Dependently-Typed Formalisation of Typed Term Graphs

Wolfram Kahl

McMaster University, Hamilton, Ontario, Canada,
`kahl@cas.mcmaster.ca`

We employ the dependently-typed programming language Agda2 to explore formalisation of untyped and typed term graphs directly as set-based graph structures, via the gs-monoidal categories of Corradini and Gadducci, and as nested **let**-expressions using Pouillard and Pottier's NotSoFresh library of variable-binding abstractions.

1 Introduction

The Coconut project [AK09a, AK09b] uses "code graphs" [KAC06], a variant of term graphs in the spirit of "jungles" [HP91, CR93], as intermediate presentation for the generation of highly optimised assembly code. This is currently implemented in Haskell, and we use the Haskell type system in an embedded domain-specific language (EDSL) for creating such code graphs via what appears to be standard Haskell function definitions, with **let**-definitions introducing sharing, and with functions representing assembly-level operations constructing hyperedges [AK09a]. However, since Haskell does not support full dependent typing, the intermediate term graph datatype interface, supporting graph navigation, traversal, and manipulation operations, cannot preserve the connection with the Haskell-level typing of the assembly operations. Therefore, although EDSL-created code graphs are *well-typed by construction, as certified by the type checker*, this does not hold anymore for code graphs that are the result of internal operations. Those internal operations either require separate proof that they preserve well-typedness, or they need to perform run-time checks, at considerable run-time cost.

In addition, our code-graph-creation EDSL has a second "simulator" implementation, which turns the EDSL expressions into Haskell functions that implement a "machine simulation". Since the code graph representation has lost its connection with the Haskell-level typing, it is "unintuitively hard" to use the simulation machinery for code graphs that result from code graph manipulation operations.

Mainly for these reasons, we are now exploring implementation of code graphs in a dependently typed programming language, where there is no need to "loose" the type information when moving to a graph representation, and where even stronger assertions about operations on code graphs than just type preservation can be proven *inside* the implementing system.

We start, in Sect. 2, with a quick introduction to the dependently typed programming language (and proof checker) Agda [Nor07]. This is followed by formalisations of set-based mathematical definitions of untyped (Sect. 3) and typed (Sect. 4) term graphs, and then a summary of the gs-monoidal category view on these term graphs in Sect. 5. Finally, we present two formalisations of acyclic term graphs as (differently structured) nested **let**-expressions (Sections 6 and 7).

2 Introduction to Agda: Types, Sets, Equality

The Agda home page[1] states:

---

[1] `http://wiki.portal.chalmers.se/agda/`

Agda is a dependently typed functional programming language. It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parametrised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.

Agda is a proof assistant. It is an interactive system for writing and checking proofs. Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

Syntactically and "culturally", Agda is quite close to Haskell. However, since Agda is strongly normalising and has no $\perp$ values, the underlying semantics is quite different. Also, since Agda is dependently typed, it does not have the distinction that Haskell has between terms, types, and kinds (the "types of the types"). The Agda constant Set corresponds to the Haskell kind *; it is the type of all "normal" datatypes. For example, the Agda standard library defines the type Bool as follows:

data Bool : Set **where** true : Bool
false : Bool

Since Set needs again a type, there is Set_1, with Set : Set_1, etc., resulting in a hierarchy of "universes". Since version 2.2.8, Agda supports *universe polymorphism*, with universes Set i where i is an element of the following special-purpose variant of the natural numbers:

data Level : Set **where** zero : Level
suc : (i : Level) → Level

With this, the conventional usage turns into syntactic sugar, so that Set is now Set zero, and $Set_1 =$ Set (suc zero). For example, the standard library includes the following universe-polymorphic definition for the parameterised Maybe type:

data Maybe {a : Level} (A : Set a) : Set a **where** just : (x : A) → Maybe A
nothing : Maybe A

Maybe has two parameters, a and A, where dependent typing is used since the type of the second parameter depends on the first parameter. The use of {...} flags a as an *implicit parameter* that can be elided where its type is implied by the call site of Maybe. This happens in the occurrences of Maybe A in the types of the data constructors just and nothing: In Maybe A, the value of the first, implicit parameter of Maybe can only be a, the level of the set A.

The same applies to implicit function arguments, and in most cases, implicit arguments or parameters are determined by later arguments respectively parameters. Frequently, implicit arguments correspond quite precisely to that part of the context of mathematical statements that is frequently left implicit by mathematicians, so that the reader may be advised to skip implicit arguments at first reading of a type, and return to them for clarification where necessary for understanding the types of the explicit parameters.

While the Hindley-Milner typing of Haskell and ML allows function definitions without declaration of the function type, and type signatures without declaration of the universally quantified type variables, in Agda, almost all types and variables need to be declared, but implicit parameters and the type checking machinery used to resolve them alleviate that burden significantly. For example, the original definition writes only Maybe {a} (A : Set a) : Set a, since the type of a will be inferred from a's use as argument to Set.

The "programming types" like Maybe can be freely mixed with "formula types", inspired by the Curry-Howard-correspondence of "formulae as types, proofs as terms". The formula types of true formulae contain their proofs, while the formula types of false formulae are empty.

The standard library type of propositional equality has (besides two implicit parameters) one explicit parameter and one explicit argument; the definition therefore gives rise to types like the type "$2 \equiv 1 + 1$", which can be shown to be inhabited using the definition of natural numbers 1 and 2 and natural number addition $+$, and the type "$2 \equiv 3$", which is an empty type, since it has no proof.

```
data _≡_ {a : Level} {A : Set a} (x : A) : A → Set a where refl : x ≡ x
```

The underscore characters occurring in the name $\_\equiv\_$ declare mixfix syntax with argument positions for explicit parameters and arguments; this mixfix syntax is already used in the type of the single constructor. The definition introduces types $x \equiv y$ for any x and y of type A, but only the types $x \equiv x$ are inhabited, and they contain the single element refl $\{a\}$ $\{A\}$ $\{x\}$.

In Agda, as in other type theories without quotient types, sets with equality are typically modelled as *setoids*, that is, carrier types equipped with an equivalence. This closely corresponds to the non-primitive nature of the "equality" test $(==)$: Eq a $\Rightarrow$ a $\rightarrow$ a $\rightarrow$ Bool in Haskell. A setoid is a dependent record consisting of a Carrier set, a relation $\_\approx\_$ on that carrier, and a proof that the relation $\_\approx\_$ is an equivalence relation:

```
record Setoid c l : Set (suc (c⊔l)) where
    field Carrier : Set c
          _≈_    : Rel Carrier l
          isEquivalence : IsEquivalence _≈_
    open IsEquivalence isEquivalence public
```

An Agda record is also a module that may contain other material besides its **field**s; the "**open**" clause makes the fields of the equivalence proof available as if they were fields of Setoid. This language feature enables incremental extension of smaller theories to larger theories at very low notational cost.

Whenever we allow arbitrary node or edge sets, and we want to prove, for example, isomorphism of certain graphs, we actually need setoids and not just sets. For such contexts, we introduce the following abbreviation for extracting the carrier set from a setoid:

```
⌊_⌋ : {c l : Level} → Setoid c l → Set c
⌊ s ⌋ = Setoid.Carrier s
```

3   Set-Based Term Graphs

We now present a simple definition of term graphs that is intentionally kept close to conventional mathematical formulations. To reduce complexity and improve readability of this initial formalisation, we present untyped term graphs here; a typed variant will be shown in Sect. 4.

In the context of an arity-indexed label type Label : $\mathbb{N} \to$ Set, we first define a type DHG_1 of directed hypergraphs with one putput per edge, indexed by input and output arities of the whole graph, with the following components (since Agda records are also modules, they can contain additional material besides their **field**s):

- A setoid Inner of non-input nodes. (For simplicity, we do not emply universe polymorphism here, and all our setoids are of type Setoid zero zero.)
  For technical reasons, we find it more convenient to have the non-input nodes separate from the input nodes. Otherwise we would have had to include an explicit injection from the input positions to the complete node set.
- The setoid Node of all nodes is then derived as the disjoint union of Inner with the setoid of input positions, which is obtained from Fin m, the set of natural numbers smaller than m.
- The second **field** is the n-element vector of output nodes, which can be either input nodes or inner nodes.
- For symmetry, we also provide the m-element vector of input nodes, constructed using allFin m which is the vector (i.e., array) containing all m elements of the set Fin m in sequence, i.e., 0, 1, . . . , m - 1.
- Edge is the setoid of hyperedges.
- eInfo maps each edge to a dependent tuple consisting of an arity k, a k-ary label, and a k-element vector of edge input nodes.
- eOut maps each edge to its output node, which cannot be an input node of the Jungle, and therefore has to be an Inner node. (The function arrow between setoids is optically not distinguishable from the general function type arrow, but is technically a different symbol. Since setoids cannot be used as types, no confusion can arise.)
- We derive the function eLabel that maps each edge e to its edge label. Since the arity of that label is not known in advance, the function eLabel returns a dependent pair consisting of the label arity k and a k-ary label.
- We also derive the function eIn that maps each edge e to the vector of input nodes of e; the type of this vector depends on the arity of e, which is the first component (proj_1) of the dependent tuple eLabel e.

```
record DHG₁ (m n : ℕ) : Set₁ where
  field Inner : Setoid zero zero
  Node = Fin.setoid m ⊎⊎ Inner
  field output : Vec ⌊ Node ⌋ n

  input : Vec ⌊ Node ⌋ m
  input = Vec.map inj₁ (allFin m)

  field Edge : Setoid zero zero
    eInfo : ⌊ Edge ⌋
        → Σ [k : ℕ] (Label k × Vec ⌊ Node ⌋ k)


    eOut : Edge → Inner




  eLabel : ⌊ Edge ⌋ → Σ [k : ℕ] Label k
  eLabel e = Product.map id proj₁ (eInfo e)
  eIn : (e : ⌊ Edge ⌋) → Vec ⌊ Node ⌋ (proj₁ (eLabel e))
  eIn = proj₂ ∘ proj₂ ∘ eInfo
```

```
record Jungle (m n : ℕ) : Set₁ where
  field Inner : Setoid zero zero
  Node = Fin.setoid m ⊎⊎ Inner
  field output : Vec ⌊ Node ⌋ n

  input : Vec ⌊ Node ⌋ m
  input = Vec.map inj₁ (allFin m)

  field Edge : Setoid zero zero
    eInfo : ⌊ Edge ⌋
        → Σ [k : ℕ] (Label k × Vec ⌊ Node ⌋ k)
    EOut : Inverse Edge Inner
  eOut : Edge → Inner
  eOut = Inverse.to EOut
  producer : Inner → Edge
  producer = Inverse.from EOut

  eLabel : ⌊ Edge ⌋ → Σ [k : ℕ] Label k
  eLabel e = Product.map id proj₁ (eInfo e)
  eIn : (e : ⌊ Edge ⌋) → Vec ⌊ Node ⌋ (proj₁ (eLabel e))
  eIn = proj₂ ∘ proj₂ ∘ eInfo
```

In this DHG_1 definition, eOut does not have to be surjective, which means that there may be "undefined nodes", and eOut also does not have to be injective, which means that there may be "join nodes" in the sense of [KAC06]. If bijectivity of eOut is desired, we can replace the setoid mapping with an inverse pair of mappings, and extract eOut and the producer mapping for inner nodes from that, as shown above to the right.

These jungles are isomorphic to conventional termgraphs, where inputs (as arguments) and labels are attached directly to inner nodes:

```
record TermGraph (m n : ℕ) : Set₁ where
   field  Inner : Setoid zero zero
   Node = Fin.setoid m ⊎⊎ Inner
   field  output : Vec ⌊ Node ⌋ n
   input : Vec ⌊ Node ⌋ m
   input = Vec.map inj₁ (allFin m)
   field  label : ⌊ Inner ⌋ → Σ [ k : ℕ ] Label k
          args : (n : ⌊ Inner ⌋) → Vec ⌊ Node ⌋ (proj₁ (label n))
```

The following basic constructor functions are highly similar for DHG_1, Jungle, and TermGraph; we show them here for Jungle.

Using the one-element setoid ⊤ (with element tt), we can define primitive jungles consisting of a single hyperedge:

```
prim : {k : ℕ} → Label k → Jungle k 1
prim {k} f = record
  {Inner   = ⊤
  ; output = [inj₂ tt]
  ; Edge   = ⊤
  ; eInfo  = λ _ → (k, (f, Vec.map inj₁ (allFin k)))
  ; EOut   = Inverse.id
  }
```

For wiring graphs, we need empty sets (⊥) of edges and inner nodes:

```
wire : {m n : ℕ} → Vec (Fin m) n → Jungle m n
wire {m} {n} v = record
  {Inner   = ⊥
  ; output = Vec.map inj₁ v
  ; Edge   = ⊥
  ; eInfo  = E.⊥-elim
  ; EOut   = Inverse.id
  }
```

With this, we can easily construct the standard wiring graphs required for defining a gs-monoidal category (see Sect. 5) of Jungles:

```
idJungle : {m : ℕ} → Jungle m m
idJungle = wire (allFin _)

dupJungle : {m : ℕ} → Jungle m (m + m)
dupJungle {m} = wire (allFin m ⁺ allFin m)

termJungle : {m : ℕ} → Jungle m 0
termJungle = wire []

exchJungle : (m n : ℕ) → Jungle (m + n) (n + m)
exchJungle m n = wire (Vec.map (raise m) (allFin n) ⁺ Vec.map (inject+ n) (allFin m))
```

Separating the inner nodes from the inputs in particular has the advantage that for sequential composition, we can just use the disjoint union of the two Inner node sets:

seqJungle : $\{k\ m\ n : \mathbb{N}\} \rightarrow$ Jungle k m $\rightarrow$ Jungle m n $\rightarrow$ Jungle k n
seqJungle $\{k\}\ \{m\}\ \{n\}\ g_1\ g_2 =$ **let**
    open Jungle
    $h_1 : \lfloor$ Node $g_1 \rfloor \rightarrow$ Fin k $\uplus (\lfloor$ Inner $g_1 \rfloor \uplus \lfloor$ Inner $g_2 \rfloor)$
    $h_1 =$ Sum.map id inj_1
    $h_2 : \lfloor$ Node $g_2 \rfloor \rightarrow$ Fin k $\uplus (\lfloor$ Inner $g_1 \rfloor \uplus \lfloor$ Inner $g_2 \rfloor)$
    $h_2 = [(\lambda\ i \rightarrow h_1\ (\text{Vec.lookup}\ i\ (\text{output}\ g_1))), \text{inj}_2 \circ \text{inj}_2]'$
  in record
    $\{$Inner $=$ Inner $g_1 \uplus\uplus$ Inner g_2
    ; output $=$ Vec.map h_2 (output g_2)
    ; Edge $=$ Edge $g_1 \uplus\uplus$ Edge g_2
    ; eInfo $= [\text{productMap}_{22}\ (\text{Vec.map}\ h_1) \circ \text{eInfo}\ g_1, \text{productMap}_{22}\ (\text{Vec.map}\ h_2) \circ \text{eInfo}\ g_2]'$
    ; EOut $=$ EOut $g_1 \oplus\oplus$ EOut g_2
    $\}$

Parallel composition works similarly; here the input positions need to be adapted.

parJungle : $\{m_1\ n_1\ m_2\ n_2 : \mathbb{N}\} \rightarrow$ Jungle $m_1\ n_1 \rightarrow$ Jungle $m_2\ n_2 \rightarrow$ Jungle $(m_1 + m_2)\ (n_1 + n_2)$
parJungle $\{m_1\}\ \{n_1\}\ \{m_2\}\ \{n_2\}\ g_1\ g_2 =$ **let**
    open Jungle
    $h_1 : \lfloor$ Node $g_1 \rfloor \rightarrow$ Fin $(m_1 + m_2) \uplus (\lfloor$ Inner $g_1 \rfloor \uplus \lfloor$ Inner $g_2 \rfloor)$
    $h_1 =$ Sum.map (inject+ m_2) inj_1
    $h_2 : \lfloor$ Node $g_2 \rfloor \rightarrow$ Fin $(m_1 + m_2) \uplus (\lfloor$ Inner $g_1 \rfloor \uplus \lfloor$ Inner $g_2 \rfloor)$
    $h_2 =$ Sum.map (raise m_1) inj_2
  in record
    $\{$Inner $=$ Inner $g_1 \uplus\uplus$ Inner g_2
    ; output $=$ Vec.map h_1 (output g_1) $^+$ Vec.map h_2 (output g_2)
    ; Edge $=$ Edge $g_1 \uplus\uplus$ Edge g_2
    ; eInfo $= [\text{productMap}_{22}\ (\text{Vec.map}\ h_1) \circ \text{eInfo}\ g_1, \text{productMap}_{22}\ (\text{Vec.map}\ h_2) \circ \text{eInfo}\ g_2]'$
    ; EOut $=$ EOut $g_1 \oplus\oplus$ EOut g_2
    $\}$

4 Typed Code Graphs

Coconut code graphs [KAC06] have types associated with nodes, and hyperedges may have not only multiple inputs, but also multiple outputs, to be able to model operations that yield multiple results; the typing of the input and output nodes needs to be compatible with the operations indicated by the edge labels.

For simplicity, we assume here a global set Type : Set of node types, and dispense with using setoids in this section. An edge label is now indexed by vectors of input and output types, so we assume Label : $\{m\ n : \mathbb{N}\} \rightarrow$ Vec Type m $\rightarrow$ Vec Type n $\rightarrow$ Set, and also define the dependent record type EdgeType for collecting these indices:

record EdgeType : Set **where**
  field inArity : $\mathbb{N}$
          outArity : $\mathbb{N}$
          inTypes : Vec Type inArity
          outTypes : Vec Type outArity

An edge label then is such an index collection together with a label drawn from the corresponding label set; the **open** declaration makes the EdgeType fields available for EdgeLabel elements as if this was a record extension:

```
record EdgeLabel : Set where
  field  eType : EdgeType
         label : Label (EdgeType.inTypes eType) (EdgeType.outTypes eType)
  open EdgeType eType public
```

For typed term graphs, there are many different ways to deal with node typing, and for any given way, different views are useful in different contexts. We will keep a node typing function as a **field**, and derive from this an indexed view of typed nodes, using the following general construct: Given a set A and a typing function type for A, the Type-indexed set Typed A type associates with every type ty all elements of A that have type ty; formally, an element of Typed A type ty is a dependent pair consisting of an element a : A together with a proof that type a $\equiv$ ty:

```
Typed : (A : Set) → (A → Type) → Type → Set
Typed A type ty = Σ [a : A] (type a ≡ ty)
```

Since the Agda standard library does not provide a variant of Vec where the element types may depend on their positions, we directly use dependently typed functions starting from these positions instead, producing "typed vectors" with elements type according to the argument type vector v:

```
TypedVec : (A : Set) → (A → Type) → {k : ℕ} → Vec Type k → Set
TypedVec A type {k} v = (i : Fin k) → Typed A type (Vec.lookup i v)
```

The EdgeInfo associated with each hyperedge then contains, besides an EdgeLabel, two such "typed node vectors", typed according to the label's typing information (for modularity, this definition is kept outside the code graph definition and parameterised with the type Nodes for "typed node vectors" to be supplied there):

```
record EdgeInfo (Nodes : {k : ℕ} → Vec Type k → Set) : Set where
  field  eLab    : EdgeLabel
         eInput  : Nodes (EdgeLabel.inTypes eLab)
         eOutput : Nodes (EdgeLabel.outTypes eLab)
  open EdgeLabel eLab public
```

A CodeGraph is now defined roughly analogous to a Jungle, with the following differences worth pointing out:
- Code graphs can be considered as "generalised hyperedges", and therefore have an EdgeType derived from the CodeGraph type parameters. Keeping the current parameters eases the implementation of the categorical view, in comparison with using the EdgeType as a parameter instead.
- We only need to explicitly represent the typing of the inner nodes; from this we can derive the typing of all Nodes by looking up the typing of the input positions in inTypes.
- A TypedNode ty is a Node with type ty; an element of TypedNodes v is a "typed node vector" according to the type vector v.
- The CodeGraph field output and each individual edge interface use TypedNode "vectors".
- We can still provide lower-level interfaces to edges; we show functions that extract the edge label, edge input arity, and edge input Node vectors (discarding the type information), both dependently-typed and existentially-typed with respect to the vector length. (The corresponding functions eOut etc.are not shown.)

```
record CodeGraph {m n : ℕ} (inTypes : Vec Type m) (outTypes : Vec Type n) : Set₁ where
  cgType : EdgeType
  cgType = record {inArity   = m
                  ; outArity  = n
                  ; inTypes   = inTypes
                  ; outTypes = outTypes}

  field Inner : Set
        iType : Inner → Type

  Node = Fin m ⊎ Inner

  nType : Node → Type
  nType = [(λ i → Vec.lookup i inTypes), iType] ′

  TypedNode : Type → Set
  TypedNode = Typed Node nType

  TypedNodes : {k : ℕ} → Vec Type k → Set
  TypedNodes = TypedVec Node nType

  field output : TypedNodes outTypes

  input : TypedNodes inTypes
  input = λ i → (inj₁ i, refl)

  field Edge : Set
        eInfo : Edge → EdgeInfo TypedNodes

  eLabel : Edge → EdgeLabel
  eLabel = EdgeInfo.eLab ∘ eInfo

  eInArity : Edge → ℕ
  eInArity = EdgeInfo.inArity ∘ eInfo

  eIn : (e : Edge) → Vec Node (eInArity e)
  eIn e = mkVec (proj₁ ∘ EdgeInfo.eInput (eInfo e))

  eIn′ : Edge → Σ [k : ℕ] (Vec Node k)
  eIn′ e = eInArity e, eIn e
```

Again, eOut is not guaranteed to reach all nodes, and, due to the possibility of multi-output operations, this cannot be amended by joining the Inner and Edge sets as in jungles. This and other degrees of generality contained in this definition can be useful for certain purposes, but also can be forbidden for other purposes by adding appropriate constraints.

We show the function for producing primitive one-edge code graphs:

```
prim : (l : EdgeLabel) → CodeGraph (EdgeLabel.inTypes l) (EdgeLabel.outTypes l)
prim l = record
  {Inner   = Fin (EdgeLabel.outArity l)
  ; output = λ i → (inj₂ i, refl)
  ; Edge   = ⊤
  ; eInfo  = λ _ → record {eLab     = l
                          ; eInput  = λ i → (inj₁ i, refl)
                          ; eOutput = λ i → (inj₂ i, refl)}}
```

While type-checking the three propositional equality proofs refl in here, Agda actually proves that the mentioned types are indeed equal: An Agda program can only produce CodeGraph values that are correctly typed, both on the external interface, and internally at each port of each edge.

5   GS-Monoidal Categories

Corradini and Gadducci proposed *gs-monoidal categories* for modelling acyclic term graphs [CG99]; extended discussion of how code graphs fit into this framework is contained in [KAC06]. Here we only present a quick summary, and tie this into the formalisation in Sect. 3.

In a category theory context, we write "$f : \mathscr{A} \to \mathscr{B}$" to declare that morphism f goes from object $\mathscr{A}$ to object $\mathscr{B}$, and use "⨾" as the associative binary *composition* operator; composition of two morphisms $f : \mathscr{A} \to \mathscr{B}$ and $g : \mathscr{B}' \to \mathscr{C}$ is defined iff $\mathscr{B} = \mathscr{B}'$, and then $(f \,⨾ g) : \mathscr{A} \to \mathscr{C}$. Furthermore, the identity morphism for object $\mathscr{A}$ is written $\mathbb{I}_{\mathscr{A}}$.

Jungle can be seen to define morphisms of an untyped term graph category where objects are natural numbers. (For CodeGraph, the collection of Objects is $\Sigma\,[\mathsf{k} : \mathbb{N}]\,(\mathsf{Vec\ Type\ k})$.)

In the Jungle category, a morphism from *m* to *n* is an element of Jungle m n, that is, a term graph with *m* input nodes and *n* output nodes. More precisely, such a morphism is an isomorphism class of jungles, since node and edge identities do not matter; we will define a Setoid where the Carrier is Jungle m n and equivalence proofs are Jungle isomorphisms.

Composition $F \,⨾ G$ "glues" together the output nodes of F with the respective input nodes of G, as we have implemented in seqJungle. The identity on *n* consists only of *n* input nodes which are also, in the same sequence, output nodes, and no edges, and is therefore constructed as a wiring graph:

```
idJungle : {m : ℕ} → Jungle m m
idJungle = wire (allFin _)
```

Definition 5.1 A *symmetric strict monoidal category* [ML71] consists of a category $\mathbf{C}_0$, a strictly associative monoidal bifunctor $\otimes$ with $\mathbb{1}$ as its strict unit, and a transformation $\mathbb{X}$ that associates with every two objects $\mathscr{A}$ and $\mathscr{B}$ an arrow $\mathbb{X}_{\mathscr{A},\mathscr{B}} : \mathscr{A} \otimes \mathscr{B} \to \mathscr{B} \otimes \mathscr{A}$ with:

$$
\begin{aligned}
(F \otimes G) \,⨾ \mathbb{X}_{\mathscr{C},\mathscr{D}} &= \mathbb{X}_{\mathscr{A},\mathscr{B}} \,⨾ (G \otimes F) \ , & \mathbb{X}_{\mathscr{A},\mathscr{B}} \,⨾ \mathbb{X}_{\mathscr{B},\mathscr{A}} &= \mathbb{I}_{\mathscr{A}} \otimes \mathbb{I}_{\mathscr{B}} \ , \\
\mathbb{X}_{\mathscr{A} \otimes \mathscr{B},\mathscr{C}} &= (\mathbb{I}_{\mathscr{A}} \otimes \mathbb{X}_{\mathscr{B},\mathscr{C}}) \,⨾ (\mathbb{X}_{\mathscr{A},\mathscr{C}} \otimes \mathbb{I}_{\mathscr{B}}) \ , & \mathbb{X}_{\mathbb{1},\mathbb{1}} &= \mathbb{I}_{\mathbb{1}} \ . \qquad \square
\end{aligned}
$$

For Jungle, the unit object $\mathbb{1}$ is the natural number 0, and $\otimes$ on objects is addition. On morphisms, $\otimes$ forms the disjoint union of code graphs, concatenating the input and output node sequences, as implemented in parJungle. $\mathbb{X}_{m,n}$ differs from $\mathbb{I}_{m+n}$ only in the fact that the two parts of the output node sequence are swapped:

```
exchJungle : (m n : ℕ) → Jungle (m + n) (n + m)
exchJungle m n = wire (Vec.map (raise m) (allFin n) ⁺ Vec.map (inject+ n) (allFin m))
```

Definition 5.2 A *strict gs-monoidal category* is a symmetric strict monoidal category where in addition ! associates with every object $\mathscr{A}$ of $\mathbf{C}_0$ an arrow $!_{\mathscr{A}} : \mathscr{A} \to \mathbb{1}$, and ∇ associates with every object $\mathscr{A}$ of $\mathbf{C}_0$ an arrow $\nabla_{\mathscr{A}} : \mathscr{A} \to \mathscr{A} \otimes \mathscr{A}$, such that $\mathbb{I}_{\mathbb{1}} = !_{\mathbb{1}} = \nabla_{\mathbb{1}}$, and the following axioms hold:

$$
\begin{aligned}
\nabla_{\mathscr{A}} \,⨾ (\mathbb{I}_{\mathscr{A}} \otimes \nabla_{\mathscr{A}}) = \nabla_{\mathscr{A}} \,⨾ (\nabla_{\mathscr{A}} \otimes \mathbb{I}_{\mathscr{A}}) \qquad & \nabla_{\mathscr{A}} \,⨾ \mathbb{X}_{\mathscr{A},\mathscr{A}} = \nabla_{\mathscr{A}} \qquad && \nabla_{\mathscr{A}} \,⨾ (\mathbb{I}_{\mathscr{A}} \otimes !_{\mathscr{A}}) = \mathbb{I}_{\mathscr{A}} \\
\nabla_{\mathscr{A} \otimes \mathscr{B}} \,⨾ (\mathbb{I}_{\mathscr{A}} \otimes \mathbb{X}_{\mathscr{B},\mathscr{A}} \otimes \mathbb{I}_{\mathscr{B}}) = \nabla_{\mathscr{A}} \otimes \nabla_{\mathscr{B}} \qquad & !_{\mathscr{A} \otimes \mathscr{B}} = !_{\mathscr{A}} \otimes !_{\mathscr{B}} && \qquad \square
\end{aligned}
$$

In Jungle, the "terminator" $!_n$ differs from $\mathbb{I}_n$ only in the fact that the output node sequence is empty.

```
termJungle : {n : ℕ} → Jungle n 0
termJungle = wire [ ]
```

The "g" of "gs-monoidal" stands for "garbage": all edges of a term graph $G : m \rightarrow n$ are garbage in the term graph $G \text{;} !_n$.

The duplicator ∇_n in Jungle differs from $\mathbb{I}_n$ only in the fact that the output node sequence is the concatenation of the input node sequence with itself:

```
dupJungle : {n : ℕ} → Jungle n (n + n)
dupJungle {n} = wire (allFin n ⁺ allFin n)
```

The "s" of "gs-monoidal" stands for "sharing": every input of $\nabla_k \text{;} (F \otimes G)$ is shared by $F : k \rightarrow m$ and $G : k \rightarrow n$.

Code graphs (and term graphs) over a fixed edge label set form a gs-monoidal category, but not a *Cartesian* category, where in addition ! and ∇ are *natural* transformations, i.e., for all $F : \mathscr{A} \rightarrow \mathscr{B}$ we have $F \text{;} !_{\mathscr{B}} = !_{\mathscr{A}}$ and $F \text{;} \nabla_{\mathscr{B}} = \nabla_{\mathscr{A}} \text{;} (F \otimes F)$. To see how these naturality conditions are violated by term graphs, the following five Jungles correspond to the expressions below them (we draw jungles and code graphs from the inputs on top to the outputs at the bottom, with numbered triangles marking input and output positions, and rectangles enclosing edge labels).

$$F : 1 \rightarrow 1 \qquad !_1 \qquad F \text{;} !_1 \qquad F \text{;} \nabla_1 \qquad \nabla_1 \text{;} (F \otimes F)$$
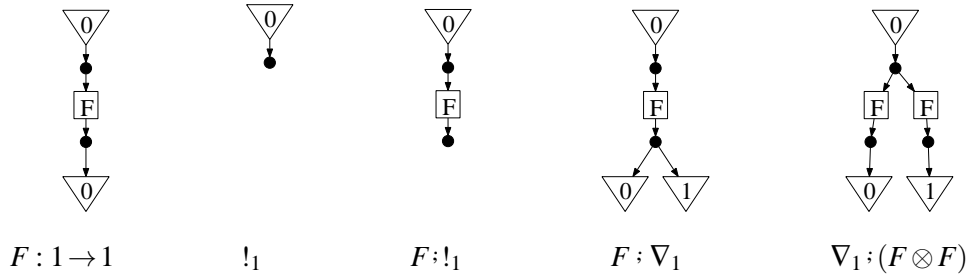
Formalising (symmetric gs-) monoidal categories in Agda is a straight-forward extension of the standard type-theoretic formalisation of category theory deriving essentially from Kanda's "effective categories" [Kan81]; this uses setoids of morphisms, but not of objects. This approach is also used by Huet and Saïbi [HS98, HS00] for their formalisation of category theory in Coq, and by Gonzalía [Gon06] for his formalisation of Freyd and Scedrov's allegory hierarchy [FS90] in Alf, a predecessor of Agda.

This approach also corresponds to the general practice in category theory to consider objects only up to isomorphism, not up to equality. However, the definition of strict monoidal categories runs counter to this approach, by assuming an object-level operation ($\otimes$) satisfying non-trivial object-level equations. Therefore we directly formalise what MacLane calls "relaxed" monoidal categories, with natural isomorphisms $\alpha : \mathscr{A} \otimes (\mathscr{B} \otimes \mathscr{C}) \rightarrow (\mathscr{A} \otimes \mathscr{B}) \otimes \mathscr{C}$ and $\lambda : \mathbb{1} \otimes \mathscr{A} \rightarrow \mathscr{A}$ and $\rho : \mathscr{A} \otimes \mathbb{1} \rightarrow \mathscr{A}$.

This explicit approach also has advantages for moving between different levels of data nesting without requiring additional features; this is important for example for reasoning about the effect of SIMD operations together with SIMD vector manipulations on individual scalar values, which is necessary for verifying numerous high-performance "tricks", see e.g.[AK08].

6   Term Graphs as Let Constructs

The code graph representation of Sect. 4 essentially is a typed variant of the current internal representation of Coconut code graphs, but, as mentioned in the introduction, we essentially write Haskell definitions to initially create code graphs.

In lazy functional programming implemented by graph reduction, since at least KRC [Tur82], local definitions (via **let** or **where**) are understood to introduce *sharing*. In a mathematical context, [AK94] represents cyclic term graphs as systems of mutually recursive equations, and [MOW98] presents sharing in the call-by-need λ-calculus via **let**-expressions.

In the following, we present two formalisations of term graphs defined by non-recursive nested **let**-expressions. For the sake of readability, we restrict ourselves to untyped term graphs and single-output primitives.

With **let**-expressions, we automatically have to deal with the complications of bound variables, involving scoping, renaming to avoid variable clashes, etc. The Agda library NotSoFresh by Pouillard and Pottier [PP10] allows us to abstract from these concerns to a large degree, at the cost of following the discipline of their World-based programming interface. At the core of their approach, there are Worlds in which different variables are in scope; for a world α, the set of usable names is Name α. Introducing a new name happens via a "world extension link"; an element of $\alpha \leftarrow \beta$ is a *weak link* that provides a variable in β that might be shadowing one of the variables in α, while an element of $\alpha \leftarrow\!\!\rightarrow \beta$ is a *strong link* that provides, in β, a variable that is *fresh* with respect to all variables in α.

For programming and in mathematics, we are used to working in a context of weak links, while symbol manipulation systems, including theorem provers and compilers, frequently disambiguate names so that they can work with strong links exclusively. To enable both settings, we will parameterise over these "world Extension relation" with a parameter E : World $\rightarrow$ World $\rightarrow$ Set.

We first present the type TG that formalises **let**-expressions with arbitrary nesting; this type is only a slight modification of the λ-term datatype Tm from [PP10].

A value of type TG E α m n is, in the context of m input nodes and of a world α providing already existing inner nodes, a term graph "suffix" producing n output nodes:

- The input node at position i can be produced as an output node by Input i.
- An existing node x : Name α is produced as an output node by V x.
- The empty suffix is called ε.
- Given two suffixes t and u of output lengths n_1 and n_2, their union, with concatenated output lists, is t $\triangledown$ u. The symbol $\triangledown$ reads "fork", as in the fork algebras of [HFBV97]; it is related with the duplicator ∇ via the equation t $\triangledown$ u $= \nabla_m \,\dot{}\, (t \otimes u)$.
- A primitive f can only be invoked while applying it to the outputs of a term graph suffix t and while at the same time creating a new node x in an expression of the shape Let x f t u, which, in more conventional notation, would read "**let** x $=$ f (t) **in** u".

  If the primitive f expects k inputs, the argument term graph suffix t, which may not use the new name x because it is in the "old" world α, has to have k outputs.

  The term graph suffix u may use also the new name x, and its outputs will be the outputs of the "Let x f t u" expression.

```
data TG (E : World → World → Set) (α : World) (m : ℕ) : ℕ → Set where
  Input : (i : Fin m)    → TG E α m 1
  V     : (x : Name α) → TG E α m 1
  ε     : TG E α m 0
  _▽_   : {n₁ n₂ : ℕ} → TG E α m n₁ → TG E α m n₂ → TG E α m (n₁ + n₂)
  Let   : {β : World} {k n : ℕ}
        → (x : E α β)                    -- let x
        → (f : Label k) → (t : TG E α m k)   -- = f (t)
        → (u : TG E β m n)               -- in u
        → TG E α m n
```

Without additional support, defining term graphs using this interface is somewhat inconvenient — the following assumes a unary label F, a binary label G, and a ternary label H:

```
TG0 : Label 1 → Label 2 → Label 3 → TG _ ←_ ∅ 3 1
TG0 F G H  =  let f0  =  fresh∅              -- a strong link
                  x0  =  FreshPack.weakOf f0    -- weak view of f0
                  n0  =  FreshPack.nameOf f0   -- Name of f0
   in Let x0 H
     (Let x0 F (Input zero) (V n0 ▽ V n0)
        ▽
        Let x0 G (Input (suc zero) ▽ Input (suc (suc zero))) (V n0)
     )
     (V n0)
```
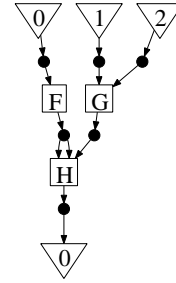

Using slightly more conventional notation, this corresponds to the following, relatively readable version, with "i" prefixing inputs and "n" prefixing node names:

```
let n0  =  H ((let n0  =  F (i0) in (n0 ▽ n0))
                ▽
                (let n0  =  G (i1 ▽ i2) in n0)
             ) in n0
```

Either by adding more notational support, or by defining a separate input language, this can provide an interface that comes reasonably close to Haskell-style programming.

The real point of the definition of TG however is that it not only provides an input language, but also a representation of term graphs that can be manipulated and transformed by programs. For example, we can turn a TG with name shadowing (i.e., using weak links) into one with strong links by replacing all node names with fresh names relative to their respective worlds:

```
strengthenTG : {α α′ : World} → Fresh α′ → CEnv (Name α′) α
              → {m n : ℕ} → TG _ ←_ α m n → TG _ ←→_ α′ m n
strengthenTG _ _ ε          = ε
strengthenTG fr Γ (t ▽ u)  =  (strengthenTG fr Γ t) ▽ (strengthenTG fr Γ u)
strengthenTG _ _ (Input i)  =  Input i
strengthenTG fr Γ (V x)     =  V (lookupCEnv Γ x)
strengthenTG fr Γ (Let x f t u)
   = let Γ′  =  mapCEnv importWith Γ, x ↦ nameOf
      in Let strongOf f (strengthenTG fr Γ t) (strengthenTG nextOf Γ′ u)
      where open FreshPack fr
```

Parallel composition is also easy to program, using fork after embedding, respectively shifting, the inputs:

```
parTG : {E : _} {α : _} {m₁ n₁ m₂ n₂ : ℕ}
      → TG E α m₁ n₁ → TG E α m₂ n₂ → TG E α (m₁ + m₂) (n₁ + n₂)
parTG {E} {α} {m₁} {n₁} g₁ {m₂} {n₂} g₂  =  extendTG m₂ g₁ ▽ shiftTG m₁ g₂
```

Sequential composition is much harder to implement directly, since the output nodes of the first argument may have been defined in separate worlds and combined with fork, and now need to be brought into a common world, which in general requires renaming and restructuring. A convenient "canonical form" for such **let**-expressions has no Let at argument positions, and no Let below fork, and therefore degenerates into a sequence of Let declarations each binding a new node to the application of some primitive

to existing nodes. When dealing with any kind of canonical forms, especially in a dependently-typed setting, it is frequently worth while declaring this as a separate datatype so that it becomes easier to exploit its properties. For this canonical form of TG, we introduce a separate datatype with additional restructuring below.

7   Term Graphs with Sequential Node Declaration

According to our explanation of TG term graphs, ∇ with ε obviously forms a monoid, but the monoid laws do not come for free in TG. Moving to the Vec container type instead provides us with the monoid laws in the standard library, and makes for a more canonical representation. With this change, and with strictly linearised node declaration, the term graph TG0 shown above could be written in a somewhat conventional notation as follows (without fully specifying the number of inputs):

```
let n0 = F i0
let n1 = G i1 i2
let n2 = H n0 n0 n1
in [n2]
```

We introduce the type Arg for individual nodes, either existing inner nodes, or input positions, and a type synonym Args for their vectors:

```
data Arg α (m : ℕ) : Set where
   Input : (i : Fin m) → Arg α m
   V     : (x : Name α) → Arg α m
Args α m n = Vec (Arg α m) n
```

The datatype TG′ has the same reading as TG, but a simpler structure:
• If all nodes have been declared, Output as assembles the vector of output nodes.
• Let x f v u, which, in more conventional notation, would read "**let** x = f (v) **in** u", binds a new node x to an edge labelled f with input nodes v, and makes x visible in the remaining term graph suffix u.

```
data TG′ E α (m : ℕ) : ℕ → Set where
   Output : {n : ℕ} → Args α m n → TG′ E α m n
   Let    : {β : World} {k n : ℕ}
        → (x : E α β)                -- let x
        → (f : Label k) (v : Args α m k)   -- = f (v)
        → (u : TG′ E β m n)              -- in u
        → TG′ E α m n
```

We first show that primitive and wiring graphs are easily programmed:

```
prim : {k : ℕ} → Label k → TG′ _←→_ ∅ k 1
prim {k} f = Let strongOf f (Vec.map Input (Vec.allFin k)) (Output [V nameOf])
   where open FreshPack fresh∅
wire : {k n : ℕ} {E : _} {α : World} → Vec (Fin k) n → TG′ E α k n
wire v = Output (Vec.map Input v)
idWire : {k : ℕ} {E : _} {α : World} → TG′ E α k k
idWire {k} = wire (Vec.allFin k)
dup : {k : ℕ} {E : _} {α : World} → TG′ E α k (k + k)
```

```
dup {k} = wire (Vec.allFin k + Vec.allFin k)
term : {k : ℕ} {E : _} {α : World} → TG' E α k 0
term = wire []
```

With these definitions, we can reconstruct the term graph TG0 from above via the gs-monoidal interface, with sequential composition seqTG' and parallel composition parTG' defined below:

```
tg0 = seqTG' (parTG' (seqTG' (prim F) dup) (prim G)) (prim H)
```

For the analogous function to strengthenTG, which replaces each link x in a Let construct with a fresh link, we present an easy generalisation to serve dual purposes:

- Starting from weak links, strengthenTG' { _ ←_ } id is proper strengthening;
- starting from strong links, strengthenTG' { _ ←→_ } StrongPack.weakOf is renaming with fresh names with respect to the new world α'.

```
strengthenTG' : {E : _} → (E ⇒ _ ←_ )
                → {α α' : World} → Fresh α' → CEnv (Name α') α
                → {m n : ℕ} → TG' E α m n → TG' _ ←→_ α' m n
strengthenTG' weak fr Γ (Output as) = Output (mapVarArgs (lookupCEnv Γ) as)
strengthenTG' weak fr Γ (Let x f as u)
   = let Γ' = mapCEnv importWith Γ, weak x ↦ nameOf
      in Let strongOf f (mapVarArgs (lookupCEnv Γ) as) (strengthenTG' weak nextOf Γ' u)
   where open FreshPack fr
```

Both sequential and parallel composition are implemented by inserting the material of one graph between the innermost Let and the Output of the other graph. We define a general helper function for this purpose:

```
inLet' :  {α β : World} → (s : α * ←→ β) → Fresh β → {m n n' : ℕ}
      → ({γ : World} → (s' : α * ←→ γ) → Fresh γ
          → Args γ m n → TG' _ ←→_ γ m n')
      →     TG' _ ←→_ β m n → TG' _ ←→_ β m n'
inLet' s fr F (Let x f t u) = Let x f t (inLet' (s ▷ x) fr' F u) where fr' = StrongPack.nextOf x
inLet' s fr F (Output as) = F s fr as
```

We first implement fork, which walks the only primitively available fresh link freshø past all the Lets of g_1, uses the resulting fresh link fr to rename g_2, and afterwards adapts the output list as_1 of g_1 to the inner world of the renamed g_2, so that the two output lists can be concatenated:

```
forkTG' : {m n₁ n₂ : ℕ}
        → TG' _ ←→_ ø m n₁
        → TG' _ ←→_ ø m n₂
        → TG' _ ←→_ ø m (n₁ + n₂)
forkTG' {m} {n₁} {n₂} g₁ g₂ = inLet' ε freshø
   (λ {γ} s' fr as₁ → inLet' ε fr
      (λ s'' _ as₂ → Output (mapVarArgs (import⊆ (* ←→-⊆ s'')) as₁ + as₂))
      (strengthenTG' { _ ←→_ } StrongPack.weakOf fr emptyCEnv g₂)
   ) g₁
```

The implementation of parallel composition then relies on fork in the same way as that for TG:

```
parTG' : {m₁ n₁ : ℕ}  → TG' _ ←→_ ø m₁ n₁
        → {m₂ n₂ : ℕ} → TG' _ ←→_ ø m₂ n₂
```

$$\to \mathsf{TG}' \; \_ \hookleftarrow \! \to \_ \; \emptyset \, (m_1 + m_2) \, (n_1 + n_2)$$

$$\mathsf{parTG}' \, \{m_1\} \, g_1 \, \{m_2\} \, g_2 \; = \; \mathsf{forkTG}' \, (\mathsf{extendTG}' \, m_2 \, g_1) \, (\mathsf{shiftTG}' \, m_1 \, g_2)$$

Sequential composition follows the same pattern as forkTG′, and first traverses the declarations of g_1, which are preserved, but uses the helper function mapArgsTG′ to properly replace any occurrence of inputs in argument and output lists of the renamed g_2 with the corresponding output nodes of g_1, after adapting them to the respective nested world.

```
seqTG′ : {k m n : ℕ}
       → TG′ _←→_ ø k m
       → TG′ _←→_ ø m n
       → TG′ _←→_ ø k n
seqTG′ g₁ g₂ = inLet′ ε freshø
  (λ {γ} s′ fr as₁ → mapArgsTG′ ε
    (λ s″ as → seqArgs (mapVarArgs (import⊆ (*←→-⊆ s″)) as₁) as)
    (strengthenTG′ { _←→_ } StrongPack.weakOf fr emptyCEnv g₂)
  ) g₁
```

Finally, it is also reasonably easy to convert a TG′ term graph into a Jungle with Fin k as Inner node set and as Edge set, where k is the number of Let declarations.

8   Conclusion and Outlook

Formalising mathematical definitions of term graphs and their operations in Agda is a remarkably straightforward exercise, and, due to the dependent typing of Agda, also carries over to typed term graphs much more easily than in the more restricted type systems of Haskell or higher-order logic.

The remarkable abstract interface to variable binding provided by Pouillard and Pottier's NotSoFresh Agda library [PP10] also makes name-binding representations of term graphs conveniently accessible to mechanised reasoning and programmed manipulation. Typing is easily added to our TG and TG′ datatypes — the original Tm datatype provided as NotSoFresh example includes typing, but we omitted it here to improve readability.

Implementing additional term graph operations, manipulations, and conversion functions, and proving the algebraic properties of the term graph operations is ongoing work.

Future work will strive to base code-graph based optimised-code generation algorithms for the Coconut project [AK09a] on our Agda formalisations of code graphs, with a fully verifying tool chain as ultimate goal.

References

[AK94]   Zena M. Ariola & Jan Willem Klop (1994): *Cyclic Lambda Graph Rewriting*. In: *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Paris, France, pp. 416–425.

[AK08]   Christopher Kumar Anand & Wolfram Kahl (2008): *Code Graph Transformations for Verifiable Generation of SIMD-Parallel Assembly Code*. In Andy Schürr, Manfred Nagl & Albert Zündorf, editors: *Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007. LNCS* 5088, pp. 217–232, doi:10.1007/978-3-540-89020-1.

[AK09a]   Christopher K. Anand & Wolfram Kahl (2009): *An Optimized Cell BE Special Function Library Generated by Coconut*. IEEE Transactions on Computers 58(8), pp. 1126–1138, doi:10.1109/TC.2008.223.

[AK09b]   Christopher K. Anand & Wolfram Kahl (2009): *Synthesizing and Verifying Multicore Parallelism in Categories of Nested Code Graphs*. In Michael Alexander & William Gardner, editors: *Process Algebra for Parallel and Distributed Processing*, chapter 1. *CRC Computational Science Series* 2, Chapman & Hall, pp. 3–45.

[CG99]   Andrea Corradini & Fabio Gadducci (1999): *An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories*. Applied Categorical Structures 7(4), pp. 299–331.

[CR93]   Andrea Corradini & Francesca Rossi (1993): *Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming*. In B. Courcelle & G. Rozenberg, editors: *Selected Papers of the International Workshop on Computing by Graph Transformation, Bordeaux, France, March 21–23, 1991*. Elsevier, pp. 7–48, doi:10.1016/0304-3975(93)90063-Y. Theoretical Computer Science **109** (1–2).

[FS90]   Peter J. Freyd & Andre Scedrov (1990): *Categories, Allegories*. North-Holland Mathematical Library 39, North-Holland, Amsterdam.

[Gon06]   Carlos Gonzalía (2006): *Relations in Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg University. Technical Report No. 14D.

[HFBV97] Armando Haeberer, Marcelo Frias, Gabriel Baum & Paulo Veloso (1997): *Fork Algebras*. In Chris Brink, Wolfram Kahl & Gunther Schmidt, editors: *Relational Methods in Computer Science*, chapter 4. Advances in Computing Science, Springer, Wien, New York, pp. 54–69.

[HP91]   Berthold Hoffmann & Detlef Plump (1991): *Implementing Term Rewriting by Jungle Evaluation*. Informatique théorique et applications/Theoretical Informatics and Applications 25(5), pp. 445–472.

[HS98]   Gérard Huet & Amokrane Saïbi (1998): *Constructive Category Theory*. In: *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. doi:10.1.1.39.4193.

[HS00]   Gérard Huet & Amokrane Saïbi (2000): *Constructive Category Theory*. In Gordon D. Plotkin, Colin Stirling & Mads Tofte, editors: *Proof, language, and interaction: Essays in honour of Robin Milner*. Foundations Of Computing Series, MIT Press, pp. 239–275.

[KAC06]   Wolfram Kahl, Christopher Kumar Anand & Jacques Carette (2006): *Control-Flow Semantics for Assembly-Level Data-Flow Graphs*. In Wendy McCaull, Michael Winter & Ivo Düntsch, editors: *8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 8, Feb. 2005. LNCS* 3929, Springer, pp. 147–160.

[Kan81]   Akira Kanda (1981): *Constructive Category Theory (No. 1)*. In Jozef Gruska & Michal Chytil, editors: *Mathematical Foundations of Computer Science, MFCS '81. LNCS* 118, Springer, pp. 563–577, doi:10.1007/3-540-10856-4_125.

[ML71]   Saunders Mac Lane (1971): *Categories for the Working Mathematician*. Springer-Verlag.

[MOW98]   John Maraist, Martin Oderski & Philip Wadler (1998): *The Call-by-Need Lambda Calculus*. J. Functional Programming 8(3), pp. 275–317.

[Nor07]   Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology. Available at http://www.cs.chalmers.se/~ulfn/papers/thesis.html.

[PP10]   Nicolas Pouillard & François Pottier (2010): *A fresh look at programming with names and binders*. In: *ICFP 2010, Intl. Conf. on Functional Programming*. ACM, New York, NY, USA, pp. 217–228, doi:10.1145/1863543.1863575. Available at http://nicolaspouillard.fr/publis/pouillard-pottier-fresh-look-agda-2010/.

[Tur82]   David A. Turner (1982): *Recursion Equations as a Programming Language*. In J. Darlington, editor: *Functional Programming and its Applications: An Advanced Course*. Cambridge Univ. Press, pp. 1–27.