstraction of dependent objects in Sect. 5.2 which is followed by our formalisation of ADHG in Sect. 5.4. Next we proceed to homomorphisms and the category of ADHG$s$ and homomorphisms in Sect. 5.5. In Sect. 5.6 we also present Matching which is a variant of homomorphism, and also the category of ADHG$s$ and Matchings. We also present the category of term graphs in Sect. 5.7 where the morphisms are ADHG$s$. One interesting aspect of the category of ADHG homomorphisms and the category of ADHG$s$ is that their connection can be modelled as a 2-category [Barr and Wells, 1990]. Finally, we present the gs-monoidal category of ADHG$s$ in Sect. 5.8. The full formalisation is available on-line (Zhao, 2018).

5.1 Directed HyperGraph Implemented with Sets

In this section, we present the set-based implementation for directed hypergraphs as an initial taste for our incrementally addressed concerns about term graph formalisation in a dependently-typed setting.

The record datatype $\text{DHG}_0$ implements a simple definition of the *directed hypergraphs* of Sect. 2.1 and it is intentionally kept close to conventional mathematical formulations. For the time being, we work in the context of an arity-indexed label type $\text{Label}_0 : \mathbb{N} \to \text{Set}$.

```plaintext
record DHG_0 (m n : \mathbb{N}) : \text{Set}_1 where
  field Inner : \text{Set}
  Node = Fin m \uplus Inner
  field output : Vec Node n
    Edge : \text{Set}
    eOut : Edge \to Inner
```

31
In this formalisation, a \( \text{DHG}_0 \) contains the following components:

- There are two natural numbers \( m \) and \( n \) as parameters of the \( \text{DHG}_0 \) datatype: \( m \) represents the number of inputs or variables of the graph, and \( n \) represents the number of outputs or results. (In rooted term graphs, we have \( n = 1 \).)

- \( \text{Node} \) is the set of all nodes including the variables and inner nodes. It is constructed as the disjoint union of \( \text{Inner} \) with the set of input positions, which is obtained as \( \text{Fin} m \) — the type of natural numbers smaller than \( m \). It is presented as a derived component from \( m \) and \( \text{Inner} \), and will be used in the type of several further fields.

- \( \text{output} \) is a size-\( n \) vector of \( \text{Node}s \) specifying a sequence of \( \text{Node}s \) as graph output nodes. Both variables and inner nodes can be output nodes of the graph. The type of the field \( \text{output} \) therefore depends on the field \( \text{Inner} \) and on the parameter \( m \).

- \( \text{Edge} \) is the set of hyperedges.

- \( \text{eOut} \) specifies the output inner node of each edge.

- \( \text{eArity} \) specifies the arity function for edges.

- \( \text{eLabel} \) specifies the function for edge labels.

- \( \text{eln} \) specifies the function for edge inputs.

- \( \text{input} \) is a size-\( m \) vector of \( \text{Node}s \) specifying the graph input (variables). As usual for term graphs, variables (i.e., graph input nodes) have no names, since input node identity is variable identity. \( \text{input} \) is constructed using \( \text{allFin} m \), which is the vector (i.e., array) containing all \( m \) elements of the set \( \text{Fin} m \) in sequence, i.e., \( 0, 1, 2, \ldots, m-1 \). It is presented as a derived component from \( m \) as the injection of variables into \( \text{Node} \).

This definition is easily extended to one for jungles by specifying that \( \text{eOut} \) is a bijection.

\[
\text{record Jungle}_0 (m, n : \mathbb{N}) : \text{Set}_1 \quad \text{where}
\]
\[
\begin{align*}
\text{field} & \quad \text{Base} : \text{DHG}_0 m n \\
\text{open} & \quad \text{DHG}_0 \text{Base} \\
\text{field} & \quad \text{eOutIso} : \text{IsIso eOut}
\end{align*}
\]

We are interested in finite term graphs, i.e., the sets of inner nodes and edges are finite. Trying to \textit{externally} impose on \( \text{DHG}_0 \) (without changing its definition) a restriction
to finite carrier set would make the resulting data type rather unwieldy to use. I.e., in order to reuse DHG₀ properties, we have to supply the proofs that they hold under the restriction.

5.2 Category Embedded in Semigroupoid

In order to obtain a definition for finite directed hypergraphs, instead of introducing restrictions on DHG₀, the natural approach is to replace Set in DHG₀ with a parameter. At the first step, we propose to use category components, i.e., replace Set with the collection of objects of a parameter category C. Therefore, Edge and Inner are replaced with C-objects, and functions are replaced with C-morphisms. For consistency, the graph input and output arities are also turned into C-objects.

Now we consider the implementations. A graph library implementer may wish to restrict in particular the node set and edge set to finite number sets in the shape of Fin n, where Fin n is the set {0, ..., n - 1} of the first n natural numbers. A good candidate for the parameter category is vecCategory, where each object is a natural number and a morphism from objects A to B is a vector in the shape of Vec (Fin B) A. In this instantiation, the edge set is a natural number n representing a finite number set Fin n, similar for the set of inner nodes. Unfortunately, this instantiation does not work for all of the functions describing the internal structure of term graphs. E.g., the graph output function and edge output function can be implemented as vectors of finite numbers, but the edge arity function can not be implemented this way. The edge arity function is from Edge to the set of infinite natural numbers N because we do not restrict the upper-bound for arity. In order to have N as the target of a morphism, N (or something representing N) must be an object of the category. However, bringing in N as an object of the category will automatically cause a problem: The identity morphism Id {N} : Mor N N must exist due to the fact that N is an object. But such a morphism can not be represented as an element of Vec (Fin B) A. Therefore the function for edge arities cannot be typed as a morphism in the assumed setting for our intended instantiation above.

We know that semigroupoids are to categories as semigroups are to monoids — no identities are assumed. We therefore refine the parameter setting from a category to a category C embedded in a semigroupoid SG of “arbitrary sets” via a full and faithful functor “F”. The category C will be understood as a category of finite sets. The semigroupoid SG is intended to be a “minimal” semigroupoid for all sets as objects (The objects represent the set of natural numbers, sets of function symbols, node vectors, and the F-images of C-objects. We will talk about details in Sect. 5.3). I.e., with the instantiation of C to vecCategory, SG can be implemented to accommodate not only the finite number sets from C, but also the set of natural numbers, the finite
number vectors, the vector of function symbols. We restrict the SG-morphisms to starting only from finite sets \( [\text{Kahl}, 2008] \), which are F-images of C-objects. Then edge arity function is a SG-morphism from the F-image of Edge to the SG-object representing the set of natural numbers. I.e., in the intended instantiation mentioned above, the edge arity function is a vector of natural numbers.

We present the refined setting as `DepObjBase`:

```plaintext
record DepObjBase (ℓ₀₁ ℓ₀₂ ℓ₁ ℓ₂₁ ℓ₂₂ : Level)
  : Set (/\{0\} (ℓ₀₁ \cup ℓ₀₂ \cup ℓ₁ \cup ℓ₂₁ \cup ℓ₂₂)) where
field
  Obj₁ : Set ℓ₀₁
  C : Category ℓ₁ ℓ₂₁ Obj₁
  hasStrictInit : CatFinColimits.HasStrictInitialObject C
  hasCoproducts : CatFinColimits.HasCoproducts C
  hasTerm : CatFinLimits.HasTerminalObject C
  Obj₂ : Set ℓ₀₂
  SG : Semigroupoid ℓ₂₁ ℓ₂₂ Obj₂
  F : CatSGFunctor C SG
  FFF : SGF.IsFullAndFaithful F
  presId : SGF.PreservesIdentity F
  presCoproduct : SGF.PreservesCoproduct F
  presInit : SGF.PreservesInitialObj F
```

In other words, `DepObjBase` is a setting where:

- `Obj₁` is a type used as the collection of objects in category C, where C has strict initial object, coproducts and a terminal object.
- `Obj₂` is a type used as the collection of objects in semigroupoid SG.
- `Functor F` from C to SG is full and faithful, and preserves identity morphisms, coproducts and the initial object of C. This functor is understood as an embedding of C into SG.

Next we introduce some notations and syntax for this setting:

Let `Mor₁` denote the type of C-morphisms and `Mor₂` denote the type of SG-morphisms. Let A be a C-object and B be a SG-object, we use \( A \rightarrow B \) to denote the SG-morphisms from the F-image of A to B. See eArity in ADHG [Sect. 5.4] as an example.

\[
\_ \rightarrow \_ : (A : \text{Obj}_1) (B : \text{Obj}_2) \rightarrow \text{Set} \_{\ell_2}
\]

\[
A \rightarrow B = \text{Mor}_2 (F.\text{obj} A) B
\]

We supply pre-composition \( \_ \rightarrow \_ \) which generates an SG-morphism by composing the F-image of C-morphism \( f \) with the SG-morphism \( U \). See ADHGMor [Sect. 5.5] as an example.
\[ \_ \triangleright \_ : \{ X, Y : \text{Obj}_1 \} \{ Z : \text{Obj}_2 \} (f : \text{Mor}_1 X \times Y) (U : Y \triangleright Z) \rightarrow X \triangleright Z \]
\[ \_ \triangleright \_ f U = F \circ f \triangleright_2 U \]

## 5.3 Categoric Abstraction of Dependent Objects

Term graphs are naturally considered as dependently typed as soon as the signature of term graph constructors is a parameter, since the length of the argument list of a term graph constructor application depends on that constructor.

The static verification power of a dependently-typed programming language allows us to make the low-level implementation of term graphs safer, by using a type constraint to connect the arity of constructors / labels with the number of arguments in their applications.

Recall in DHG\(_0\) the types of eLabel and eIn are “dependent function types”. In general, the type “\((x : X) \rightarrow Tx\)” is the type of functions that map each argument \(x\) to a result of type \(Tx\). That is, not only the value of the result depends on the value of the argument, but also the type of the result depends on the value of the argument.

An appropriate category-theoretic abstraction for such dependent functions is supplied for example with the concept of type-category as described by Pitts (2001). We need to integrate this aspect into our use of semigroupoid which is part of the DepObjBase setting. We know that we want dependent types for Label\(_0\) and Vec Node (see DHG\(_0\)), but we do not want to make assumptions about possibly other dependent types.

We therefore introduce the concept of dependent object over an “index” object of a semigroupoid, which can be thought of as an individual building block of a type-category as described by Pitts (2001).

A “dependent object” is a structure over a SG-object called “index” (argument \(\mathbb{A}\) in record DepObj). It contains a SG-object TotalObj as a representative for it. It has a function \(\text{ind}\) extracting an SG-morphism \(X \rightarrow \mathbb{A}\) from \(X \rightarrow \text{TotalObj}\) for every C-object \(X\). For example, Label is assumed as a dependent object over natural numbers for ADHG (Sect. 3.4).

```verbatim
record DepObj { \ell_0_1 \ell_0_2 \ell_c_1 \ell_c_2 \ell_s_1 \ell_s_2 : \text{Level} }
  (Base : DepObjBase \ell_0_1 \ell_0_2 \ell_c_1 \ell_c_2 \ell_s_1 \ell_s_2)
  (\mathbb{A} : \text{DepObjBase.Obj}_2 \text{Base}) -- “anchor” of the dependent object
  : \text{Set} (\ell_0_1 \cup \ell_0_2 \cup \ell_c_1 \cup \ell_c_2 \cup \ell_s_1 \cup \ell_s_2)

where
  field TotalObj : \text{Obj}_2
  ind : \{ X : \text{Obj}_1 \} \rightarrow X \rightarrow \text{TotalObj} \rightarrow X \rightarrow \mathbb{A}
```

We now turn to introducing some notations and syntax for dependent objects:
Given \( D : \text{DepObj Base} \) and \( X : \text{Obj}_1 \), we define \( X \rightarrow D \) as abbreviation for \( X \rightarrow \text{DepObj.TotalObj} \) \( D \), which is the type of \( \text{SG-morphism} \) from the \( \mathcal{F} \)-image of \( C \)-object \( X \) to the \( \text{TotalObj} \) of a dependent object \( D \). See \text{eLabel} in ADHG (Sect. 5.4) as an example.

Given additional \( f : X \rightarrow \mathcal{A} \), we write \( f \nearrow \mathcal{A} \) for \( \Sigma_{\text{U}} : X \rightarrow \text{TotalObj}^\approx \text{ind} \mathcal{U} \approx_2 f \), that is, for the type of pairs consisting of a morphism \( \mathcal{U} \) of type \( X \rightarrow D \) and a theorem \( \text{ind} \mathcal{U} \approx_2 f \). See \text{ELabel} in Sect. 5.4 as an example.

A “dependent functor” is a structure over an “index”. For example, \( \text{VEC} \) is assumed as a dependent functor over natural numbers for ADHG in Sect. 5.4. It has a function \( \text{DF} \) mapping all \( \text{SG} \)-objects to dependent objects indexed over \( \mathcal{A} \), see \text{NodeVec} in ADHG (Sect. 5.4) as an example.

We also introduce some notations and syntax for dependent functors:

Post-composition \( \overset{\_}{\rightarrow} \) generates an \( \text{SG-morphism} \) from the \( \text{SG-image} \) of \( C \)-object \( X \) to the dependent object of \( \text{DF} \mathcal{B} \) by composing an \( \text{SG-morphism} \) from \( X \) to the dependent object of \( \text{DF} \mathcal{A} \) with an \( \text{SG-morphism} \) \( g : \text{Mor}_2 \mathcal{A} \mathcal{B} \). See \text{prop-eln} in ADHGMor (Sect. 5.5) as an example.

\[
\begin{align*}
\_\overset{\_}{\rightarrow} \_ & : \{ X : \text{Obj}_1 \} \{ A B : \text{Obj}_2 \} (U : X \rightarrow \text{DF} A) \\rightarrow \{ g : \text{Mor}_2 \mathcal{A} \mathcal{B} \} \rightarrow X \rightarrow \text{DF} B \\
\_\overset{\_}{\rightarrow} \_ & = U \uparrow_2 \text{DF-mor} g
\end{align*}
\]

In summary, with this approach, we can obtain finite term graphs by instantiating the underlying category to the category of finite sets and functions. In fact, arbitrary (including infinite) term graphs can be obtained by instantiating the underlying category to the category of sets (respectively setoids) and functions. In this way, we will achieve a reusable definition of \emph{abstract} hypergraphs, whereas the original \( \text{DHG}_0 \) definition is not reusable for different purposes in that way.
5.4 Abstract Directed HyperGraph (ADHG)

In this section, we formalise term graphs as abstract directed hypergraphs, and also provide some variants.

First we fix some notation: Let $\text{Obj}_1$ denote the collection of objects in the parameter category $C$, and $\text{Obj}_2$ the collection of objects in the parameter semigroupoid $SG$, then $\text{Mor}_1 a b$ denotes the collection of morphisms of $C$ from $a : \text{Obj}_1$ to $b : \text{Obj}_1$. Similarly, $\text{Mor}_2 a b$ denoted the collection of morphisms of $SG$ from $a : \text{Obj}_2$ to $b : \text{Obj}_2$. The composition of $\text{Mor}_1$s is denoted as $\circ_1$, and similarly $\circ_2$ for $\text{Mor}_2$s.

We assume the setting of $\text{DepObjBase}$, an index object $N$, a dependent object $\text{Label}$, and a dependent functor $\text{VEC}$ are available; and opening most of this for re-export with some convenient renamings:

```haskell
module Data.ADHG3
  { l0_1 l0_2 lC_1 lC_2 lS_1 lS_2 : Level }
(Base : DepObjBase l0_1 l0_2 lC_1 lC_2 lS_1 lS_2)
(N : DepObjBase.Obj 2 Base)
(Label : DepObjBase.N)
(VEC : DepFuncto r Base N)
where
  open DepObjBase Base public
  open DepFunctor VEC public
    renaming
      (DF to Vec
      ; DF-obj to Vec-obj
      ; DF-mor to Vec-mor
      ; DF-cong to Vec-cong
      ; DFF to VecF
      ; DF-preIdentit y to Vec-preIdentit y)
  open DepObj
```

In our formalisation, an ADHG contains the following components:

```haskell
record ADHG (m n : Obj1) : Set (l0_1 lC_1 lC_2 lS_1 lS_2) where
  field Inner : Obj1
    Edge : Obj1
  Node = m \& Inner
  NodeVec : DepObj Base N
  NodeVec = Vec (F-obj Node)
  gln : Mor1 m Node
  gln = t
```

37
There are two parameters \( m \) and \( n \) for an ADHG: \( m \) represents the set of inputs or variables of the hypergraph, and \( n \) represents the set of outputs or results. They are objects of the parameter category \( C \).

- **Inner** stands for the set of all inner nodes (non-input nodes), and **Edge** stands for the set of all graph edges. **Inner** and **Edge** are objects of the parameter category \( C \). For technical reasons, we find it more convenient to have the non-input nodes separate from the input nodes. Otherwise, we would have to include an explicit injection from the input positions to the complete node set. **Inner** and **Edge** are presented as fields of the record.

**Node** is the set of all nodes, it is constructed as the coproduct of the input node set \( m \) and the inner nodes **Inner**. The underlying category should have coproduct already for expressing the construction, and this is one motivation why DepObjBase requires a choice of coproduct for the parameter category. **Node** is presented as a component derived from graph input \( m \) and the field **Inner**.

**NodeVec** is a dependent object used to form the type for vectors containing **Node** elements. The dependent parameter specifies the size of the vector. The “type” **NodeVec** will be used in **Eln** below. **NodeVec** is presented as a component derived from **Node** through **Vec** in VEC and the functor \( F \) in DepObjBase.

**gIn** is the coproduct injection of inputs into **Node**; it is also presented as a component derived from **Node**.

**gOut** specifies for each graph output node which **Node** it is taken from. It is a \( C \)-morphism from set of graph output nodes \( n \) to **Node**. Both variables and inner nodes can be output nodes of a hypergraph. **gOut** is presented as a field of the record.

**eOut** is a \( C \)-morphism specifying the output inner node of each edge. General directed hypergraphs differ from jungles in that **eOut** does not have to be a bijection: The possibility that **eOut** is not surjective allows for “undefined” inner nodes, and the possibility that **eOut** is not injective allows for “join nodes” in the sense of [Kahl et al.](2006). **eOut** is presented as a field of the record.
• eArity specifies the arity function for edges. It is of type \( \text{Edge} \rightarrow \mathbb{N} \) which is a SG-morphism from the F-image of C-object Edge to SG-object \( \mathbb{N} \). It is presented as a field of the record.

• ELabel is of type eArity \( \triangleright \) Label, which is a tuple: Assume we work in the context of an arity-indexed label type \( \text{Label}_0 : \mathbb{N} \rightarrow \text{Set} \), which corresponds to function symbols. “The first part” is ELabel listed below, and “The second part” is ind-eLabel listed below.

• Eln is of type eArity \( \triangleright \) NodeVec which is also a tuple. “The first part” is Eln listed below, and “The second part” is ind-Eln listed below.

• eLabel specifies the function for edge labels. It is of the type \( \text{Edge} \rightarrow \text{Label} \) as the SG-morphism from the F-image for C-object Edge to TotalObj of Label. It is the first component of tuple ELabel.

• ind-eLabel is a theorem stating that the “index component” of the function for edge labels is equivalent to the SG-morphism eArity. In another word, applying ind of the “dependent structure” \( \text{Label} : \text{DepObj Base} \mathbb{N} \) on ELabel (which is the function for edge labels) results in the morphism equivalent to eArity. It is the second component of tuple ELabel.

• Eln specifies the function for edge inputs. It is of type \( \text{Edge} \rightarrow \text{NodeVec} \) which is a SG-morphism from the F-image of Edge to NodeVec. It is the first component of tuple Eln.

• ind-Eln is a theorem stating that the “index component” of the function for edge inputs is equivalent to the morphism eArity. It is the second component of tuple ElLabel.

Jungles are what is called “term graphs” in Corradini and Gadducci (1999a). We formalise Jungles as a record containing a field \( \text{ADHG3} \) and also a field \( \text{eOutIso} \) stating that \( \text{eOut} \) is an isomorphism in the parameter category \( \mathbb{C} \). To facilitate the access to the inverse of \( \text{eOut} \), it also contains a component \( \text{eOut}^{-1} \) derived from field \( \text{eOutIso} \).

```
record Jungle (m n : Obj_1) : Set (ℓ_0 ∪ ℓ_1 ∪ ℓ_2 ∪ ℓ_s_1 ∪ ℓ_s_2) where
  field ADHG3 : ADHG m n
  open ADHG ADHG3
  field eOutIso : C.IsIso eOut
  eOut^{-1} : Mor_1 Inner Edge
  eOut^{-1} = eOutIso C.IsIso^{-1}
  open ADHG ADHG3 public
```

Function \( \text{wireADHG} \) generates an ADHG from a morphism in the parameter category \( \mathbb{C} \). Such ADHGs are called wiring graphs because they are used like “wiring”, connecting the graph input nodes and output nodes. In the implementation of \( \text{wireADHG} \) below:

• The parameter morphism \( F \) from the output object \( n \) to the input object \( m \) specifies for each output which input should be passed to it.
• Inner and Edge are implemented as the initial object ⊤ of the parameter category C representing the empty collection.
• gOut is implemented as $F_{\ast_1} \iota$ where $\iota$ serves as an adaptation from input nodes to Node.
• The remaining components are determined by the choice of the initial objects for Inner and Edge

```plaintext
wireADHG : {m n : Obj₁} → Mor₁ n m → ADHG m n
wireADHG {m} F = record
  {Inner  = ⊤,
   Edge   = ⊤,
   gOut   = F_{\ast_1} \iota,
   eOut   = ⊤,
   eArity = ⊤₂,
   ELabelf = ⊤₂ ↗ Labelf ⊤₂,
   EIn    = ⊤₂ ↗ (Vec(F-obj (m ⊕ ⊤₁))) ⊤₂
  }
```

Function wireJungle generates a Jungle from a morphism of the underlying category. Because Jungle is formalised as an ADHG whose eOut is isomorphic, it is straightforward to construct the component ADHG through wireADHG, which uses the initial objects for both Edge and Inner, so that eOut is an isomorphism due to initiality.

```plaintext
wireJungle : {m n : Obj₁} → Mor₁ n m → Jungle m n
wireJungle {m} F = record
  {ADHG₃ = wireADHG F,
   eOutIso = record
     {⁻¹ = ⊤,
      rightInverse = ⊤ᵣ,
      leftInverse = ⊤ᵣ}
  }
```

For a category $C$ with coproducts and initial object, the opposite category $C^{\text{op}}$ has products and a terminal object, which induces a gs-monoidal category denoted as $\boxtimes$-opGsMoncat. Then wireADHG / wireJungle is the morphism part of the gs-monoidal functor from $\boxtimes$-opGsMoncat to the gs-monoidal category of ADHG / Jungle.

The following are some special ADHGs implemented as wiring graphs.

• idADHG are ADHGs where inputs are passed directly to corresponding outputs;
• dupADHG will be used as $\nabla$ of the gs-monoidal category of ADHG;
• termADHG will be used as $!$ of the gs-monoidal category of ADHG;
• exchADHG will be used as swap of the symmetric monoidal category of ADHG.

Let $\text{Id} : \{A : \text{Obj}\} \to \text{Mor} A A$ denote the identity morphism in the underlying category $C$. 
\[ \text{idADHG} : \{ A : \text{Obj}_1 \} \rightarrow \text{ADHG} A A \]
\[ \text{idADHG} = \text{wireADHG} \text{Id} \]

\[ \_\_\_\text{join} : \{ A : \text{Obj} \} \rightarrow \text{Mor} (A \boxplus A) A \] is the morphism from \( A \boxplus A \) to \( A \) supplied in the underlying category \( C \). It is implemented as \( \text{Id} \bigtriangleup \text{Id} \), where \( \bigtriangleup \) is the coproduct construction because we assume \( C \) has coproduct in our setting \( \text{DepObjBase} \).

\[ \text{dupADHG} : \{ A : \text{Obj}_1 \} \rightarrow \text{ADHG} A (A \boxplus A) \]
\[ \text{dupADHG} \{ m \} = \text{wireADHG} (A \boxplus A) \]

\[ \_\_\_\_\text{swap} : \{ A: B : \text{Obj} \} \rightarrow \text{Mor} (A \boxplus B) (B \boxplus A) \] is the morphism from object \( A \boxplus B \) to \( B \boxplus A \) in the underlying category \( C \). It is implemented as \( \kappa \bigtriangleup \iota \).

\[ \text{exchADHG} : (A B : \text{Obj}_1) \rightarrow \text{ADHG} (A \boxplus B) (B \boxplus A) \]
\[ \text{exchADHG} m n = \text{wireADHG} \_\_\_\_\text{swap} \]

We show only the implementation of parallel composition \( \text{parADHG} \). Input, output, \text{Inner} and \text{Edge} are all constructed as direct sum objects; the auxiliary functions \( h_1 \) and \( h_2 \) map the node sets of the argument graphs into \( \text{Node}_3 \), which has four components.

\[ \text{parADHG} : \{ m_1 n_1 m_2 n_2 : \text{Obj}_1 \} \]
\[ \rightarrow \text{ADHG} m_1 n_1 \rightarrow \text{ADHG} m_2 n_2 \rightarrow \text{ADHG} (m_1 \boxplus m_2) (n_1 \boxplus n_2) \]
\[ \text{parADHG} \{ m_1 \} \{ n_1 \} \{ m_2 \} \{ n_2 \} G_1 G_2 = \text{let} \]
\[ \text{open ADHG} \]
\[ \text{Inner}_3 = \text{Inner} G_1 \boxplus \text{Inner} G_2 \]
\[ \text{Node}_3 = (m_1 \boxplus m_2) \boxplus \text{Inner}_3 \]
\[ h_1 : \text{Mor}_1 (\text{Node} G_1) \text{Node}_3 \]
\[ h_1 = \iota \boxplus \iota \]
\[ h_2 : \text{Mor}_1 (\text{Node} G_2) \text{Node}_3 \]
\[ h_2 = \kappa \boxplus \kappa \]
\[ \text{in record} \]
\[ \{ \text{Inner} = \text{Inner}_3 \} \]
\[ ; \text{Edge} = \text{Edge} G_1 \boxplus \text{Edge} G_2 \]
\[ ; \text{gOut} = \text{gOut} G_1 h_1 \bigtriangleup \text{gOut} G_2 h_2 \]
\[ ; \text{eOut} = \text{eOut} G_1 \boxplus \text{eOut} G_2 \]
\[ ; \text{eArity} = \text{eArity} G_1 \bigtriangleup \text{eArity} G_2 \]
\[ ; \text{ELab} = \bigtriangleup \nearrow \text{Label} (\text{ELabLabel} G_1) (\text{ELabLabel} G_2) \]
\[ ; \text{EIn} = \bigtriangleup \nearrow (\text{Vec} (\text{F-obj Node}_3)) (\text{EIn} G_1 \nearrow F\text{-mor} h_1) (\text{EIn} G_2 \nearrow F\text{-mor} h_2) \]
The sequential composition seqADHG is quite similar.

\[
\text{seqADHG} : \text{ADHG}_{m\ n} \to \text{ADHG}_{n\ k} \to \text{ADHG}_{m\ k}
\]

\section{Homomorphism and Category}

In this section we supply the homomorphism of ADHGs and also categories of ADHGs and homomorphisms.

\subsection{Homomorphism}

An ADHG homomorphism is a mapping between two ADHGs which preserves not only the internal structure of graphs, but also the graph input and output interfaces. ADHG homomorphisms are used in our formalisation of term graph rewriting, and also to define the isomorphisms that we will use for graph equivalence.

Note that this homomorphism concept is not the conventional concept of term graph homomorphism \cite{Corradini and Gadducci, 1999}, which we formalised as matching in Sect. 5.6 (matchings do not require the preservation of graph input/output interfaces).

Below we supply our formalisation for the homomorphism where the “structure preservation” property is expressed in the form of \emph{morphism equivalences} (point-free) in the underlying category. ADHGMor use the same context as ADHG. It consists of mappings for each of the two constituent objects \text{Inner} and \text{Edge} satisfying the preservation-of-structure conditions.

\begin{verbatim}
record ADHGMor {m n : Obj} (G₁ G₂ : ADHG m n) : Set (ℓc₁ ⊔ ℓc₂ ⊔ ℓs₂) where
  open ADHG
  field f-I : Mor₁ (Inner G₁) (Inner G₂)
  f-E : Mor₁ (Edge G₁) (Edge G₂)
  f-N : Mor₁ (Node G₁) (Node G₂)
  f-N = id ⊕ f-I

  field prop-gOut : gOut G₁ ≃₁ gOut G₂
  prop-eOut : eOut G₁ ≃₁ eOut G₂
  prop-eArity : f-E ↠ eArity G₂ ≃₂ eArity G₁
  prop-eLabel : f-E ↠ eLabel G₂ ≃₂ eLabel G₁
  prop-eIn : f-E ↠ eIn G₂ ≃₂ eIn G₁ ≈ F-mor f-N
\end{verbatim}
5.5.2 Homomorphism Category

The category for \( \text{ADHG}_{m \, n} \) and homomorphism is defined as follows:

- **Objects** are \( \text{ADHG}s \) and **arrows** between two objects are homomorphisms.
- **Two arrows are equivalent** if their two component mappings \( f-I \) and \( f-E \) are individually equivalent. Since \( f-I \) and \( f-E \) are morphisms from the underlying category \( C \), then the individual equivalence is the morphism equivalence in \( C \).
- **The composition** of homomorphisms is defined component-wise.
- **The identity homomorphism** is also defined component-wise.

We do not present the formalisation for semigroupoid, because it is quite similar to the formalisation for category in Sect. 3.1 except no identities are assumed. For each input-output interface, we obtain a category constructed by \( \text{ADHGHomCategory} \) of the following type:

\[
\text{ADHGHomCategory} : (m \, n : \text{Obj}_1) \rightarrow \text{Category} (\ell c_1 \cup \ell c_2 \cup \ell s_2) \cup c_2 (\text{ADHG}_{m \, n})
\]

Next we introduce the names and notations for isomorphisms of \( \text{ADHG}s \), because these are frequently used as equivalences of term graphs in the further development. The isomorphisms of \( \text{ADHG}s \) are defined as \( \text{Iso} \) (Sect. 3.1) of \( \text{ADHGHomCategory} \):

```kotlin
private module ADHGIso0 {m n : Obj1} where
    open Category.ISo (ADHGHomCategory m n) public
    Iso = Category.ISo (ADHGHomCategory m n)

open ADHGIso0

private module ADHGIso1 {m n : Obj1} = Category (ADHGHomCategory m n)
open ADHGIso1 using (IdIso; invIso; _Iso_) renaming (Iso to ADHGIso)
```

5.6 Matching and Category

In this section we supply the matching of \( \text{ADHG}s \) and the category of \( \text{ADHG} \) and matchings.

5.6.1 Matching

A **matching** from \( G_1 : \text{ADHG}_{m_1 \, n_1} \) to \( G_2 : \text{ADHG}_{m_2 \, n_2} \) is a mapping for internal nodes and edges. The difference between matching and homomorphism is that matching does not need to preserve the graph input-output interface. Therefore, we might
also use matchings as building blocks in the definition of homomorphisms. Note that our matchings are what Corradini and Gadducci (1999a) call “term graph homomorphisms”; they are the morphisms used to constitute the DPO in our formalisation of term graph rewriting. Below is the formalisation for matching:

\[
\text{record ADHGMatching} \{ m_1 n_1 : \text{Obj}_1 \} \ (G_1 : \text{ADHG} \ m_1 n_1) \\
\{ m_2 n_2 : \text{Obj}_1 \} \ (G_2 : \text{ADHG} \ m_2 n_2) : \text{Set} (\ell c_1 \cup \ell c_2 \cup \ell s_2)
\]

where

\begin{align*}
\text{open ADHG} \\
\text{field } f-N & : \text{Mor}_1 (\text{Node} \ G_1) (\text{Node} \ G_2) \\
\text{f-E} & : \text{Mor}_1 (\text{Edge} \ G_1) (\text{Edge} \ G_2) \\
\text{field } \text{prop-eOut} & : eOut \ G_1 \sim_1 \text{f-N} \ eOut \ G_2 \sim_1 \text{f-E} \\
\text{prop-eArity} & : \text{f-E} \rightsquigarrow eArity \ G_2 \approx eArity \ G_1 \\
\text{prop-eLabel} & : \text{f-E} \rightsquigarrow eLabel \ G_2 \approx eLabel \ G_1 \\
\text{prop-eIn} & : \text{f-E} \rightsquigarrow eIn \ G_2 \approx eIn \ G_1
\end{align*}

\[\uparrow \text{F-mor} \ text{f-N}\]

\section{5.6.2 Matching Category}

Next we formalise the category of matchings for the rewriting purposes in Chapter 9. The category for ADHGs and matchings is the category where:

- **Objects** are ADHGs with arbitrary input / output interface.
- **arrows** between two objects are matchings.
- **Two arrows are equivalent** if their two component mappings f-N and f-E are individually equivalent. Since f-N and f-E are morphisms from the underlying category C, then the individual equivalence is the morphism equivalence in C.
- **The composition of matchings** is defined component-wise.
- **The identity matching** is also defined component-wise.

The objects of the matching category can have arbitrary input and output interfaces, so we define a special datatype ADHG’ that allows us to treat ADHGs with arbitrary input and output interfaces as being of the same type. Then we define function \(\_\) converting a ADHG m n to a ADHG’.

\[
\text{record ADHG'} : \text{Set} (\ell c_1 \cup \ell c_2 \cup \ell s_1 \cup \ell s_2) \text{ where} \\
\text{field} \\
\text{input output} : \text{Obj}_1 \\
\text{graph} : \text{ADHG} \text{ input output} \\
\text{open ADHG'} \text{ public} \\
\_ : \{ m n : \text{Obj}_1 \} \ (G : \text{ADHG} \ m n) \rightarrow \text{ADHG'} \\
G' = \text{record} \{ \text{graph} = G \}
\]

\text{MatchingCategory} : \text{Category} (\ell c_1 \cup \ell c_2 \cup \ell s_2) \ell c_2 \text{ ADHG'}

It is easy to see that ADHG homomorphisms are matchings.
5.7 Term Graph Category

In this section, we supply the category for ADHG. In our work, the main purpose of this category is to be extended to the gs-monoidal category of ADHG which is useful in the semantics preservation of term graph rewriting in Chapter 9.

The category of ADHG is the category where:

- **Objects** are inherited from the underlying category, and **arrows** between two objects are ADHGs using the two objects respectively as input and output interface.
- Two parallel arrows are **equivalent** if they are isomorphic in the respective category of ADHG homomorphisms.
- The **composition** of ADHGs is the sequential composition of two ADHGs.
- The **identity morphism** is the identity ADHG, that is, \( \text{id}_{\text{ADHG}} \).

The formalisation of the term graph category is as following:

\[
\text{ADHGCategory} : \text{Category} (\ell_{o1} \cup \ell_{c1} \cup \ell_{c2} \cup \ell_{s1} \cup \ell_{s2}) (\ell_{c1} \cup \ell_{c2} \cup \ell_{s2}) \text{Obj}_1
\]

Furthermore, one interesting aspect of the category of ADHG homomorphisms and the category of ADHGs is that their connection could be modelled as a 2-category [Barr and Wells, 1990]:

- **0-cells** are the objects of \( \text{ADHGCategory} \)
- **1-cells** (morphisms) are morphisms of \( \text{ADHGCategory} \)
- **2-cells** (Cell) are the morphisms of \( \text{ADHGHomCategory} \)
- **Vertical composition** of 2-cells is the morphism composition in \( \text{ADHGHomCategory} \), see Fig. 5.1.

![Figure 5.1: The vertical composition of 2-cells](image)

- **Horizontal composition** of 1-cells is the sequential composition of ADHGs. Note that, this is a bicategory because the horizontal composition is not strictly associative since we defined the equivalence of the morphisms as ADHG isomorphism.
The horizontal composition of 2-cells takes a homomorphism $\alpha$ from graph $F$ to $G$ and a homomorphism $\beta$ from graph $H$ to $K$ to the homomorphism $\alpha \# \beta$ from the sequentially composed term graph $F \# H$ to $G \# K$, see Fig. 5.2.

This homomorphism composition is what we used in proving the isomorphisms between two ADHG compositions $F \# H$ and $G \# K$ as below, if $F$ is isomorphic to $G$ and $H$ is isomorphic to $K$.

seqADHGMor is the sequential composition of homomorphisms:

$$\text{seqADHGMor} : \{k \, m \, n : \text{Obj}\} \{F \, G : \text{ADHG} \, k \, m\} \{H \, K : \text{ADHG} \, m \, n\}$$
$$\rightarrow (\alpha : \text{ADHGMor} \, F \, G) (\beta : \text{ADHGMor} \, H \, K)$$
$$\rightarrow \text{ADHGMor} \, (F \# H) \, (G \# K)$$

Then seqADHGIso is built from seqADHGMor in a straight-forward manner.

$$\text{seqADHGIso} : \{k \, m \, n : \text{Obj}\} \{F \, G : \text{ADHG} \, k \, m\} \{H \, K : \text{ADHG} \, m \, n\}$$
$$\rightarrow \text{Iso} \, F \, G \rightarrow \text{Iso} \, H \, K \rightarrow \text{Iso} \, (F \# H) \, (G \# K)$$

### 5.8 GS-Monoidal Category

An algebraic approach to modelling acyclic term graphs with sequential and parallel composition has been proposed by Corradini and Gadducci (1999a) as gs-monoidal categories. They are not only a theoretical tool, but also a useful programming interface for term graph manipulations. In this section, we extend the category of ADHG to a gs-monoidal category.

We first extend the category of ADHG to a monoidal category ADHGMonCat by constructing the components and the proofs according to the specification of monoidal category in Sect. 3.2. The unit object $\otimes$ of the monoidal category is the initial object in the underlying category for ADHG, and the object part of $\otimes$ is the coproduct $\oplus$ of objects. On morphisms, the parallel composition of term graphs parADHG (see
Sect. 5.4) forms the disjoint union of term graphs, concatenating their input node sequences and output node sequences respectively into their coproducts.

Then we extend \( \text{ADHGMonCat} \) to the symmetric monoidal category \( \text{ADHGMonCatSym} \) by supplying component \( \text{swap} \) according to the specification in Sect. 3.3. \( \text{swap} \) is implemented as \( \text{exchADHG} \) which swaps the input node sequence \( A \sqcup B \) to \( B \sqcup A \) as output node sequence.

\[
\text{exchADHG} : (A, B : \text{Obj}) \to \text{ADHG}(A \sqcup B) (B \sqcup A)
\]

\[
\text{exchADHG} m n = \text{wireADHG}(\kappa \triangle \iota)
\]

Figure 5.3: \( \text{exchADHG} \)

Next we proceed to construct the gs-monoidal category of ADHG. We start from the construction of \( \text{ADHGMonCatG} \) which is a monoidal category permitting “garbage” according to the specification in Sect. 3.3. More precisely, the “g” of “gs-monoidal” stands for “garbage”: All edges of a term graph \( G : m \to n \) are garbage in the term graph \( G\triangleright!_n \). The “terminator” \(!\) is implemented as \( \text{termADHG} \) for which the output node sequence is empty, see Fig. 5.4.

\[
\text{termADHG} : \{A : \text{Obj}\} \to \text{ADHG} A \uplus
\]

\[
\text{termADHG} = \text{wireADHG} \uplus
\]

Figure 5.4: \( \text{termADHG} \)

Then \( \text{ADHGMonCatS} \) is a symmetric monoidal category equipped with “sharing” according to the specification in Sect. 3.3. More precisely, the “s” of “gs-monoidal” stands for “sharing”: Every input of \( \triangledown_k \triangleright(F \otimes G) \) is shared by \( F : k \to m \) and \( G : k \to n \). The duplicator \( \triangledown \) is implemented as \( \text{dupADHG} \) where the output node sequence is the concatenation of the input node sequence with itself, see Fig. 5.5 below.
dupADHG : \{A : \text{Obj}_1\} \rightarrow \text{ADHG} (A \oplus A)  
\text{dupADHG} \{m\} = \text{wireADHG} (\text{ld} \triangle \text{ld})

Finally, we obtain \text{ADHGMonCatGS} which is a \text{MonCatGS} specified in Sect. 3.3. That is, \text{ADHGMonCat}, \text{ADHGMonCatSym} and \text{ADHGMonCatGS} form a gs-monoidal category of ADHG which is named as \text{ADHG-GSMonCat}.

Note that ADHGs over a fixed edge label set form a gs-monoidal category, but not a Cartesian category, where in addition ! and \nabla are natural transformations, i.e., for all F : A \rightarrow B we have F \otimes !_B \approx !_A which means a graph F composed with “garbage collection” of graph output nodes of F is isomorphic to “garbage collection” of graph input nodes of F, and F \otimes \nabla_B \approx \nabla_A \otimes (F \otimes F) which means a graph F composed with duplication of graph output nodes of F is isomorphic to first duplicating the graph input nodes of F and then composed with a parallel composition of F. To see how these naturality conditions are violated by term graphs, Fig. 5.6 illustrates the five Jungles corresponding to the expressions:

\begin{align*}
\text{F:} 1 & \rightarrow 1  \\
!_1 &  \\
\text{F;}_! &  \\
\nabla_1 &  \\
\nabla_1; (F \otimes F) & 
\end{align*}

Figure 5.6: Illustration for the violation of naturality
Chapter 6

GS-Monoidal Expressions

In the decomposition of acyclic term graphs (Chapter 7), we will denote the result as expressions \cite{Corradini and Gadducci 1999a}. (In this chapter, we present GSME — GS-Monoidal Categorical Expression as the type of expressions which we choose to denote the result of the decomposition.) The datatype GSME implements the gs-monoidal specification, and it is intentionally for representing morphisms of the gs-monoidal category over the signature of edge labels, e.g., term graphs, because term graphs are morphisms in a gs-monoidal category (see Sect. 5.8). The transformation from GSMEs to morphisms of any gs-monoidal category is straight-forward.

We restrict our attention to the gs-monoidal categories where the object monoid is the additive monoid of the natural numbers. That is, in our gs-monoidal categories, the source and target objects of the morphisms are natural numbers. This is also the approach used by \cite{Corradini and Gadducci 1999a}. The two natural number parameters \(m\) and \(n\) for gs-monoidal expressions correspond to source and target of morphisms. For term graphs, \(m\) is the number of input nodes, and \(n\) is the number of outputs.

In Sect. 6.1 we introduce GSME as a data type. In Sect. 6.2 we sketch the gs-monoidal category of GSMEs which is called \textit{gs-monoidal theory} in \cite{Corradini and Gadducci 1999a}. In Sect. 6.3 we introduce GSME0 which is a variant of GSME for encoding the result of the decomposition of discrete term graphs.

6.1 GSME

For the purpose of representing morphisms of the gs-monoidal category as expressions, we define GSME (GS-Monoidal Categorical Expression) below according to the specification of gs-monoidal category in Sect. 3.3. GSME is defined as an Agda data type
supplying the constructors corresponding to the primitive morphisms of gs-monoidal category, e.g., the constructor \( \nabla_e : (n : N) \to \text{GSME } n (n + n) \) is corresponding to the transformation \( \nabla : \{A : \text{Obj}\} \to \text{Mor } (A \otimes A) \) required for every gs-monoidal category, and the constructor \( !_e : (n : N) \to \text{GSME } n 0 \) is corresponding to the transformation \( ! : \{A : \text{Obj}\} \to \text{Mor } (A \otimes 1) \).

Let \( \Sigma \) denote the signature for edge labels (\( \Sigma \) is an arity-indexed set of function symbols). Because we restrict the edge function of our hypergraphs (Sect. 2.1) to single-target functions, therefore the GSME for an edge is of type \( \text{GSME } n 1 \) where \( n \) denotes the number of edge input nodes, i.e., the GSME constructor \( \text{prim}_e \) produces a GSME \( n 1 \).

Below is the definition for GSME. Note that, in order to avoid confusion, the constructors have subscript “\( e \)”, which is different to the Agda code in [Zhao 2018].

```haskell
data GSME : (m n : N) \to \text{Set where}
  \text{prim}_e : \{n : N\} \to (f : \Sigma n) \to \text{GSME } n 1
  id_e : (n : N) \to \text{GSME } n n
  X_e : (m n : N) \to \text{GSME } (m + n) (n + m)
  \nabla_e : (n : N) \to \text{GSME } n (n + n)
  !_e : (n : N) \to \text{GSME } n 0
  _\otimes_e_ : \{m_1 n_1 m_2 n_2 : N\} \to \text{GSME } m_1 n_1 \to \text{GSME } m_2 n_2
    \to \text{GSME } (m_1 + m_2) (n_1 + n_2)
  _\#_e_ : \{m k n : N\} \to \text{GSME } m k \to \text{GSME } k n \to \text{GSME } m n
```

To construct the functor from the gs-monoidal category of term graphs to the gs-monoidal category chosen for semantics, we are also interested in the transformation from \( \Sigma\)-GSMEs to acyclic \( \Sigma\)-Jungles (ADHGs with the restriction that \( e\text{Out} \) is isomorphic). This function is the arrow mapping of the functor from the gs-monoidal category of GSMEs (Sect. 6.2) to the gs-monoidal category of term graphs. Note that, when considering propositional equality of GSMEs, this function is not injective which has been reflected in our formalisation of jungle decomposition (see Sect. 8.3). For example, in Fig. 6.1, to the left is the jungle for the GSME \( (\text{prim}_e f \otimes_e \text{prim}_e g) \otimes_e \text{prim}_e h \) and in the middle is the jungle for the GSME \( (\text{prim}_e f \otimes_e \text{prim}_e h) \otimes_e (\text{prim}_e g \otimes_e \text{id}_e) \).

Note that, only for this example we attach indices to the inner nodes to distinguish their identities. The two jungles are isomorphic; we draw all jungles in this isomorphism class in the way of the right drawing in Fig. 6.1. These two GSMEs are equivalent morphisms in the gs-monoidal category of GSMEs sketched in Sect. 6.2.

Also note that the transformation function from \( \Sigma\)-GSMEs to the gs-monoidal category of all acyclic \( \Sigma\)-ADHGs (not only jungles) is not surjective. Fig. 6.2 is an example.
In this section, we sketch the gs-monoidal category of GSMEs according to the specification in Chapter 3. In the category of GSMEs, objects are natural numbers and morphisms are GSMEs. The category components \( \text{Id} \) and \( \text{Id} \) are the GSME constructor \( \text{Id} \) and \( \text{Id} \), because we intentionally define GSME in a way to be arrows of the gs-monoidal category. Similarly, for the constructors of the gs-monoidal category, we supply the constructors of GSME for them, i.e., \( \text{X}_e \), \( \nabla_e \), \( !_e \) and \( \otimes_e \).

Then we have to supply the equivalence between two elements of GSME m n to define

\[ \text{Figure 6.1: The isomorphic jungles} \]

\[ \text{Figure 6.2: An ADHG not expressible as GSME} \]

### 6.2 GS-Monoidal Category of GSMEs

In this section, we sketch the gs-monoidal category of GSMEs according to the specification in Chapter 3. In the category of GSMEs, objects are natural numbers and morphisms are GSMEs. The category components \( \text{Id} \) and \( \text{Id} \) are the GSME constructor \( \text{Id} \) and \( \text{Id} \), because we intentionally define GSME in a way to be arrows of the gs-monoidal category. Similarly, for the constructors of the gs-monoidal category, we supply the constructors of GSME for them, i.e., \( \text{X}_e \), \( \nabla_e \), \( !_e \) and \( \otimes_e \).

Then we have to supply the equivalence between two elements of GSME m n to define
the hom-setoids. The equivalence of two GSMEs is defined as the following data type 
\[ _{-e\approx-} \] :

\[
\begin{align*}
\text{data } _{-e\approx-} : & \{ m n : N \} \to \text{GSME } m n \to \text{GSME } m n \to \text{Set}_1 \\
\text{eq-reflexivity} : & \{ m n : N \} \to E : \text{GSME } m n \to E \approx E \\
\text{eq-symmetry} : & \{ m n : N \} \to E_1 : \text{GSME } m n \to E_2 : \text{GSME } m n \to E_1 \approx E_2 \to E_2 \approx E_1 \\
\text{eq-transitivity} : & \{ m n : N \} \\
& \to E_1 : \text{GSME } m n \to E_2 : \text{GSME } m n \to E_3 : \text{GSME } m n \\
& \to E_1 \approx E_2 \to E_2 \approx E_3 \to E_1 \approx E_3 \\
\text{eq-\_\_\_\_\_ cong} : & \{ m n k : N \} \to \{ E_1 E_2 : \text{GSME } m n \} \{ F_1 F_2 : \text{GSME } n k \} \\
& \to E_1 \approx E_2 \to F_1 \approx F_2 \to (E_1 \approx F_1) \approx (E_2 \approx F_2) \\
\text{eq-\_\_\_\_\_ assoc} : & \{ m n k d : N \} \to \{ f : \text{GSME } m n \} \{ g : \text{GSME } n k \} \{ h : \text{GSME } k d \} \\
& \to ((f \approx g) \approx h) \approx (f \approx (g \approx h)) \\
\text{eq-leftId} : & \{ m n : N \} \to \{ E : \text{GSME } m n \} \to (\text{Id}_e \{ m \} \approx E) \approx E \\
\text{eq-rightId} : & \{ m n : N \} \to \{ E : \text{GSME } m n \} \to (E \approx \text{Id}_e \{ m \}) \approx E \\
\end{align*}
\]

Because GSME is defined intentionally to supply the arrows of a gs-monoidal category, therefore according to the specification of gs-monoidal category, we define it to have all the components of the gs-monoidal category, and also define it to satisfy all the laws of the gs-monoidal category. I.e., we define the composition of GSME to satisfy the associativity and congruence laws, i.e., eq-\_\_\_\_\_ cong and eq-\_\_\_\_\_ assoc, as well as the identity law as: eq-leftId and eq-rightId.

All of the coherence axioms (see sections 2.2.2 and 2.2.3) are proved through the (omitted part of the) definition of the equivalence _-e\approx-_, similar to the associativity and congruence of composition.

In a conventional mathematical approach as in [Corradini and Gadducci (1999a)] or in an extensional type theory such as NQThm [Boyer et al. (1995)], one would have defined the morphism to be equivalence classes of GSMEs, with respect to the equivalence relation _-\_\_\_\_\_, induced by the gs-monoidal theory.

We define a mapping GSMEsem from GSMEs to morphisms of a gs-monoidal category. GSMEsem is contained in module Sem inside of module Data.GSME.Sem2. The modules require parameters: \( \Sigma : (\text{arity} : N) \to \text{Set} \) as the signature, \( C' : \text{GSMonoidalCategory } j k \text{ Obj} \) as the gs-monoidal category where GSMEs are mapped to the morphisms, and \( \text{obj}'' : N \to \text{Obj} \) as the mapping from natural numbers to objects of the gs-monoidal category.

\[
\begin{align*}
\text{module } \text{Data.GSME.Sem2} \ (\Sigma : (\text{arity} : N) \to \text{Set}) \\
\{ i j k : \text{Level} \} \{ \text{Obj} : \text{Set} i \} \\
(C' : \text{GSMonoidalCategory } j k \text{ Obj}) \\
\text{where} \\
\text{module } \text{Sem} (\text{obj}'' : N \to \text{Obj})
\end{align*}
\]
where

\[ \text{GSMEsem} : \{m, n : \mathbb{N}\} \rightarrow \text{GSME} m n \rightarrow \text{Mor} (\text{obj}'' m) (\text{obj}'' n) \]

The actual definition of $\text{GSMEsem}$ is straightforward. Note that, $\text{GSMEsem}$ is the mapping for morphisms in the gs-monoidal functor from the gs-monoidal category of GSMEs to the gs-monoidal category $C'$.

### 6.3 GSME0

GSME0 is a variant of GSME. It has all the constructors of GSME except that it does not have `prim` which accommodates function symbols from the signature (function symbol set).

It is created for the convenience of the proofs for the correctness of discrete term graph decomposition. For more details, see the discussions in Sect. 8.4.

```haskell
data GSME0 : (m n : \mathbb{N}) → Set where
  id_{e0} : (n : \mathbb{N}) → GSME0 n n
  X_{e0} : (m n : \mathbb{N}) → GSME0 (m + n) (n + m)
  \nabla_{e0} : (n : \mathbb{N}) → GSME0 n (n + n)
  !_e0_ : (n : \mathbb{N}) → GSME0 n 0
  _⊗_{e0}_ : \{m_1 n_1 m_2 n_2 : \mathbb{N}\} → GSME0 m_1 n_1 → GSME0 m_2 n_2 → GSME0 (m_1 + m_2) (n_1 + n_2)
  _♯_{e0}_ : \{m k n : \mathbb{N}\} → GSME0 m k → GSME0 k n → GSME0 m n
```

In case of type GSME for the expression is required, we can use the natural embedding $\text{GSME0-GSME}$:

$$\text{GSME0-GSME} : \{m, n : \mathbb{N}\} \rightarrow \text{GSME0} m n \rightarrow \text{GSME} \Sigma m n$$

When $\Sigma$ is empty, this is obviously an isomorphism.
Chapter 7

Decomposition of Acyclic Term Graph

Corradini and Gadducci (1999a) showed us a road to the semantics of term graphs. That is, the semantics of term graphs can be obtained through the functor from the gs-monoidal category of term graphs to the gs-monoidal category chosen as semantics. We will see in Sect. 7.1 that the decomposition of term graphs, i.e., using an expression to present the compositional structure of the given acyclic term graph, is an essential part of this functor.

In this chapter, we will discuss how to decompose acyclic term graphs and encode the result as GSME, the type of expressions we chose. The correctness of the decomposition will be shown in the next chapter.

The decomposition algorithm we constructed targets term graphs of type VDHG, which is the instantiation of ADHG by supplying vecCategory as the underlying category (see Chapter 5). In this instantiation, since the objects are natural numbers, therefore the graph interfaces are natural numbers which could be understood as representing the number of CPU registers in the spirit of the code generation system Coconut by Anand and Kahl (2009); this is also the approach to the formalisation of term graph in Corradini and Gadducci (1999a). From now on, we also call VDHG “term graph” in this chapter.

The decomposition algorithm targets VDHG, due to limitations of the current setting DepObjBase (see Sect. 5.2). I.e., ADHG does not contain sufficient interface to work on for decomposition, the collection of edges is an object in the underlying category with no further restriction. Therefore we have no means to access the individual edges, nor the inner nodes.

The restriction to the instantiation VDHG, at first appears to constitute a loss of generality. However, as long as the underlying category is equivalent to some category of finite sets, our VDHG-based decomposition can be made to work on other ADHG instances via an appropriate forgetful functor.
The full formalisation is available on-line (Zhao, 2018).

7.1 Purpose

We are interested in obtaining the semantics of term graphs. Because the $gs$-monoidal category of term graphs (Sect. 5.7) is freely generated from the signature (Corradini and Gadducci, 1999a), it is an initial object in the category of all $gs$-monoidal categories and functors. Therefore there always exists a uniquely-determined functor from the $gs$-monoidal category of term graphs to any $gs$-monoidal category. Because the $gs$-monoidal category of expressions (Sect. 6.2) is another initial object in the category of all $gs$-monoidal categories and functors, there always exists a uniquely-determined functor from the $gs$-monoidal category of expressions to the $gs$-monoidal category we choose for semantics. E.g., if we choose some set $\mathcal{V}$ as the set of values of term graph nodes; a term graph with $m$ inputs and $n$ outputs then has a function of type $\mathcal{V}^m \to \mathcal{V}^n$ as semantics.

The semantics of term graphs is obtained by a functor from the $gs$-monoidal category of term graphs to a $gs$-monoidal category chosen as semantics. The functor is constructed as the composition of two functors: The “decomposition” functor from the $gs$-monoidal category of term graphs to the $gs$-monoidal category of expressions, and the functor constructed on $GSME_{sem}$ (see Sect. 6.2) from the $gs$-monoidal category of expressions to the $gs$-monoidal category chosen as semantics.

7.2 Function Type and Working Mechanism

The algorithm for the decomposition is formalised as $\text{decomposeVDHG}_1$:

$$
\text{decomposeVDHG}_1 : \{m, n : \mathbb{N}\}
\rightarrow (G : \text{VDHG}_{3, m, n})
\rightarrow C.\text{IsIso} (\text{eOut} G)
\rightarrow \text{IsProgressivelyFinite} (\text{predRelD} G)
\rightarrow \text{GSME} \Sigma m, n
$$

This states that: For any given graph $G$, if $\text{eOut} G$ is an isomorphism in the parameter category (that is, $G$ is a jungle, or a term graph in the sense of Corradini and Gadducci (1999a)), and also if the “node adjacency relation” $D$ (Kahl, 1996) of $G$ is progressively finite (that is, $G$ is acyclic in the sense of Schmidt and Ströhlein (1993)), a $\text{GSME}$ (intended to be the encoding for the decomposition of $G$) is produced by $\text{decomposeVDHG}_1$. 

55
The decomposition algorithm inducts on the number of edges. The brief idea of the implementation is illustrated as below:

- For an acyclic non-discrete term graph $G$ (see Fig. 7.1) which has at least one edge, we pick an edge (denoted as chosenEdge) from the term graph $G$, and consider $G$ as the composed graph $(G_1 \mathbin{\hat{\circ}} (\text{id}_{ADHG} \otimes G_2)) \mathbin{\hat{\circ}} G'$ where $G_1$ duplicates $G$ graph inputs, $G_2$ contains only chosenEdge, and $G'$ is the remainder of $G$ (see Fig. 7.2). Let GSME $E_1$ denote $G_1$, $E_2$ denote $G_2$, and $E'$ denote $G'$. The result in GSME for the algorithm is $(E_1 \mathbin{\hat{\circ}}_e (\text{id}_{m} \otimes e E_2)) \mathbin{\hat{\circ}}_e E'$, where $\text{id}_{m}$ is also a GSME, $\mathbin{\hat{\circ}}_e$ and $\otimes_e$ are GSME compositions. We will talk about the details in Sect. 7.3.2.

- For a discrete term graph (also called “wiring graph” in Sect. 5.4), the algorithm applies function WireTG-GSME0 (see Sect. 7.3.1) to encode $\text{gOut } G$ as a GSME in order to encode the discrete term graph. We will talk about the details in Sect. 7.3.1.

### 7.3 Implementation

In this section, we supply the details for the decomposition of acyclic term graphs: In Sect. 7.3.1 we will talk about the decomposition of the discrete term graphs, and Sect. 7.3.2 for the non-discrete term graphs.
Figure 7.2: A decomposition for G
7.3.1 Discrete Term Graphs

The case for discrete term graphs in the decomposition algorithm decomposeVDHG₁ (see Sect. 7.2) is implemented as:
\[
\text{GSME₀-GSME } \Sigma \left( \text{WireTG-GSME₀ } (g\text{Out } G) \right)
\]

We implement by applying function composition GSME₀-GSME \( \Sigma \) (WireTG-GSME₀) to encode \( g\text{Out } G \) as a GSME₀ (see Sect. 6.3) and then convert it to a GSME (see Sect. 6.1), instead of encoding \( g\text{Out } G \) as a GSME directly. This is because it reduces the complexity of the proofs for the correctness of discrete term graph decomposition quite a lot. For more details, see the discussion in Sect. 8.4.

For a discrete term graph (also called “wiring graph” in Sect. 5.4), i.e., there is no edge or inner node in the graph, \( g\text{Out} \) is the only component containing relevant information about the graph functionality (see Sect. 5.4) which specifies for each output which input should be passed to it. That is, with \( g\text{Out} \) we are able to reproduce the discrete term graph through the function application of wireADHG (see Sect. 5.4) on \( g\text{Out} \). In the current setting, \( g\text{Out} \) of the discrete graph \( V\text{DHG} \) is a \( \text{vec} (\text{Fin } n ) m \), where \( m \) and \( n \) are natural numbers. The algorithm WireTG-GSME₀ (see Sect. 7.3.1) can be understood as resulting in a GSME₀ encoding for the transformation steps from the vector allFin \( m \) to the given vector \( v : \text{vec} (\text{Fin } m ) n \). That is, since \( g\text{Out} \) of the discrete \( V\text{DHG} \) \( G \) is a \( \text{vec} (\text{Fin } m ) n \), we apply WireTG-GSME₀ to encode \( g\text{Out} \) \( G \) as a GSME₀ in order to encode the discrete term graph.

In the remainder of this subsection, we present the sub-algorithms for WireTG-GSME₀ which encodes the generalisation of a finite number vector as a GSME₀.

From the Sorted Vector to the Given Vector

First, we present GSME₀-unsortedVfromSortedV which is a sorting algorithm that returns also the performed permutation encoded as a GSME₀ \( n \ n \). In this level of the approach, the sorting procedure involves only swapping of the elements but no duplication and elimination. The procedure is encoded as a GSME₀ \( n \ n \), where the first \( n \) is understood as the size of procedure inputs and the second \( n \) as the size of outputs. We will see in Sect. 8.3 the GSME₀ generated by this sub-algorithm is mapped to a wiring graph which outputs the given vector from the input sorted vector.

\[
\text{GSME₀-unsortedVfromSortedV } : \{ m \ n : \mathbb{N} \} \\
\quad \rightarrow \text{vec} (\text{Fin } m ) n \\
\quad \rightarrow \Sigma \ v : \text{vec} (\text{Fin } m ) n \\
\quad \quad \bullet \text{isSorted'} v \times \text{GSME₀ } n \ n
\]

Note that \( \Sigma \ x : \mathbb{A} \bullet \mathbb{B} \) is a variant of the syntax supplied in the Agda standard library for dependent products: \( \mathbb{B} \) is a dependent type on \( x : \mathbb{A} \), and the type \( \Sigma \ x : \mathbb{A} \bullet \mathbb{B} \)
contains pairs of shape (x, y) where x : A and y : B x. The operator \( x \times \) is the nondependent variant of this, the usual Cartesian product type constructor: \( I \times J \) denotes the type of pairs with constituents taken from I and J. GSME0-unsortedVfromSortedV uses insertion sort to sort a given vector. It starts from the last element to get a partially sorted vector, and then inserts the element next to the last one into the partially sorted vector, and so on. Therefore the procedure to obtain the given vector (no matter whether it is unsorted) from the sorted vector is reverse to the insertion sorting. The procedure is encoded in the result in GSME0. Denote \( x \rhd xs \) as the result of inserting \( x \) into a sorted vector \( xs \).

- If the given vector is \( [ \ ] \), then it results in \( id_{e0} 0 \).
- If the given vector is \( x :: [ \ ] \) (where \( :: \) is the vector type constructor, i.e., \( x :: xs \) denotes the vector generated from pre-pending an element \( x \) to a vector \( xs \)), then it results in \( id_{e0} 1 \).
- If the given vector is \( x :: y :: xs \), then it results in \( e_1 \%id_{e0} 1 \otimes e_0 e_2 \), where \( e_1 \) denotes the transformation from the sorted vector \( w = x \rhd (y \rhd xs) \) to \( x :: (y \rhd xs) \) which is the result of sub-algorithm insertElem (see below) on \( (x , (y \rhd xs)) \), and \( id_{e0} 1 \otimes e_0 e_2 \) denotes the transformation from \( x :: (y \rhd xs) \) to the given vector \( x :: y :: xs \) where \( e_2 \) is the result of insertElem on \( (y , xs) \).

Sub-algorithm insertElem inserts an element into a partially sorted vector to obtain a sorted vector. It will keep comparing the “to be inserted element” with vector elements in order. The “to be inserted element” element will be placed before the first element not smaller than it. Note that \( \equiv \) is the “propositional equivalence” in Agda.

\[
\text{insertElem} : \{ m n : \mathbb{N} \} \\
\hspace{1cm} \rightarrow (x : \text{Fin} m) \\
\hspace{1cm} \rightarrow (v : \text{Vec} (\text{Fin} m) n) \\
\hspace{1cm} \rightarrow \text{isSorted'} v \\
\hspace{1cm} \rightarrow \Sigma w : \text{Vec} (\text{Fin} m) (\text{suc} n) \bullet \\
\hspace{2cm} \text{isSorted'} w \\
\hspace{2cm} \times \text{head } w \equiv \text{minNewElem-VecHead} x v \\
\hspace{2cm} \times \text{GSME0} (\text{suc} n) (\text{suc} n)
\]

From allFin m to a Given Sorted Vector

Next we present GSME0-sortedVfromPrimitiveV which encodes the procedure for obtaining the given sorted vector \( v \) of type \( \text{vec} (\text{Fin} m) n \) from the vector allFin m (which is \( [0, 1, 2, ..., m-1] \)). In this level of the approach, this sub-algorithm only duplicates and eliminates the elements of allFin m (which is a size-m vector) to obtain the given vector (which is a size-n vector). The procedure is encoded as a GSME0 m n, where \( m \) is understood as the size of procedure inputs and \( n \) as the size of outputs. We will see
in Sect. 8.3 that the GSME0 generated by this sub-algorithm is mapped to a wiring graph which outputs the given vector $v$ from the input vector allFin $m$.

$$\text{GSME0-sortedV from PrimitiveV} : \{ n : \mathbb{N} \}
\to (v : \text{Vec (Fin m) n})
\to \text{isSorted'}v
\to \text{GSME0 m n}$$

Note that the result is of type GSME0 which is a variant of GSME, and recall that we have conversion function from GSME0 to GSME available. Below are the patterns we distinguish for an input of type vec ($\text{Fin m}$) $n$, because they are easy to apply in an inductive setting:

- If the given $v$ is $[ ]$, then the result in GSME0 is $!_{e_0}m$, because $!_{e_0}m$ denotes eliminating all of the elements in vector $0, 1, 2,..., m-1$;
- If the given vector is zero :: [ ], then the result in GSME0 is id$_{e_0}1 \otimes _{e_0} !_{e_0}m$, because it denotes the copying of the first vector element zero in $0, 1, 2,..., m-1$ and also eliminating all the elements from vector $0, 1, 2,..., m-1$;
- If the given vector is zero :: zero :: xs, then the result in GSME0 is $(\nabla_{e_0}1 \otimes _{e_0} \text{id}_{e_0}m) \overset{!}{_{e_0}} (\text{id}_{e_0}1 \otimes _{e_0} e'$), where $\nabla_{e_0}1 \otimes _{e_0} \text{id}_{e_0}m$ denotes duplicating the first vector element zero in $0, 1, 2,..., m-1$ and copying all the rest in the vector to produce $0, 0, 1, 2,..., m-1$ from $0, 1, 2,..., m-1$, and then $\text{id}_{e_0}1 \otimes _{e_0} e'$ denotes copying the first element from $0, 0, 1, 2,..., m-1$ and also producing zero :: xs from $0, 1, 2,..., m-1$ (by $e'$ because it is the output of the recursive call on zero :: xs since a sub-vector of a sorted vector is also sorted) to produce zero :: zero :: xs;
- If the given $v$ is zero :: suc $i$ :: xs, then the result in GSME0 is $\text{id}_{e_0}1 \otimes _{e_0} e'$, where $\text{id}_{e_0}1$ copies zero from vector $0, 1, 2,..., m-1$. Because $v$ is sorted, then the sub-vector suc $i$ :: xs contains no zero. Then $e'$ is the result of the recursive call for the degraded form (each vector element is subtracted by 1) of vector suc $i$ :: xs from $0, 1, 2,..., m-2$, where $e'$ is the same as the GSME0 to obtain suc $i$ :: xs from $1, 2,..., m-1$;
- If the given vector is suc $i$ :: xs, then the result in GSME0 is $!_{e_0}1 \otimes _{e_0} e'$, because $!1$ will eliminate 0 from vector $0, 1, 2,..., m-1$ and then $e'$ is the GSME0 for the degraded form of vector suc $i$ :: xs from $0, 1, 2,..., m-2$ (which is same as suc $i$ :: xs from $1, 2,..., m-1$).
From allFin m to the Given Vector

Combining them together we present WireTG-GSME0 which produces a gs-monoidal expression e encoding the transformation steps from allFin m to the given vector v. The result e is a composition $e_1 \circ e_0 \circ e_2$ where $e_1$ encodes the transformation (through copying, duplicating and eliminating elements) from allFin m to vector $v'$ which is the sorted form of v, and $e_2$ denotes the transformation (through copying and switching of the elements) from $v'$ to v.

$$\text{WireTG-GSME0} : \{m, n : \mathbb{N}\} \rightarrow \text{Vec} (\text{Fin} m) n \rightarrow \text{GSME}0 m n$$

$$\text{WireTG-GSME0} v = \text{let } (v', \text{isSorted}v', e_2) = \text{GSME0-unsortedVfromSortedV} v \text{ in } e_1 = \text{GSME0-sortedVfromPrimitiveV} v' \text{ isSorted}v'$$

7.3.2 Non-discrete A cyclic Term Graphs

In this subsection, we give more details of generating a GSME for a non-discrete term graph.

As mentioned in the illustration of the algorithm (Sect. 7.2), for an acyclic non-discrete term graph $G$ (see Fig. 7.1) which has at least one edge, first we pick an edge (denoted as chosenEdge) from the “very top” of $G$, and then construct the component graph $G'$ (see Fig. 7.2) which is $G$ with chosenEdge removed, and also construct a wire graph $G_2$ which only contains the edge chosenEdge. Next we construct a component graph $G_1$ which not only forwards the graph input nodes to graph output nodes, but also duplicates the graph input nodes for the edge input nodes of chosenEdge. Finally, we encode each component graph as a GSME: Let GSME $E_1$ denote $G_1$, $E_2$ denote $G_2$, and $E'$ denote the GSME for the recursive call of the algorithm on the component graph $G'$. Then the compositional structure of $(G_1 ; (\text{idADHG} \otimes G_2)) ; G'$ is presented as $(E_1 \circ_e (\text{id} m \otimes_e E_2)) \circ_e E'$ where $\text{id} m$ is also a GSME; $\circ_e$ and $\otimes_e$ are GSME compositions. Note that, the graph composition $(G_1 ; (\text{idADHG} \otimes G_2)) ; G'$ results in a single instance ADHG whose record fields are constructed by the applications of graph composition over the component graphs. I.e., there is no information of the compositional structure in the data of $(G_1 ; (\text{idADHG} \otimes G_2)) ; G'$.

Therefore, the case of non-discrete acyclic term graphs for decomposeVDHG$_1$ (Sect. 7.2) is implemented as

$$(E_1 \circ_e (\text{id} m \otimes_e E_2)) \circ_e \text{decomposeVDHG}_1 G' \equiv \text{refl} \equiv \text{refl} C. \text{Id-isIso} \text{predRelDG'}-\text{progFin}$$

where $E_1$ is obtained through using WireTG-GSME0 since $G_1$ is a discrete graph, $E_2$ is obtained through constructor prim$_c$ (Sect. 6.1) because $G_2$ only contains chosenEdge,
and the part \( \text{decomposeVDHG}_{1} \) \( G' \) \( k = -\text{refl} = -\text{refl} \) \( \mathcal{C}.\text{Id-isIso} \) \( \text{predRelDG}' \)-\( \text{progFin} \) is the recursive call on of the algorithm on \( G' \).

From the implementation, we see the construction of \( G' \) is an essential part of the decomposition algorithm. It starts from generating the candidate set.

**Generating the Candidate Set**

As mentioned in the illustration of the algorithm, the algorithm picks an edge from \( G \) which is called \( \text{chosenEdge} \). A candidate is an edge of the term graph where:

- If the edge is a constant, then it is a good candidate.
- If the edge has only graph input nodes as its input, then it is a good candidate.
- If the edge has any other edge output inner nodes as its input, then it is NOT a good candidate.

For that purpose, it generates a set of edges where each one is qualified to be a \( \text{chosenEdge} \). We call this set “candidate set”.

We obtain the candidates through the set difference between the set of all edge and the set of the edges which use any edge output inner nodes as inputs. In order to obtain the minuend of this set difference, we use the “node adjacency relation” (denoted as \( D \) by Kahl (1996)) of the term graph. This relation relates each inner node \( i \), if \( i \) is the unique output of an edge \( e \), to all the input nodes of \( e \). It can be constructed from the corresponding relations of the edge input and edge output functions. However, in the setting of VDHG, the functions of term graphs are represented by the morphisms in the underlying category which is a vector of finite numbers. Therefore, we have to transform the vectors of finite numbers into relations. In RATH-Agda library, there is the relation-algebraic interface (a Kleene category plus some additional materials) which has relations represented as set-valued finite maps (partial functions). Through this interface, since \( \text{vecCategory} \) has been proven equivalent to the subcategory of mappings of the concrete relation algebra implemented there, a morphism in \( \text{vecCategory} \) can be transformed into a set-valued finite map representing a relation between \( \text{Fin}a \) and \( \text{Fin}b \) where \( a, b : \mathbb{N} \).

Note that in the application of the algorithm in Sect. 9.2, the targeted term graphs are jungles where \( e\text{Out} \) is an isomorphism. With this constraint on \( e\text{Out} \), it is easy to obtain the corresponding edge from the edge output inner node, such as in the calculation of the candidate set through relation \( D \). Besides, this constraint also ensures that the candidate set is non-empty. We also need an assertion for that the term graph is acyclic, which for finite graphs is equivalent to “the relation \( D \) is progressively finite” [Schmidt and Ströhlein 1993].
Constructing $G'$

According to the specification of ADHG in Sect. 5.4 with assuming $G : \text{VDHG}_3 \equiv m n$ and that Inner $G$ is suc(k) (because it is non-discrete), we construct $G'$ as below:

$$G' : \text{VDHG}_3 (m \equiv 1) n$$

$$G' = \text{record}$$

\[
\begin{align*}
\{ & \text{Inner} = k \\
& \text{Edge} = k \\
& \text{gOut} = \text{gOut} G \phi \psi \\
& \text{eOut} = \text{ld}_1 \\
& \text{eArity} = (\phi \psi \text{eOut}^{-1}) \rightarrow \text{eArity} G \\
& \text{ELabel} = (\phi \psi \text{eOut}^{-1}) \rightarrow (\text{el} G \nearrow Mor \psi) \\
& \quad \quad \text{DepObjInd-Label} G (\phi \psi \text{eOut}^{-1}) \\
& \text{El} = (\phi \psi \text{eOut}^{-1}) \rightarrow (\text{el} G \nearrow \nearrow Mor \psi) \\
& \quad \quad \text{DepObjInd-El} G (\phi \psi \text{eOut}^{-1}) \\
\} 
\end{align*}
\]

- The graph input of $G'$ is $m \equiv 1$, because besides the graph input of $G$, it also has one new graph input which connects to the output of the graph for chosenEdge. That new graph input node is prepared for any edges of $G$ which used the output of the chosenEdge as inputs, or for being one of the graph outputs of $G'$ if the edge output of chosenEdge in $G$ was also a graph output node.
- The graph output interface is $n$ which is the same as $G$, because the decomposition does not change the output interface of the graph to be decomposed.
- We construct Inner $G'$ as Inner $G$ reduced by 1, since the output inner node of chosenEdge is taken out with chosenEdge from Inner $G$.
- We construct Edge $G'$ as Edge $G$ reduced by 1, because Edge $G$ has one edge (chosenEdge) taken out.

By definition (Sect. 5.4), Node $G'$ is the coproduct of the graph input nodes and the inner nodes of $G'$. Although Node $G$ and Node $G'$ have the same size, their structures are different: Let $\iota_G$ and $\kappa_G$ denote the injections for coproduct Node $G$, as well as $\iota_{G'}$ and $\kappa_{G'}$ the injections for Node $G'$. The finite set of inner nodes of $G'$ is the finite set of inner nodes of $G$ having one inner node (the edge output inner node of chosenEdge) taken out. The finite set of $G'$ graph input nodes is the finite set of $G$ graph input nodes having the new input node appended to it. Below we use $\psi : \text{Mor}_1 (\text{Node} G) (\text{Node} G')$ to reflect this change of structure. See Fig. 7.3 for an illustration for the types of these morphisms; the diagram there also commutates. We will give the details for how $\psi$ is constructed in “Constructing $\phi$ and $\psi$” below.

- gOut $G'$ is a morphism from the graph output object $n$ to Node $G'$. Since gOut $G$ is a morphism from the graph output object $n$ to Node $G$, we use the adaption
ψ : Mor₁(\text{Node } G)(\text{Node } G') to reflect the structuring change from \text{Node } G to \text{Node } G'. That is, gOut G' is constructed as the composition gOut G \circ₁ \psi.

- eOut G' is a morphism mapping each edge to its output inner node. We use the identity morphism for eOut, letting each edge and its unique output inner node have the same index. Alternatively, we could have eOut G' implemented as eOut G with the chosen inner node removed from the range and also the edge which produces the chosen inner node removed from the domain. It could be achieved through composing a morphism from Edge G' to Edge G before eOut G and a morphism from Inner G to Inner G' behind eOut G.

- eArity G' is a morphism from Edge G' to \text{SG-object } N. We reuse eArity from Edge G to N in the construction of eArity G'. Therefore we need a morphism from Edge G' to Edge G. For this purpose, we use \phi : Mor₁(\text{Inner } G')(\text{Inner } G) illustrated in Fig. 7.3. \phi will be constructed in “Constructing \phi and \psi” below. Then \phi \circ₁ eOut^{-1} is such a morphism we need from Edge G' to Edge G. Since eOut G' is identity morphism, we use \phi = (\text{Id}_1 : \text{Edge } G' \rightarrow \text{Inner } G') \circ₁ \phi as the embedding from Edge G' to Inner G. Therefore we construct eArity G' as \phi \circ₁ eOut^{-1} composed with eArity G. With this definition of \phi \circ₁ eOut^{-1} : \text{edge } G' \rightarrow \text{edge } G we can further reuse the components of G in the construction of G'.

- Elab G' is constructed in a similar way as eArity G', using \phi \circ₁ eOut^{-1}.

- Eln G' is also constructed using \phi \circ₁ eOut^{-1}. One difference is that the target object of the morphism is the “dependent object” representing Vec (Node G'), that is, \sum n : \mathbb{N} \cdot \text{Vec (Node } G') \cdot n. Therefore the node adaptation \psi is involved in the construction of eln G' from eln G reflecting the transformation from Node G to Node G'.
Constructing $\nu$, $\phi$ and $\psi$

Below we construct $\phi : \text{Mor}_1 (\text{Inner G}') (\text{Inner G})$ and $\nu : \text{Mor}_1 1 (\text{Inner G})$. We obtain $\phi$ as the component $\kappa$ of the constructed $\nu$-$\phi$-Coproduct, and $\nu$ as $\iota$. Note that in the construction, Coproduct-$\iota$-Iso is a Coproduct construction (shown in Sect. 3.1), and splitElem is a Coproduct which will be supplied with details in Sect. 7.4.

\[
\begin{align*}
\text{chosenInner}' : & \text{Fin (succ)} \\
\text{chosenInner}' = & \equiv -\text{subst Fin InnerG} \\
\equiv & \text{num chosenInner}
\end{align*}
\]

\[
\begin{align*}
\nu$-$\phi$-Coproduct & = \text{Coproduct-$\iota$-Iso (splitElem chosenInner')} \\
& (C.\text{invIso} (C.\equiv -\text{Iso InnerG} \equiv \text{num}))
\end{align*}
\]

\[
\begin{align*}
\text{open } \text{Coproduct } \nu$-$\phi$-Coproduct & \text{ public using ( ) renaming ( } \\
& (\iota \text{ to } \nu; \kappa \text{ to } \phi; ...)
\end{align*}
\]

Below we obtain $\psi$ from $\psi$Iso which is an isomorphism between Node G and Node G'. In the implementation of $\psi$Iso, splitElemIso chosenInner' (which will be introduced in Sect. 7.4) supplies the structuring transformations between object Inner G and object $1 \equiv \text{Inner G}'$, where the latter stands for chosenInner together with Inner G'.

We present the proof of the isomorphism between Node G and Node G' in the calculational style. The whole proof is a compositional isomorphism which is enclosed in a $\iota$-begin - $\iota$-end pair. The lines without ↓-( ... ) or ↑-( ... ) are objects where each one is isomorphic to its previous line and the line behind. E.g., $m \equiv \text{suc k}$ is isomorphic to $m \equiv \text{Inner G}$ and $m \equiv (1 \equiv k)$. Each line formatted by ↓-( f ) is just isoMor f from the previous object to the object behind, and ↑-( f ) is the inverse $f^{-1}$ of the isomorphism $f$. As a special case, ↓-() denotes the identity isomorphism required.

\[
\begin{align*}
\psi$Iso : & C.\text{Iso (Node G) NodeG}' \\
\psi$Iso = & $\iota$-begin \\
& \text{Node G} \\
& ↓-() \\
& \text{m }\equiv (\text{Inner G}) \\
& ↓-( C.\text{IdIso} \{ m \} \oplus C.\equiv -\text{Iso InnerG} \equiv \text{num} ) \\
& \text{m }\equiv (\text{suc k}) \\
& ↑- ( C.\text{IdIso} \{ m \} \oplus \text{iso splitElemIso chosenInner}' ) \\
& \text{m }\equiv (1 \equiv k) \\
& ↓-( \equiv -\text{assocL-Iso} \{ m \} \{1\} \{k\} ) \\
& \text{(m }\equiv 1 \equiv k \\
& ↓-() \\
& \text{NodeG'} \\
& \iota$-end \\
\text{open C.\text{Iso } psiIso public using ( ) renaming ( isoMor to psi; _1 to psi^{-1} )}
\end{align*}
\]
7.4 Object Structuring Transformation

In this section, we supply the isomorphism \( \text{splitElemIso} \) which contains the transformation morphisms between two different structures of object \( 1 \sqcup k \). It is used by the construction of the coproduct \( \text{splitElem} \) below, and also by the implementation of \( \psi_{\text{Iso}} \) in Sect. 7.3.2. Then we supply \( \text{splitElem} \) which has been used by \( \upsilon_{\phi}\text{-Coproduct} \) in Sect. 7.3.2.

\( \text{splitElemIso} \) and \( \text{splitElem} \) are contained in a module which requires an argument \( i : \text{Fin} (\text{suc } k) \). This argument is used in specifying the structure of object \( 1 \sqcup k \) (which is also the natural number \( \text{suc } k \)) by three components: \( k_1, 1 \) and \( k_2 \) where \( k_1 = \text{toN } i \) and \( k_1 \sqcup k_2 \equiv k \) (\( \text{toN } \) converts \( i \) to a natural number).

\( \text{splitElemIso} \) is an isomorphism in \( \text{vecCategory} \) of object \( 1 \sqcup k \) which contains a transformation between two different structures. isoMor of \( \text{splitElemIso} \) is the transformation morphism from the structure \( 1 \sqcup (k_1 \sqcup k_2) \) to \( k_1 \sqcup (1 \sqcup k_2) \) used in the construction of Coproduct \( \text{splitElem} \) below. Besides that, \( \text{splitElemIso} \) is used in Sect. 7.3.2 for the implementation of \( \psi_{\text{Iso}} \).

Recalling our formalisation for Coproduct in Sect. 3.1, the choice of coproducts — \( \text{vecCoproduct} \) is fixed along with the instantiation of the parameter category by \( \text{vecCategory} \).

Next we construct \( \text{splitElem} \) which is the Coproduct \( 1 k \) used to define \( \upsilon_{\phi}\text{-Coproduct} \) in Sect. 7.3.2. It is constructed through applying \( \text{Coproduct-}\sharp\text{-Iso} \) (see Sect. 3.1) to the coproduct \( \text{vecCoproduct} \ 1 k \) and the isomorphism \( \text{splitElemIso} \) where the structuring transformation happens.

\[
\text{splitElem} : \text{Coproduct } 1k \\
\text{splitElem} = \text{Coproduct-}\sharp\text{-Iso}(\text{vecCoproduct } 1k)\text{splitElemIso}
\]
Assume open Coproduct splitElem using () renaming (ι to ι'; κ to κ'; Δ to Δ') for the following. In Coproduct splitElem:

- obj is object 1 ⊔ k;
- ι' is ι {1} {k} ; isoMor splitElemIso which is a morphism from 1 to obj where ι is from vecCoproduct;
- κ' is κ {1} {k} ; isoMor splitElemIso which is a morphism from k to obj where κ is from vecCoproduct;
- for any object x, morphisms F : Mor₁ 1 x and G : Mor₁ k x, the coproduct universal morphism (denoted as F ⊔' G) is a unique morphism from 1 ⊔ k to x.
Chapter 8

Correctness of Acyclic Term Graph Decomposition

In this chapter, we provide an overview of the proof for the correctness of the acyclic term graph decomposition presented in Chapter 7. That is, for any given acyclic term graph \( G \), the result in GSME of the decomposition specifies an acyclic term graph assembled from primitive acyclic term graphs which is isomorphic to \( G \). The details of the formalisation are supplied on-line (Zhao, 2018).

8.1 Formalisation of the Correctness

The correctness for acyclic term graphs decomposition is formalised as follows:

\[
\text{decomposeVDHG-correctness} : \{m \; n : \mathbb{N}\}
\rightarrow (G : \text{VDHG3} \; m \; n)
\rightarrow (\text{eOutIso} : C.\text{IsIso} (\text{eOut} \; G))
\rightarrow (\text{D-progFin} : \text{IsProgressivelyFinite} (\text{predRelD} \; G))
\rightarrow \text{GSMEsem-VDHG3GSM} (\text{decomposeVDHG1} \; G \; \text{eOutIso} \; \text{D-progFin}) \approx G
\]

The theorem states that: For any given graph \( G \) of type VDHG3 (therefore it is finite), if \( \text{eOut} \; G \) is an isomorphism (that is, \( G \) is a jungle, or a term graph in the sense of Corradini and Gadducci (1999a)), and if further the node adjacency relation \( D \) of \( G \) is progressively finite (that is, \( G \) is acyclic in the sense of Schmidt and Ströhlein (1993)), then there is a graph isomorphism between \( G \) and the term graph evaluated from the GSME encoding produced by the decomposition (Sect. 7.2) of \( G \).

\text{GSMEsem-VDHG3GSM} is the mapping for morphisms of the gs-monoidal functor mentioned in Sect. 7.1 from the gs-monoidal category of expressions (Sect. 6.2) to the

68
gs-monoidal category of VDHGs. That is, GSMEsem-VDHG3GSM evaluates the GSME encoding for the decomposition of G to a term graph.

However, due to the limited ability of the Agda version 2.5.2, it runs out of heap (we use a machine with the maximum CPU speed of 3300MHz and RAM of 16GB) when computing graph isomorphism with GSMEsem-VDHG3GSM. Instead, we use a locally directly-defined mapping GSME-LocalVDHG3:

\[
\begin{align*}
\text{GSME-LocalVDHG3} : \{m n : \mathbb{N}\} & \rightarrow \text{GSME} \Sigma m n \rightarrow \text{VDHG3} m n \\
\text{GSME-LocalVDHG3} (\text{prim}_e f) &= \text{Prim} f \\
\text{GSME-LocalVDHG3} (\text{id}_e n) &= \text{idADHG} \{n\} \\
\text{GSME-LocalVDHG3} (X_{e m n}) &= \text{exchADHG} m n \\
\text{GSME-LocalVDHG3} (∇_e n) &= \text{dupADHG} \{n\} \\
\text{GSME-LocalVDHG3} (l_{e n}) &= \text{termADHG} \{n\} \\
\text{GSME-LocalVDHG3} (e_1 \otimes_{e} e_2) &= \text{parADHG} (\text{GSME-LocalVDHG3} e_1) \\
&\hspace{1cm} (\text{GSME-LocalVDHG3} e_2) \\
\text{GSME-LocalVDHG3} (e_1 \triangleright_{e} e_2) &= \text{seqADHG} (\text{GSME-LocalVDHG3} e_1) \\
&\hspace{1cm} (\text{GSME-LocalVDHG3} e_2)
\end{align*}
\]

To show the equivalence of GSME-LocalVDHG3 and GSMEsem-VDHG3GSM, we have the following theorem:

\[
\text{GSME-LocalVDHG3} \approx \text{GSMEsem-VDHG3GSM} : \{m n : \mathbb{N}\} \{e : \text{GSME} \Sigma m n\} \\
\rightarrow \text{ADHGIso} (\text{GSME-LocalVDHG3} e) \\
&\hspace{1cm} (\text{GSMEsem-VDHG3GSM} e)
\]

If Agda’s ability is improved in the future, then this by-pass will no longer be needed.

Therefore, we formalise the correctness of acyclic term graph decomposition as the following theorem:

\[
\text{decomposeVDHG-correctness' :} \{m n : \mathbb{N}\} \\
\rightarrow (G : \text{VDHG3} m n) \\
\rightarrow (eOutIso : \text{C.IsIso} (eOut G)) \\
\rightarrow (\text{D-progFin :} \text{IsProgressivelyFinite (predRelD G)}) \\
\rightarrow \text{GSMEsem-LocalVDHG3} (\text{decomposeVDHG1}_{eOutIso D-progFin}) \approx G
\]

Theorem \text{decomposeVDHG-correctness'} is proved by case analysis on non-discrete acyclic term graphs (Sect. 8.2) and discrete acyclic term graphs (Sect. 8.3).

### 8.2 Non-discrete Acyclic Term Graphs

In this section, we supply the correctness proof for the decomposition of non-discrete term graphs (Sect. 7.3.2).
Recall in Sect. 7.3.2, the case of non-discrete acyclic term graphs in decomposeVDHG₁ (illustrated by Fig. 7.1 and Fig. 7.2 in Sect. 7.2) is implemented as below:

\[(E₁ \bowtie_e E₂) \bowtie_e \text{decomposeVDHG₁} G' \leftarrow \text{refl} \equiv \text{refl} C.\text{Id-isIso predRelDG'} \equiv \text{progFin}\]

Let \(E'\) denote the GSME encoding for the constructed graph \(G'\), which is:

\[
\text{decomposeVDHG₁} G' \equiv \text{refl} \equiv \text{refl} C.\text{Id-isIso predRelDG'} \equiv \text{progFin}
\]

Then the correctness is formalised as:

\[
\text{GSME-LocalVDHG3} ((E₁ \bowtie_e E₂) \bowtie_e E') \approx G
\]

It is proved by a composition of two graph isomorphisms below:

### First Isomorphism

The first one is the graph equivalence between \(\text{GSME-LocalVDHG3} ((E₁ \bowtie_e E₂) \bowtie_e E')\) and the graph \(\text{seqADHG} (\text{seqADHG} G₁ G₂) G'\). Because \(E₁, E₂\) and \(E'\) are GSME encodings for \(G₁, G₂\) and \(G'\) (constructed in Sect. 7.3.2), the graph evaluation of the GSME composition is equivalent to the corresponding composition of the graphs.

### Second Isomorphism

The second isomorphism is the equivalence of \(\text{seqADHG} (\text{seqADHG} G₁ G₂) G'\) and \(G\), formalised as \(\text{lemma-nonzeroEdge}\) below:

\[
\text{lemma-nonzeroEdge} : \{m n : N\} \rightarrow (G : \text{VDHG3 m n}) \rightarrow (k : N) \rightarrow (\text{EdgeG} \equiv \text{num} : \text{Edge G} \equiv \text{suc k}) \rightarrow (\text{InnerG} \equiv \text{num} : \text{Inner G} \equiv \text{suc k}) \rightarrow (\text{eOut-isIso} : C.\text{IsIso} (\text{eOut G})) \rightarrow (\text{D-progFin} : \text{IsProgressivelyFinite} (\text{predRelD G})) \rightarrow (\text{let open DecompLocal} \{m\} \{n\} G k \rightarrow \text{ADHGIsIso} (\text{seqADHG} (\text{seqADHG} G₁ G₂) G') G)
\]

The main idea of the proof is to construct an instance graph isomorphism (see Sect. 5.5). That is, starting from defining the mappings \(f-I\) of inner nodes and \(f-E\) of edges between two graphs, we incrementally construct the component proofs of the homomorphisms using the details of the decomposition algorithm. For example, among the proofs,
prop-gOut1 : \text{ADHG.gOut}(\text{seqADHG}(\text{seqADHG } G_1 G_2) G') \circ_1 (\text{Id}_1 \circ f) \simeq_1 \text{gOutG}

shows the graph output preservation between \text{seqADHG}(\text{seqADHG } G_1 G_2) G' and \text{G};

and

prop-eOut1 : \text{ADHG.eOut}(\text{seqADHG}(\text{seqADHG } G_1 G_2) G') \circ_1 f \simeq_1 f' \circ_1 \text{eOutG}

shows the preservation of the function for edge outputs.

lemma-nonzeroEdge \{m\} \{n\}
(\text{record } \{\text{Inner} = \circ (\text{succ } k); \text{Edge} = \circ (\text{succ } k); \text{gOut} = \text{gOutG};
: \text{eOut} = \text{eOutG}; \text{eArity} = \text{eArityG};
: \text{ELabel} = \text{ELabelG}; \text{EIn} = \text{EInG})
\}

k \equiv \text{refl } \equiv \text{refl } \text{eOut-isIso } D\text{-progFin}

= \text{record } \{\text{isoMor} = \text{record } \{f_{-1} = f_{-1};
: fE = fE';
: \text{prop-gOut} = \text{prop-gOut1}
: \text{prop-eOut} = \text{prop-eOut1}
: \text{prop-eArity} = \text{prop-eArity1}
: \text{prop-eLabel} = \text{prop-eLabel1}
: \text{prop-eIn} = \text{prop-eIn1}
\}

; \text{isIso} = \text{record } \{f_{-1} = f_{-1};
: fE = fE';
: \text{prop-gOut} = \text{prop-gOut2}
: \text{prop-eOut} = \text{prop-eOut2}
: \text{prop-eArity} = \text{prop-eArity2}
: \text{prop-eLabel} = \text{prop-eLabel2}
: \text{prop-eIn} = \text{prop-eIn2}
\}

; \text{rightInverse} = \text{record } \{f_{-1} = \text{rightInverse}-f_{-1};
: fE = \text{rightInverse}-fE;\}

; \text{leftInverse} = \text{record } \{f_{-1} = \text{leftInverse}-f_{-1};
: fE = \text{leftInverse}-fE;\})\}

where

open \text{DecompLocal } \{m\} \{n\}
(\text{record } \{\text{Inner} = _{-}; \text{Edge} = _{-}; \text{gOut} = \text{gOutG}; \text{eOut} = \text{eOutG};
: \text{eArity} = \text{eArityG}; \text{ELabel} = \text{ELabelG}; \text{EIn} = \text{EInG})
\}

k \equiv \text{refl } \equiv \text{refl } \text{eOut-isIso } D\text{-progFin}

open \text{NodeG-setup } m k
8.3 Discrete Term Graphs

In this section, we supply the correctness proof for the case of discrete term graphs for which decomposition is defined in Sect. 7.3.1. Recall from Sect. 7.3.1 that

\[ \text{GSME0-GSME} \; \Sigma \; (\text{WireTG-GSME0} \; (gOut \; G)) \]

is the implementation for the case of discrete term graphs. Therefore, the correctness is formalised as

\[ \text{lemma-noEdge : } \{m \; n : \mathbb{N}\} \]
\[ \rightarrow (G : \text{VDHG3} \; m \; n) \]
\[ \rightarrow (\text{InnerG} \equiv \text{num} : \text{Inner} \; G \equiv \text{zero}) \]
\[ \rightarrow (\text{EdgeG} \equiv \text{num} : \text{Edge} \; G \equiv \text{zero}) \]
\[ \rightarrow \text{ADHGIsom} \; (\text{GSME-LocalVDHG3} \; (\text{GSME0-GSME} \; \Sigma \; (\text{WireTG-GSME0} \; (gOut \; G)))) \]
\[ G \]

This theorem is proved by the composition of two graph isomorphisms:

\[ \text{lemma-noEdge} \{m\} \{n\} \ (\text{record} \ \{\text{Inner} = .0 \}
\]
\[ ; \text{Edge} = .0 \]
\[ ; gOut = gOut \]
\[ ; eOut = eOut \]
\[ ; eArity = eArity \]
\[ ; \text{ELabel} = \text{ELabel} \]
\[ ; \text{EIn} = \text{EIn} \}) \]
\[ \equiv \text{-refl} \equiv \text{-refl} \]
\[ = \text{evalGSMEv-EQ} \{v = gOut\} \]
\[ A_0.\text{Iso w-gOutDiscG-EQ} \]

where

\[ G = \text{record} \ \{\text{Inner} = 0; \text{Edge} = 0; gOut = gOut; eOut = eOut \]
\[ ; eArity = eArity; \text{ELabel} = \text{ELabel}; \text{EIn} = \text{EIn}\} \]

The first one is \( \text{evalGSMEv-EQ} \) (will be supplied in Sect. 8.4) with argument \( v \) instantiated to \( \{v = gOut\} \). It is the isomorphism between the term graph evaluated from
the GSME encoding for \( G \) and the graph built by applying the discrete term graph construction function \( \text{wireVDHG} \) (Sect. 5.4) on \( gOut \ G \).

The second one is \( \text{w-gOutDiscG-EQ} \) (will be supplied in Sect. 8.4) which is the equivalence of the graph built by applying the discrete term graph construction function \( \text{wireVDHG} \) on \( gOut \ G \) and the given graph \( G \).

### 8.4 Core Lemmas

In this section, we supply the core lemmas for the correctness proof in Sect. 8.3.

**GSME0ToGSME0Eq**

Theorem **GSME0ToGSME0Eq** is a property of GSME0-GSME which is the conversion from GSME0 to GSME (Sect. 6.3). It states that the GSME converted from the GSME0 over any signature \( \Sigma \) is equivalent to the GSME obtained below: First convert the GSME0 to the GSME over the empty signature, then map it to the GSME over \( \Sigma \).

\[
\text{GSME0ToGSME0Eq} : \{ m n : \mathbb{N} \} \{ e : \text{GSME0} m n \} \\
\rightarrow \text{GSME0-GSME} \Sigma e \equiv \text{GSME-map} (\lambda (\_)) (\text{GSME0-GSME} (\lambda n \rightarrow \text{Data.Empty}) e)
\]

We introduce this theorem for the purpose of simplifying the proofs: Assume in the decomposition of VDHG, we used algorithm WireTG-GSME which encodes \( gOut \) as a GSME directly, then in the correctness, we have to prove

\[
\text{WireTG-GSME} \Sigma v \equiv \text{GSME-map} (\lambda (\_)) (\text{WireTG-GSME} (\lambda n \rightarrow \text{Data.Empty}) v)
\]

which states that: The application of algorithm WireTG-GSME on a vector \( v \) over a signature \( \Sigma \) will have the same result as applying WireTG-GSME on \( v \) over empty signature. For the proof we would have to go over the algorithm WireTG-GSME and all its sub-algorithms to prove it.

Since the signature \( \Sigma \) is only required for edge labels, WireTG-GSME does not use signature in intermediate results. Therefore we can use type GSME0 instead of GSME to have a variant algorithm WireTG-GSME0. Then in the correctness we only have to prove

\[
\text{GSME0ToGSME0Eq} : \\
\text{GSME0-GSME} \Sigma (\text{WireTG-GSME0} v) \\
\equiv \text{GSME-map} (\lambda (\_)) (\text{GSME0-GSME} (\lambda n \rightarrow \text{Data.Empty}) (\text{WireTG-GSME0} v))
\]

which is much easier than the previous one.
evalGSMEv-EQ

evalGSMEv-EQ states the graph equivalence between the term graph evaluated from the GSME encoding for vecCategory morphism \( \nu \) and the term graph wireADHG \( \nu \).

\[
evalGSMEv-EQ : \{ m n : N \} \{ \nu : \text{Mor}_1 m n \} \\
\rightarrow \text{ADHGIso} \left( \text{GSME-LocalVDHG3} \left( \text{GSME0-GSME} \Sigma \left( \text{WireTG-GSME0} \nu \right) \right) \right) \\
\left( \text{wireADHG} \nu \right)
\]

Below we briefly introduce the proof ideas by steps:

First, from theorem GSME0ToGSMEΣ-EQ we know that for any \( e : \text{GSME0} m n \):

\[
\text{GSME0-GSME} \Sigma e \equiv \text{GSME-map} (\lambda()) (\text{GSME0-GSME} (\lambda n \rightarrow \text{Data.Empty}))(\text{WireTG-GSME0} \nu)
\]

Therefore, when \( e \) is instantiated to \( \text{WireTG-GSME0} \nu \), we obtain the two equivalent GSMEs. After that, evaluating the equivalent GSMEs by GSME-LocalVDHG3, it will obtain two equivalent graphs. This is the main idea for the first step in the proof for evalGSMEv-EQ:

\[
\text{Step1} : \{ m n : N \} \{ \nu : \text{Mor}_1 m n \} \\
\rightarrow \text{ADHGIso} \left( \text{GSME-LocalVDHG3} \left( \text{GSME0-GSME} \Sigma \left( \text{WireTG-GSME0} \nu \right) \right) \right) \\
\left( \text{wireADHG} \nu \right)
\]

Second, we have theorem GSME⊗ΣPrim-LocalADHG3-Iso-wireADHG, which states that: Given a GSME \( e \) over empty signature \( \varnothing \Sigma \) for encoding the vector \( \nu : \text{vec} \left( \text{Fin} m \right) n \), we construct \( G \) by mapping \( e \) to a GSME \( e' \) over \( \Sigma \) and then evaluating \( e' \) to a graph by GSME-LocalVDHG3; we construct \( G' \) by applying wireVDHG on the result of evaluating \( e \) to a morphism (which is a vec \( \left( \text{Fin} m \right) n \)) in the gs-monoidal category VecWireMonCatGS (which is built from vecCategory). Then \( G \) is equivalent to \( G' \). This is the main idea for the second step in the proof for evalGSMEv-EQ:

\[
\text{Step2} : \{ m n : N \} \{ \nu : \text{Mor}_1 m n \} \\
\rightarrow \text{ADHGIso} \left( \text{GSME-LocalVDHG3} \left( \text{GSME-map} (\lambda()) \left( \text{GSME0-GSME} \left( \lambda n \rightarrow \text{Data.Empty} \right) \right) \right) \right) \\
\left( \text{wireADHG} \left( \text{GSMEsem-VecWireGS} \left( \lambda n \rightarrow \text{Data.Empty} \right) \right) \right) \\
\left( \lambda() \right) \\
\left( \text{GSME0-GSME} \left( \lambda n \rightarrow \text{Data.Empty} \right) \right) \\
\left( \text{WireTG-GSME0} \nu \right)
\]
Finally, we have theorem WireTG-GSME0-correctness, which states that: For any signature $\Sigma$ and Prim (which is the function for generating VDHGs from $\Sigma$), we evaluate a GSME encoding for $v : \text{vec} (\text{Fin} m) n$ to a morphism $v'$ in $\text{VecWireMonCatGS}$. $v'$ is equivalent to $v$. Therefore the graphs constructed by applying wireADHG on $v'$ and $v$ are equivalent. Note that, since theorem WireTG-GSME0-correctness holds for any signature $\Sigma$ and any Prim, then it holds for the empty signature and empty Prim. This is the main idea for the last step in the proof for evalGSMEv-EQ.

$$\begin{align*}
\text{Step3} : \{m n : \mathbb{N}\} \{v : \text{Mor}_1 m n\} \\
\to \text{ADHGiso} \\
\quad (\text{wireADHG} (\text{GSMEsem-VecWireGSM} (\lambda n \to \text{Data.Empty} \bot)) \\
\quad (\lambda ())) \\
\quad (\text{GSME0-GSME} (\lambda n \to \text{Data.Empty} \bot) \\
\quad (\text{WireTG-GSME0} v))) \\
\quad (\text{wireADHG} v)
\end{align*}$$

Combining those three steps in order, we proved evalGSMEv-EQ.

**w-gOutDiscG-EQ**

w-gOutDiscG-EQ states that the graph built by applying wireADHG on $\text{gOut}$ of a discrete term graph $G$ is equivalent to $G$.

$$\begin{align*}
w\text{-gOutDiscG-EQ} : \text{ADHGiso} (\text{wireADHG} (C.\equiv\text{substTrg} (+-0 m) \text{gOut})) G \\
w\text{-gOutDiscG-EQ} = \text{wireEqMorADHGiso gOutG-substTrg} \\
\quad \text{A}_0, \text{Iso w-gOutGInner@-EQ}
\end{align*}$$

It is proved by a composition of two graph isomorphisms:

The first isomorphism is $\text{wireEqMorADHGiso gOutG-substTrg}$, which is the equivalence of the two graphs constructed by applying wireADHG on two equivalent vectors $\text{C.\equiv substTrg} (+-0 m) \text{gOut}$ and $\text{gOut} \uparrow_1 \text{allFin m} + [$], because the vector for $\text{gOut}$ is equivalent to the vector for $\text{gOut}$ composed with the vector $\text{allFin m}$ appended with an empty vector $[]$.

The second isomorphism is the instantiation of $w\text{-gOutGInner@-EQ}$ for the equivalence between the graph constructed by applying wireADHG on $\text{gOut} \uparrow_1 \text{allFin m} + [$] and the given discrete term graph. $w\text{-gOutGInner@-EQ}$ states that: The ADHG constructed by applying wireADHG on $G.\text{gOut} \uparrow_1 (\text{Id } \triangle \text{Inner-isInitial.@})$ is equivalent to the given discrete term graph $G$. Because in the instantiation, $\text{Id}$ is $\text{allFin m}$, $\text{Inner}$ and $\text{Edge}$ of the discrete graphs are initial objects in the underlying category, and all the morphisms starting from $\text{Inner}$ or $\text{Edge}$ are empty vectors ($[]$), then $G.\text{gOut} \uparrow_1 (\text{Id } \triangle \text{Inner-isInitial.@})$ is exactly $\text{gOut} \uparrow_1 \text{allFin m} + [$].
Chapter 9

Term Graph Rewriting

Our ultimate goal is to supply a trusted (correctness guaranteed) tool for term graph rewriting (Sect. 2.1) which is based on a rule concept providing a good mental model. Among the various characterisation of term graph rewriting, the categorial presentation provides the definition of the rewriting over term graphs as categorial concepts and lays the foundation for the development of proof and analysis. That is, in the category theoretic framework, we are allowed to formulate all the concepts of term graph rewritings such as graphs, matches, productions and direct derivations in pure category terms.

When attempting to employ traditional categorial approaches to graph rewriting, two main problems arise: First, categories of “standard” term graph homomorphisms typically do not have all pushouts, since unification translates into pushouts; Second, the interface graphs needed for the double-pushout (DPO) approach are typically not term graphs, but some kind of “term graphs with holes”. Therefore, we adopt a variant of the standard DPO approach for the categorial presentation specialised for Term Graph Rewriting (Kahl, 1997; Habel and Plump, 2012). In the remaining contents, we use “DPO” to refer to our variant of the original double pushout approach where the horizontal morphisms are ADHG homomorphisms (Sect. 5.5.1) and the vertical morphisms are ADHG matchings (Sect. 5.6).

We are also interested at the presentation of the correctness of the term graph rewriting. Corradini and Gadducci (1999a) showed us a road to the categorial formalisation of the semantics of term graphs. This contribution enables us to formalise the semantics of term graphs to obtain an appropriate concept of semantics preservation for term graph rewriting in the DPO approach. That is, the gs-monoidal category of term graphs are freely generated from the signature, therefore the functorial semantics of term graphs is obtainable. More detailed, the gs-monoidal category of term graphs is the initial object in the category of all gs-monoidal categories as objects and gs-monoidal functors as arrows, and there always exists functors from the gs-monoidal
category of term graphs to any gs-monoidal categories, e.g., the semantics for term graphs.

In this chapter, we present the formalisation of term graph rewriting in a variant of the standard DPO approach in Sect. 9.1 and the semantics preservation in the term graph rewriting steps in Sect. 9.2. The development in this chapter has not yet been fully formalised in Agda; the theorems in this chapter are formalised as “postulate” (Bove and Dybjer, 2009) which is a mechanism of Agda for assuming that certain constructions exist, without actually defining them. In this way we can write down postulates and reason on the assumption that these postulates are true. Postulates are introduced by the keyword `postulate`.

The full formalisation is available on-line (Zhao, 2018).

9.1 Term Graph Rewriting

In this section, we formalise term graph rewriting (Kahl, 1997; Habel and Plump, 2012) using a variant of the DPO approach.

9.1.1 The Variant of DPO Approach

In the DPO approach, a term graph rewrite step is characterised by a double pushout (DPO) in a suitable category of graphs and graph morphisms. As seen from the concrete example in Fig. 9.1, the “context graph” $H$ can be thought of as obtained from the “application graph” $A$ by deleting all edges and inner nodes of $A$ which have pre-images in $L$, but have no pre-image in $G$. Via the left pushout diagram this deletion is described as an inverse insertion operation, while the right pushout diagram models the actual insertion into $H$ of all edges and inner nodes of $R$ that do not have pre-images in $G$.

Fig. 9.2 is the diagram characterising the example Fig. 9.1 in a DPO structure. From this diagram, we see that the rewriting rule, matching and direct derivation (Sect. 2.1) are represented as graphs and graph morphisms which form two attached pushouts in the category of ADHGs and ADHG matchings.

Our “suitable category” is the category of ADHGs and ADHG matchings (Sect. 5.6), although we would prefer the category of jungles (Sect. 5.4) and jungle matchings (jungles are what is called “term graph” in Corradini and Gadducci (1999a)).

In rewrite steps such as that shown in Fig. 9.2 the “gluing graph” $G$ of $L$ and $R$ in a rewriting rule $p : L \leftarrow G \rightarrow R$ is not typically a jungle, i.e., $G$ always has some “empty area” to accommodate the difference between the internal structures of $L$ and
Figure 9.1: A term graph rewriting derivation in DPO approach
Figure 9.2: A double pushout characterising a direct derivation

R (see Fig. 9.1). More precisely, \( L \) and \( R \) share input and output nodes, but not inner structures, so that there are inner nodes in \( G \) that do not have any producer edge, which violates the jungle property. Therefore, the gluing graph \( G \) and the host graph \( H \) are not jungles, but they still are directed hypergraphs. In fact, we define jungles as acyclic DHGs (see Sect. 5.4), where each non-input node is the output node of exactly one edge.

The interfaces of \( L \) and \( R \) in the rewriting rule remain same in order to preserve type. Same reason is applied to the graph interfaces of \( A \) and \( B \). As we defined in Sect. 5.3, an ADHG homomorphism preserves not only the graph structure but also the graph interface, which is a restricted ADHG matching. Therefore, in our variant DPO, the horizontal morphisms are ADHG homomorphisms instead of matchings. It could be seen in Fig. 9.2 the horizontal arrows \( \Phi, \Psi, \Xi, \Omega \) are ADHG homomorphisms and the vertical arrows \( M_1, X, M_2 \) are ADHG matchings.

9.1.2 Gluing Condition

As in the general DPO graph transformation (Corradini et al., 1997), we also have two kinds of problematic situations to avoid in our DPO approach for term graph rewriting.

One difficulty may occur if a vertex shall be deleted which is connected to an edge that is not part of the match, as shown in Fig. 9.3. Deleting vertex 3 of \( A \) in the PO-complement, as specified by production \( p \), would leave behind the edge \( e \) without a source vertex in \( H \), i.e., later when we obtain \( A' \) by the pushout construction from \( L \leftarrow G \rightarrow H \), we have the source vertex of edge \( e \) connected to nowhere, since we have no idea where should connect to. We say that the match contains a conflict.

Another difficulty may occur as in Fig. 9.4. If a production \( p \), assuming two vertices 4 and 5 and deleting both of them, is applied to a graph \( A \), i.e., 4 and 5 are both mapped to one vertex 5 in \( A \), we obtain \( H \) through PO-complement. Then we obtain \( A' \) by the pushout construction from \( L \leftarrow G \rightarrow H \). However, \( A \) and \( A' \) are not isomorphic.
Figure 9.3: Dangling conflict in term graph rewriting
Figure 9.4: Identification conflict in term graph rewriting
Fig. 9.3 and Fig. 9.4 are examples of the problematic situations represented in the shape of the diagram in Fig. 9.5.

In order to avoid the problematic situations illustrated in Fig. 9.3 and Fig. 9.4 in the DPO approach the match must satisfy an application condition, called *gluing condition* in Corradini et al. (1997). This condition consists of two parts: First, to ensure that $H$ will have no dangling edges, the *dangling condition* requires that if production $p$ specifies the deletion of a vertex of $A$, then it must specify also the deletion of all edges of $A$ incident to that node. Second, *identification condition* requires every element of $A$ that should be deleted by the application of $p$ has only one pre-image in $L$. This condition ensures that the application of $p$ deletes exactly what is specified by the production.

In our approach, using two constraints, we ensure that the two problematic situations will not happen in the direct derivation. We state the two constraints separately as noDanglingConflict and noIdentificationConflict below:

**noDanglingConflict**:

\[
\text{noDanglingConflict} : \{ m_1 m_2 n_1 n_2 : \text{Obj}_1 \} \\
\{ G : \text{ADHG m}_1 n_1 \} \{ L : \text{Jungle m}_1 n_1 \} \{ A : \text{Jungle m}_2 n_2 \} \\
(\Phi : \text{ADHGMor G (Jungle,ADHG3 L)}) \\
(M_1 : \text{JungleMatching L A}) \\
\rightarrow \text{Set} 
\]

**noDanglingConflict** ensures: If $L \xrightarrow{M_1} A$ specifies the deletion of a node of $A$, then among those edges of $A$ which are not in the range of the matching, none is incident to that node. I.e., in the set-based instantiation of the parameter category:

\[
\forall n : \text{Ran } M_1.f-N \bullet \forall e : A.\text{Edge} - \text{Ran } M_1.f-E \bullet \forall i : \text{Ran } (A.\text{eln } e) \bullet i \neq n 
\]

**noIdentificationConflict**:

\[
\text{noIdentificationConflict} : \{ m_1 m_2 n_1 n_2 : \text{Obj}_1 \} \\
\{ G : \text{ADHG m}_1 n_1 \} \{ L : \text{Jungle m}_1 n_1 \} \{ A : \text{Jungle m}_2 n_2 \} \\
(\Phi : \text{ADHGMor G (Jungle,ADHG3 L)}) \\
(M_1 : \text{JungleMatching L A}) \\
\rightarrow \text{Set} 
\]
nullIdentificationConfict ensures: If $L \xrightarrow[M_1]{\text{A}} A$ specifies the deletion of a node of $A$, then the node has only one pre-image in $L$. I.e.:

$$\forall n : \text{Ran } M_1.f\cdot N \land \forall m_1, m_2 : \text{L.Node } \cdot M_1.f\cdot N \cdot m_1 = n \land M_1.f\cdot N \cdot m_2 = n \Rightarrow m_1 = m_2$$

### 9.1.3 The Pushout Complement Construction

In the DPO approach for term graph rewriting, the rewrite step is formulated as two attached pushouts. In this subsection, we construct the left pushout of the two attached pushouts from $G \to L \to A$, and in the next subsection, we will continue to construct the other one.

Given an ADHG homomorphism $\Phi$ from ADHG $G$ to jungle $L$ and a jungle matching $M_1$ from $L$ to jungle $A$, if $\Phi$ and $M_1$ satisfy the gluing condition, function $\text{PO-Complement}'$ results in an ADHG $H$, an ADHG matching $X$ from $G$ to $H$ and an ADHG homomorphism $\Xi$ from $H$ to $A$ such that $X$, $\Phi$, $\Xi$ and $M_1$ form a pushout in ADHG matching category, see Fig. 9.6.

![Figure 9.6: The pushout complement construction](image)

**postulate**

$\text{PO-Complement}' : \{ m_1 \, m_2 \, n_1 \, n_2 : \text{Obj}_1 \}$

$\{ G : \text{ADHG} \, m_1 \, n_1 \} \{ L : \text{Jungle} \, m_1 \, n_1 \} \{ A : \text{Jungle} \, m_2 \, n_2 \}$

$(\Phi : \text{ADHGMor} \, G \, (\text{Jungle}.\text{ADHG3} \, L))$

$(M_1 : \text{JungleMatching} \, L \, A)$

$(\text{no-Dang} : \text{noDanglingConfict} \, \{ G = G \} \, \{ L \} \, \{ A \} \, \Phi \, M_1)$

$(\text{no-Iden} : \text{noldentificationConfict} \, \{ G = G \} \, \{ L \} \, \{ A \} \, \Phi \, M_1)$

$\to \Sigma H : \text{ADHG} \, m_2 \, n_2 \bullet$

$\Sigma X : \text{ADHGMatching} \, G \, H \bullet$

$\Sigma \Xi : \text{ADHGMor} \, H \, (\text{Jungle}.\text{ADHG3} \, A) \bullet$

$\text{IsPushout} \, (\text{MatchingCategory})$

$\{ G' \} \{ H' \} \{ (\text{Jungle}.\text{ADHG3} \, L) \}' \{ (\text{Jungle}.\text{ADHG3} \, A) \}'$

$X \, (\text{matching } \Phi) \, (\text{matching } \Xi) \, M_1$

Note that the result of $\text{PO-Complement}'$ is in the shape of $\Sigma x : A \bullet B$ which is a
variant of the syntax supplied in the Agda standard library for dependent products (see Sect. 7.3.1).

In the implementations, which we leave to future work, the context graph $H$ can be thought of as obtained from the application graph $A$ by deleting all of the edges and inner nodes of $A$ which have a pre-image in $L$, but none in $G$. This idea is formalised as \textit{straight host construction} in (Kahl, 2001), which constructs $G \xrightarrow{X} H \xrightarrow{Ξ} A$ from $G \xrightarrow{Φ} L \xrightarrow{M_1} A$. We can then instantiate \textit{Theorem 5.4.8} in (Kahl, 2001) to prove that the straight host construction produces a pushout complement in \texttt{MatchingCategory}. It is easy to show that the resulting $Ξ$ is actually an \texttt{ADHGMor}, which preserves source and target.

Since we will need $Φ$ to be injective for $B$ being a jungle in the right pushout (Sect. 9.1.4), we need a condition for that $Φ$ is monic Sect. 9.1.4. Since we will also need $X$ to be monic in the right pushout construction, \texttt{PO-Complement} generates a function $\texttt{isMonic (matching} Φ) \to \texttt{isMonic X}$ which is used to derive “$X$ is monic” from “$Φ$ is monic”.

\begin{verbatim}
postulate

PO-Complement : {m_1 m_2 n_1 n_2 : Obj_1}
{ G : ADHG m_1 n_1} {l : Jungle m_1 n_1} {A : Jungle m_2 n_2}
(Φ : ADHGMor G (Jungle.ADHG3 L))
(Φ-isMonic : Category.IsMono MatchingCategory (matching Φ))
(M_1 : JungleMatching L A)
(no-Dang : noDanglingConflict \{G = G\} \{L\} \{A\} Φ M_1)
(no-Iden : noIdentificationConflict \{G = G\} \{L\} \{A\} Φ M_1)
→ let isMonic = Category.IsMono MatchingCategory

in
Σ H : ADHG m_2 n_2 •
Σ X : ADHGMatching G H •
Σ Ξ : ADHGMor H (Jungle.ADHG3 A) •
IsPushout (MatchingCategory)
{G'} \{H'} \{(Jungle.ADHG3 L)'} \{(Jungle.ADHG3 A)'}
X (matching Φ) (matching Ξ) M_1
× (isMonic (matching Φ) → isMonic X)
\end{verbatim}

9.1.4 The Right Pushout Construction

After the left one of the two attached pushouts in DPO is constructed, the span for the right pushout is available, i.e., $H \leftarrow G \to R$. In this subsection, we formalise the construction of the \textit{right} pushout, which is illustrated as the diagram in Fig. 9.7. The pushout construction models the actual insertion into $H$ of all edges and inner nodes of $R$ that do not have pre-images in $G$. 
In this right pushout construction, we desire the result in pushout object $B$ to be a jungle, which is not trivial.

First, the situation shown in Fig. 9.8 would lead to $B$ not being a jungle.

The problem is caused by the following: The inner node $a$ in $G$ is the output inner node of no edge, and it is mapped to $b$ and $c$ in $R$ respectively $H$, where $b$ and $c$
are output inner nodes of edges labelled with $F$ respectively $S$. This situation can be avoided if $X$ preserves the hole of $G$ in $H$.

However, since the $\Phi$-image of node $a$ in $L$ has to be either an input node or the output of an edge, "$X$ preserves the hole of $G$ in $H$" holds if the rule LHS $\Phi$ is injective. (If the image of $a$ is an input node, then, with $\Phi$ preserving the graph interface, it cannot be injective. If the image of $a$ is the output node of an edge in $L$, then the image in $A$ (which is a jungle) of that edge needs to be also the image of the $S$-edge in $H$, which contradicts the left-hand pushout.)

Second, also the example DHG matching pushout in Fig. 9.9 fails to produce a jungle $B$. This situation can be avoided by restricting the matching $X$ to be injective.

Figure 9.9: Non-monic $X$ result in $B$ not being a jungle
In effect, both constraints together correspond to the restriction to the “regular monos” of Corradini and Gadducci [2005, Prop. 4.3].

Then the right pushout construction is formalised as RightPO: Given an ADHG homomorphism $\Psi$ from ADHG $G$ to jungle $R$ and an monic ADHG matching $X$ from $G$ to ADHG $H$, it results in a jungle $B$, an ADHG homomorphism $\Omega$ from $H$ to $B$ and a jungle matching $M_2$ from $R$ to $B$ such that $X$, $\Psi$, $\Omega$ and $M_1$ form a pushout in the ADHG matching category.

The edges and inner nodes of $B$ are obtained by the function application of pushout construction from the underlying category on the given span $H \leftarrow G \rightarrow R$. Then we construct the necessary graph components of $B$ such as $eOut$, $eLabel$ and $eIn$. For $H \xrightarrow{\Omega} B \xrightarrow{M_2} R$, we have $f-I$ and $f-E$ of $\Omega$ and $M_2$ also obtained from pushout construction function, we then have to prove that they preserve the graph structures. Following this way, what we obtain are: $B : \text{ADHG}_2 n_2$, $\Omega' : \text{ADHGMatching}_2 R$ and $M_2' : \text{ADHGMatching}_2 R$.

Then since $B$ is a jungle, ADHG matching $M_2$ from jungle $R$ to $B$ is automatically a jungle matching. Also from the construction, $B$ has the same interface as $H$, then ADHG matching $\Omega$ is an ADHG homomorphism. Finally, the proof for that $X$, $\Psi$, $\Omega$ and $M_2$ constitute a pushout is constructed based on the above constructions.

9.2 Semantics Preservation of Term Graph Rewriting

Corradini and Gadducci [1999a] showed that the category of term graphs can be extended to a gs-monoidal category. Since the term graphs over a given signature are
arrows of the gs-monoidal category freely generated by it, then there always exists
a unique functor from the gs-monoidal category of term graphs to any gs-monoidal
category. These showed us the direction to obtain the functorial semantics for the cat-
egorical characterisation of term graphs, i.e., the semantics of a term graph is obtained
through a uniquely defined functor to a gs-monoidal category chosen for semantics.
(This will most typically be some category of sets, with some set \( V \) chosen as set of
values of term graph node; a term graph with \( m \) inputs and \( n \) outputs then has a
function of type \( V^m \rightarrow V^n \) as semantics.)

In order to prove the semantics preservation of the application graph \( A \) and the result
graph \( B \) in the DPO, we need to transfer the necessary information “across the host
graph \( H \)” at the ADHG level. Rather than the decomposition of term graphs into
gs-monoidal expressions as described by Corradini and Gadducci (1999a) which needs
to extend this expression type into a type of contexts by including “placeholders” as
proposed by Corradini and Gadducci (2002), we define contexts at the level of graphs.

9.2.1 Semantics of Term Graphs

Semantics of untyped term graphs can be formalised in a category of setoids, i.e., the
objects are setoids and arrows are functions between these setoids. We can choose a
setoid \( V \) as the type of values that can be associated with a single term graph node,
which, in applications such as the code generation system Coconut by Anand and
Kahl (2009), stand for a single CPU register. Therefore, \( V \) will be the semantics of
the natural number 1, considering 1 to be the object standing for single-node term
graph interfaces. Since the monoidal operation on objects in the gs-monoidal category
of setoids and functions used for semantics is the Cartesian product, the semantics
for the natural number 1+1 is interpreted as \( V \times V \), and any natural number \( n \) has the
set of \( n \)-tuples of \( V \) values as semantics. Term graphs where the numbers of inputs
and outputs are the same can be understood to define state changing computations.
For instance, each carrier set element of the setoid represents a specific content of
a register which is understood as the state of a register, and the functions between
setoids can be understood as state changing. The equivalence relation of the setoid
is used as the equivalence of states. This is the “functional semantics” in the sense of
“functional requirements”.

9.2.2 Context

In order to prove the semantics preservation of the application graph \( A \) and the result
graph \( B \) in the DPO, we need to transfer the necessary information “across the host
graph \( H \)” at the ADHG level. For this purpose, we have to factor out the unaffected
internal structure of \( A \) as term graphs during the rewriting. Fig. 9.10 is the illustration
for how \( A \) is expressed by the unaffected internal structures and the \( M_1 \)-image of \( L \).

\[
\begin{align*}
\text{Figure 9.10: The context view of the application graph}
\end{align*}
\]

Let context denote the term graph containing the unaffected internal structures of \( A \), and context \( [L] \) denote the composition of context and \( L \). According to the illustration of Context in Fig. 9.10, a \( m,n \)-context \( (k, A_1, A_2) \) for a \( i,j \)-parameter consists of:

- an internal interface object \( k \),
- a top part jungle \( A_1 : m \rightarrow (k \uplus i) \), and
- a bottom part jungle \( A_2 : (k \uplus j) \rightarrow n \).

\[
\begin{align*}
\text{record Context} (m \, n \, i \, j : \text{Obj}_1) : \text{Set} (\ell_{o_1} \cup \ell_{c_1} \cup \ell_{c_2} \cup \ell_{s_1} \cup \ell_{s_2}) \text{ where} \\
\text{field} \quad k : \text{Obj}_1 \\
& A_1 : \text{Jungle} \, m \, (k \uplus i) \\
& A_2 : \text{Jungle} \, (k \uplus j) \, n
\end{align*}
\]

IsAcyclic ensures the given term graph is acyclic:

\[
\begin{align*}
\text{postulate} \\
\quad \text{IsAcyclic} : \{ \text{m n : \text{Obj}_1} \} \, (G : \text{ADHG m n}) \rightarrow \text{Set} (\ell_{o_1} \cup \ell_{c_1} \cup \ell_{c_2} \cup \ell_{s_1} \cup \ell_{s_2})
\end{align*}
\]

It can be formalised as “the node adjacency relation \( D \) [Kahl 1996] of \( G \) is progressively finite” (that is, \( G \) is acyclic in the sense of Schmidt and Ströhlein 1993).

Below we obtain context through function getContext. Based on the above description, in order to make our life easier, the matching \( L^{M_1} \cdot A \) is restricted to be monic to ensure that different input nodes of \( L \) will not be mapped into a same node in \( A \). Otherwise the graph interface of \( L \) might be different to the “hole” in context for
embedding $L$. Although we may do some adaptations on the “improper” context to make the replacing possible, it would be expensive, i.e., we have to identify the occurrences of the nodes in $A$ for which have different pre-images, and then add duplications for them.

In the following, we use “$\cdot$” as sequential composition operator for term graphs, and “$\boxtimes$” for parallel composition. “$\mathbb{I}_k$” denotes the identity term graph with $k$ inputs that are also its outputs, in the same sequence. Context is obtained through function $\text{getContext}$ below: Assume $L : \text{Jungle } m_1 n_1$ and $A : \text{Jungle } m_2 n_2$ are given which are acyclic, and $M_1 : L \to A$ is an injective jungle matching, then there is a context $(k,A_1,A_2)$ such that $A = A_1 \# (\mathbb{I}_k \boxtimes L) \# A_2$.

\[
\text{open Category JungleCategory using } () \text{ renaming } (_\approx \text{ to } \approx \text{J} ; _\boxtimes \text{ to } \boxtimes \text{J} )
\]
\[
\text{open Category ADHGCategory using } () \text{ renaming } (_\approx \text{ to } \approx \text{G} ; _\boxtimes \text{ to } \boxtimes \text{G} )
\]
\[
_\boxtimes \text{J} = \text{parJungle}
\]
\[
_\boxtimes \text{G} = \text{parADHG}
\]

\text{postulate}
\[
\text{getContext} : \{m n i j : \text{Obj}\}
\]
\[
(L : \text{Jungle i j})
\]
\[
(L\text{-isAcyclic : IsAcyclic (Jungle.ADHG 3 L))}
\]
\[
(A : \text{Jungle m n})
\]
\[
(A\text{-isAcyclic : IsAcyclic (Jungle.ADHG 3 A))}
\]
\[
(M_1 : \text{JungleMatching L A})
\]
\[
(M_1\text{-isMono : Category.IsMono}
\]
\[
\text{JungleMatchingCategory} \{A = L J' \} \{A J'\} M_1)
\]
\[
\rightarrow
\]
\[
\Sigma \text{ context} : \text{Context m n i j} \bullet
\]
\[
\text{let open Context context in}
\]
\[
A \approx J A_1 \# J (\text{idJungle } \{k\} \boxtimes J L) \# J A_2
\]

We can construct $A_1$ and $A_2$ by reachability, e.g., in $A$, all the elements reachable from the image of the graph input i of $L$ belong to the image of $L$ and $A_2$, and the rest elements belong to $A_1$. Next, among the elements belong to the image of $L$ and $A_2$, those reachable from the image of the graph output j of $L$ belong to $A_2$.

\subsection{Express Graphs Through Context}

In this subsection, we supply the formalisation for that the host graph $H$ and the result of the rule application $B$ in the DPO characterisation for term graph rewriting can be factored through $\text{context}$.

\text{AHBcomp} states that: Given ADHG homomorphisms $L \xrightarrow{\Phi} G \xrightarrow{\Psi} R$ and a monic jungle matching $L \xrightarrow{M_1} A$ where $L$, $G$, $R$ and $A$ are acyclic, as well as that $\Phi$ and $M_1$ are
monic and satisfy the gluing condition, there exists a context which contains jungles $A_1$ and $A_2$ such that $H$ is isomorphic to

$$(\text{Jungle.ADHG}3 \ A_1) \ R (\text{Jungle.ADHG}3 \ (\text{idJungle} \ \{x\}) \ R G) \ R (\text{Jungle.ADHG}3 \ A_2)$$

and $B$ is isomorphic to $A_1 \ R (\text{idJungle} \ \{x\} \ R J) \ R A_2$, where $H$ is the result of the pushout complement construction and $B$ is the pushout object of the right pushout construction.

**postulate**

$$\text{AHBcomp} : \{m_1 \ m_2 \ n_1 \ n_2 : \text{Obj}1\}$$

$$\{G : \text{ADHG} \ m_1 \ n_1\} \ \{L \ R : \text{Jungle} \ m_1 \ n_1\} \ \{A : \text{Jungle} \ m_2 \ n_2\}$$

$$(L\text{-isAcyclic} : \text{IsAcyclic} (\text{Jungle.ADHG}3 \ L))$$

$$(A\text{-isAcyclic} : \text{IsAcyclic} (\text{Jungle.ADHG}3 \ A))$$

$$(R\text{-isAcyclic} : \text{IsAcyclic} (\text{Jungle.ADHG}3 \ R))$$

$$(G\text{-isAcyclic} : \text{IsAcyclic} G)$$

$$(\Phi : \text{ADHG} \text{Mor} \ G \ (\text{Jungle.ADHG}3 \ L))$$

$$(\Psi : \text{ADHG} \text{Mor} \ G \ (\text{Jungle.ADHG}3 \ R))$$

$$(M_1 : \text{JungleMatching} \ L \ A)$$

$$(M_1\text{-isMono} : \text{Category}\text{.IsMono} \ \text{JungleMatchingCategory} \ M_1)$$

$$(\Phi\text{-isMono} : \text{Category}\text{.IsMono} \ \text{MatchingCategory} \ \text{matching} \ \Phi)$$

$$(\text{no-Dang} : \text{noDanglingConflict} \ \{G = G\} \ \{L\} \ \{A\} \ \Phi \ M_1)$$

$$(\text{no-Iden} : \text{noIdentificationConflict} \ \Phi \ M_1)$$

$$\rightarrow \ \text{let} \ (H , X , , \text{(_, mono} \ \Phi \ \text{toX}) ) = \text{PO-Complement}$$

$$\Phi \ M_1 \ \text{no-Dang} \ \text{no-Iden} \ \Phi\text{-isMono}$$

$$\text{in}$$

$$\text{let} \ (B , , , ) = \text{RightPO} \ \{G = G\} \ \{R\} \ \{H\}$$

$$\Psi \ X \ (\text{mono} \ \Phi \ \text{toX} \ \Phi\text{-isMono})$$

$$\text{in}$$

$$\text{let} \ (\text{context} , \text{isContext}) = \text{getContext} \ L \ \text{L-isAcyclic}$$

$$\text{A A-isAcyclic} \ M_1 \ M_1\text{-isMono}$$

$$\text{in}$$

$$\text{let} \ \text{open} \ \text{Context context} \ \text{in}$$

$$H = G \ (\text{Jungle.ADHG}3 \ A_1)$$

$$\ R (G \ (\text{Jungle.ADHG}3 \ (\text{idJungle} \ \{k\}) \ R G)$$

$$\ R (G \ (Jungle.ADHG3 \ A_2)$$

$$\times B = J A_1 \ R (idJungle \ \{k\} \ R J) \ R J A_2$$

The rule based transformation system interacts with individual graph items such as edges and inner nodes, but it presents the result to user at a higher level and hides details in a “blackbox”. We use the intuitive understanding of the procedure for reasoning the implementation.
Proof sketch:

The pushout complement which obtains $H$ can be thought of as deleting all of the edges and inner nodes of $A$ which have pre-images in $L$, but none in $G$. In other words, it can be thought of as first deleting the set of edges and inner nodes of $A$ which have pre-images in $L$, and then filling in the set of edges and inner nodes of $A$ which have pre-image in $G$. Since $A$ is isomorphic to the composition of context and $L$ (by the definition of context), then the above manipulations of edges and inner nodes have the same effect as removing $L$ and filling $G$ in the composition of context and $L$.

The right pushout models obtaining $B$ as the actual insertion into $H$ of all edges and inner nodes of $R$ that do not have pre-images in $G$. In other words, it could be thought of as removing the set of edges and inner nodes from $H$ which have pre-images in $G$, and then filling in the set of edges and inner nodes which have pre-images in $R$. Because from above we have $H$ is isomorphic to the composition of context and $G$, and in the similar way of obtaining that result, the manipulations of edges and inner nodes have the same effect as removing $G$ and filling $R$ in the composition of context and $H$.

9.2.4 The Formalisation of Semantics Preservation

Finally, in this subsection, we formalise the semantics preservation of the term graph rewriting characterised in DPO approach.

Assume $\text{semGSMCategory}$ is the gs-monoidal category chosen as the semantics for term graphs, and $\text{semanGSMFunctor}$ is the functor from the gs-monoidal category of term graphs to $\text{semGSMCategory}$, where $\text{jungleSem}$ is the morphism mapping in the functor.

```haskell
open import Category.Functor using (Functor)
open import Category.MonoidalCategory
  using (module MonoidalCategory ; MonoidalCategory)
module _ (semGSMCategory : GSMonoidalCategory)
  (ℓ₁ ⟦ c₁ ≪ c₂ ≪ s₁ ≪ s₂) (ℓ₂ ⟦ c₁ ≪ c₂ ≪ s₂) Obj₁)
  (semanGSMFunctor : GSMFunctor Jungle-GSMonCat semGSMCategory)
where
open GSMonoidalCategory semGSMCategory using ()
  renaming (monCat to semMonCat)
open MonoidalCategory semMonCat using () renaming (category to semCategory)
open Category semCategory using () renaming (_ ≈ _) to _≈Sem_
open GSMFunctor semanGSMFunctor using (functor)
open Functor functor renaming (mor to jungleSem)
```

92
Since the term graphs over a given signature are arrows of the gs-monoidal category freely generated by it, then there always exists a unique functor $F_1$ from the gs-monoidal category of term graphs to any gs-monoidal category, e.g., the gs-monoidal category of expressions (Sect. 6.2). This functor $F_1$ can be constructed on the mapping from term graphs to gs-monoidal expressions (by term graph decomposition in Chapter 7). We know that gs-monoidal expressions are also freely generated from signature, therefore there also exists a unique functor $F_2$ from the gs-monoidal category for expressions to any gs-monoidal category, e.g., $\text{semGSMCategory}$. This functor $F_2$ can be constructed on the mapping from gs-monoidal expressions to arrows of any gs-monoidal category (by functor $\text{GSMEsem}$ for the category of expressions in Sect. 6.2). Therefore functor $\text{semanFunctor}$ from the gs-monoidal category for term graphs to $\text{semGSMCategory}$ can be constructed as the composition of $F_1$ and $F_2$.

Then we have the semantics preservation of the rewriting procedure if given the semantics preservation of the rewriting rules: Given ADHG homomorphisms $\Phi$ and $M_1$ are monic and satisfy the gluing condition, the equivalence of applications of $\text{jungleSem}$ on $L$ and $R$ implies the equivalence of applications of $\text{jungleSem}$ on $A$ and $B$, where $B$ is the result of the right pushout construction.

\[
\text{postulate}
\]

\[
\text{semanPreserved} : \{ m_1 m_2 n_1 n_2 : \text{Obj}_1 \}
\]

\[
\{ G : \text{ADHG} m_1 n_1 \} \{ L R : \text{Jungle} m_1 n_1 \} \{ A : \text{Jungle} m_2 n_2 \}
\]

\[
(L:\text{-isAcyclic} : \text{IsAcyclic}\ (\text{Jungle}.\text{ADHG}\ A))
\]

\[
(A:\text{-isAcyclic} : \text{IsAcyclic}\ (\text{Jungle}.\text{ADHG}\ A))
\]

\[
(R:\text{-isAcyclic} : \text{IsAcyclic}\ (\text{Jungle}.\text{ADHG}\ G))
\]

\[
(G:\text{-isAcyclic} : \text{IsAcyclic}\ G)
\]

\[
(\Phi : \text{ADHGMor} G (\text{Jungle}.\text{ADHG}\ L))
\]

\[
(\Psi : \text{ADHGMor} G (\text{Jungle}.\text{ADHG}\ R))
\]

\[
(M_1 : \text{JungleMatching} L A)
\]

\[
(M_1:\text{-isMono} : \text{Category}.\text{IsMono}
\]

\[
\text{JungleMatchingCategory}\ \{ A = L J' \} \{ A J' \} M_1
\]

\[
(\Phi:\text{-isMono} : \text{Category}.\text{IsMono} \text{MatchingCategory} (\text{matching} \Phi))
\]

\[
(\text{no-Dang} : \text{noDanglingConflict}\ \{ G = G \} \{ L \} \{ A \} \Phi M_1)
\]

\[
(\text{no-Iden} : \text{noIdentificationConflict}\ \Phi M_1)
\]

\[
\rightarrow \text{let}\ (H, X, _ , (_ , \text{mono}\Phi\text{toX})) = \text{PO-Complement} \Phi M_1
\]

\[
\text{no-Dang} \text{no-Iden} \Phi:\text{-isMono}
\]

\[
\text{in}
\]

\[
\text{let}\ (B, _ , _ , _ ) = \text{RightPO}\ \{ G = G \} \{ R \} \{ H \}
\]

\[
\Psi X (\text{mono}\Phi\text{toX} \Phi:\text{-isMono})
\]

\[
\text{in}
\]

\[
\text{jungleSem} L \approx \text{Sem}\ jungleSem R \rightarrow \text{jungleSem} A \approx \text{Sem}\ jungleSem B
\]
Proof sketch:

Let us now assume a semantics to be chosen, that is, some gs-monoidal category (e.g., Set), one of its objects \( V \) as interpretation of 1, and an appropriate monoidal bifunctor “\( \_ \times \_ \)”. For a jungle \( J : \text{Jungle} \ m \ n \), we denote \([J]_{m,n}\) for its semantics, which is a morphism from \( V^m \) to \( V^n \). In other words, we denote the morphism component of the semantics functor with \([J]_m \) and \([J]_n\); since this is a gs-monoidal functor, we have in particular \([J_1] \odot [J_2] = [J_1] \times [J_2]\), assuming “\( \odot \)” to stand for sequential composition in the semantics category.

As a result of AHBcomp, the context decomposition carries over to the result \( B \) of the original DPO rewrite step, we have:

\[
B = A_1 \odot (I_k \otimes R) \odot A_2
\]

Under the assumption that the rule \( L \longrightarrow G \longrightarrow R \) is semantics preserving, that is, \([L]_{i,j} = [R]_{i,j}\), we therefore easily obtain semantics preservation of the rewrite result:

\[
[A]_{m,n} = [A_1 \odot (I_k \otimes L) \odot A_2]_{m,n}
\]
\[
= [A_1]_{m+i+k} \odot ([I_k]_{k,k} \times [L]_{i,j}) \odot [A_2]_{j+k,n}
\]
\[
= [A_1]_{m+i+k} \odot ([I_k]_{k,k} \times [R]_{i,j}) \odot [A_2]_{j+k,n}
\]
\[
= [A_1 \odot (I_k \otimes R) \odot A_2]_{m,n}
\]
\[
= [B]_{m,n}
\]
Chapter 10

Conclusion and Outlook

We introduced a formalisation of term graphs as ADHGs where the concrete choice of representation of node and edge sets is abstracted to objects in the parameter category. Through such abstraction, the underlying category can be instantiated to not only sets and functions, but also settings other than that of (all) sets and functions, e.g., finite numbers and vector of finite numbers. We also implemented the categorial and gs-monoidal interfaces for ADHGs with executable formalisation and machine-checked proofs of correctness. They lay out the foundation for our study of term graphs rewriting.

We developed a verified term graph decomposition algorithm as the essential part of the functorial semantics for term graphs. With it, we are able to study the semantics preservation of term graph rewriting.

By using the variant of the standard DPO approach, we obtained an easily understandable concept of rule application. With the context decomposition, we have been able to transfer the semantics from the left-hand side to the right-hand side, obviating the need to consider any semantics for general DHGs such as those containing empty areas. Finally, we obtained the semantics preservation theorem which will be used as an important tool in the generation verified code optimisation tools employing rule-based transformation of data-flow graphs, as outlined for example in (Kahl, 2014).

The current decomposition algorithm targets VDHG, due to limitations of the current setting DepObjBase (see Sect. 5.2). That is, we have no means to access the individual edges, nor the inner nodes, because the collection of edges is an object in the underlying category with no further restriction. We are looking forward to find an interface to extend or replace the DepObjBase interface, in order to make a fully abstract proof of the decomposition correctness possible with avoiding the VecGSMonCat instantiation we used in Chapter 7.
We also need to implement some auxiliary functions for term graph rewriting, and then implement the material which is currently postulated in Chapter 9. This includes the proof for the context formulation for our DPO variant which is for the purpose of the semantics preservation.

On the whole, our work will constitute an important building block in fully verified program transformation and code generation systems, for example, as outlined by Kahl (2014).
Bibliography


Index

≈, 18
!, 42
!, 45
!, 47
$, 20
≡, 35

≡, 53
§, 18
§, 45
§, 47
§, 33
§, 33
§, 31
\iota, 20
κ, 20

Δ, 19
∇, 42
∇, 45
∇, 47
\zeta, 32
\zeta, 32
\zeta, 32
⊗, 22
⊗-LeftUnit, 22
⊗-RightUnit, 22
⊗, 45
⊗, 47
⊗m, 22
⊗∞, 22
→, 31
χ, 53
Xo, 45
Xo, 47

x, 53
0-cell, 40
1-cell, 40
2-category, 28, 40
2-cell, 40

Acyclic, 50, 56
ADHG, 27
ADHG, 33
ADHG homomorphism, 37
ADHG homomorphism category, 38
ADHG matching, 39
ADHG matching category, 39
ADHG-GSMonCat, 42
ADHGCategory, 40
ADHGHomCategory, 38
ADHGIso, 38

ADHGMatching, 39
ADHGMonCat, 41
ADHGMonCat, 41
ADHGMonCatGS, 42
ADHGMonCatS, 42
ADHGMor, 37

Agda, 4
AHBcomp, 81

Algebra, 13

Algebraic approach, 13
allFin, 29

Application graph, 69

Arrow, 9

Associative law, 9

Associativity, 22

Bi-functor, 11
Bicategory, 40
C, 29
Calculational style, 58
Cartesian category, 43
Category, 9
Category, 18
Category of ADHG, 40
Choice of coproducts, 21
chosenEdge, 50
CoCone2Univ, 19
Coconut, 49
Context, 79
Context, 79
Coproduct, 19
Coproduct, 20
coproduct, 21
Coproduct\text{-}\text{-}Iso, 21
Curry-Howard correspondence, 4
Dangling condition, 71
decomposeVDHG\text{-}correctness, 61
decomposeVDHG, 50
Decomposition, 49
Dependent functor, 32
Dependent object, 31
Dependent types, 4
DepObj, 31
DepObjBase, 30
DF, 32
DHG, 28
Direct derivation, 13
Directed hypergraph, 8
Discrete term graph, 50
Double pushout, 15
dupADHG, 36
eArity, 29, 34
Edge, 29
EIn, 34
eIn, 29, 35
ELabel, 34
eOut, 29, 34
eOutIso, 35
evalGSMEv\text{-}EQ, 66
Evaluation of expression, 1
exchADHG, 36
F, 30
f-E, 38
f-I, 38
f-N, 37
Fin, 27, 29
 Functor, 9
Garbage, 11
getContext, 80
gIn, 34
Gluing condition, 71
Gluing graph, 15, 69
gOut, 34
Graph transformation rule, 13
GS-monoidal category, 11
GS-Monoidal category of ADHG, 11
GSME, 44, 45
GSME-LocalVDHG3, 62
GSME0, 47
GSME0-GSME, 47
GSME0\text{-}sortedVfromPrimitiveV, 54
GSME0\text{-}sortedVfromSortedV, 53
GSME0\text{\text{To}}\text{GSME}\Sigma\text{-}EQ, 65
GSMEsem, 47
GSMEsem\text{-}VDHG3\text{GSM}, 61
GSMonoidalCategory, 24
HasCoproducts, 21
Hom, 18
Homomorphism, 7
HOPS, 2
Horizontal composition, 40
Host graph, 15, 78
Hyperarc, 8
Hyperedge, 8
Hypergraph, 7