Electronic Notes in Theoretical Computer Science 69 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume69.html 18 pages

# GS·Λ Theories: A Syntax for Higher-Order Graphs [1]

Matteo Coccia, Fabio Gadducci and Ugo Montanari

*Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.*

**Abstract**

Graphs and term graphs have proved strikingly flexible and expressive in modeling and specifying distributed, concurrent systems. However, even if the theory of (term) graph transformation systems is well developed, it has not yet included higher-order features, which are essential, in ordinary programming languages, to guarantee the levels of parametrisation and modularity that are required in practice.

Our proposal aims at filling that gap by equipping graph-based formalisms with higher-order constructs. Our starting point has been graph substitution (gs) theories by Corradini and Gadducci, which are proved to be syntactical counterparts for term graphs (in the same way as algebraic theories are for terms). As for cartesian categories, gs categories specify symmetric monoidal categories equipped with two transformations, representing sub-term copying and garbage collection. However, for gs theories the naturality of these transformations does not hold.

GS·Λ theories enrich gs theories by requiring the existence of a right adjoint $a \multimap \_$ to the tensor $\_ \otimes a$, for each object $a$. As such, they can be considered as transformation enriched autonomous categories. The resulting categories are tightly linked to the higher-order sharing theories introduced by Hasegawa: these latter are in fact reflective sub-categories of gs·Λ theories, since they are captured by an equivalent axiomatisation, except for the laws concerning the duplication of a set of values (namely, of $\lambda$-abstractions, interpreted as finished computations). Dropping those axioms allow for a neat graphical representation, expanding the wire-and-box notation developed for gs theories in a conservative manner.

*Key words:* Graphical formalisms, higher-order, monoidal theories.

[1] This work has been partially supported by the Italian MIUR project COMETA (*Computational Metamodels*); and by EU within the FET – Global Computing initiative, project AGILE IST-2001-32747 (*Architecture for Mobility*). The funding bodies are not responsible for any use that might be made of the results presented here.

# 1 Introduction

### 1.1 Motivations

Among the conceptual tools and the various formalisms proposed through the years, graphs and graph-based frameworks have proved strikingly flexible and expressive. They have strong cognitive appealing due to their visual nature and their ability to describe complex systems and situations on an intuitive level. Moreover, they have well-established theoretical foundations. One of the most successful and complete frameworks is provided by the theory of graph grammars and graph transformation systems, now at the point of technical maturity as witnessed by [30] and the long list of areas where it has found application [9]. Another related framework is term graph rewriting [32]: term graphs are an expressive generalisations of terms (represented by labeled graphs instead of trees, thus allowing the explicit description of sharing), and they have been often used in compiler construction. A rich variety of high-level programming languages and executable specification formalisms based on graph and term graph rewriting has been devised.

Another general graph based framework is given by the Unified Modeling Language [1] consisting of a set of mostly graphical and semi-formal languages for the specification and documentation of object oriented systems. As largely noted, it misses solid semantic foundations, especially with respect to the description of the dynamics and the behaviour of the system and to the ability of *abstracting out parameters* for large class diagrams. Actually, most of the graph-based frameworks and (programming or specification) languages lack abstraction techniques, which instead prove useful for describing real complex systems and behaviours. The presence of higher-order has been recognised fundamental in practice and rather expressive from the theoretical point of view (the ubiquity of lambda-calculus in semantics being a definitive testimonial).

Our proposal aims to fill this gap by equipping a graph-based formalism with higher-order features.

### 1.2 On gs theories

Our starting point has been *graph substitution* (shortly *gs*) *theories* by Corradini and Gadducci [5,14]. They present the advantageous point of shifting the quest to the categorical logic framework in which higher-order concepts has been largely treated. Roughly, a gs theory is the internal language of a particular symmetric monoidal category equipped with two transformations (called duplicator and discharger) for which, in general, the naturality constraints do not hold. Actually, the theory of gs categories differs from that of cartesian ones for exactly those naturality laws. From a concrete point of view, that generalisation reduces to the passage from terms to term graphs [5], and a suitable extension of the notion of signature, having input/output arities over a free monoid, is then induced.

Interesting applications often require the presence of the categorical opposite of duplicators and dischargers, or a suitable combination of them, and in [2] a wide variety of proposals are presented and compared. GS theories and their extensions are moreover equipped with an intuitive *wire-and-box* graphical notation, giving a visual understanding of the name sharing and garbaging mechanism they want to model: wires represent variables and interfaced boxes represent arrows of the theory. In this way the transformations added to the monoidal structure (symmetries, duplicators and dischargers) suggestively become wire machinery.

### *1.3 Introducing gs·$\Lambda$ theories*

GS·$\Lambda$ theories enrich gs theories by requiring the existence of a right adjoint $a \multimap \_$ to the tensor $\_ \otimes a$, for each object $a$. As such, they can be considered transformation-enriched symmetric monoidal closed categories.

In recent years there has been many research papers on classes of autonomous and *-autonomous categories, mostly in the quest of models for intuitionistic fragments of linear logic. A rich variety of internal languages for those classes of categories has flourished (e.g. [17,23,27]), often having the drawback of a plethora of laws concerning the interweaving of variables and operators. In our work we decided to stick for a typed *combinator language*, in the style of that devised by Curry for the lambda calculus (see e.g. [26]). The relevant axioms of gs·$\Lambda$ theories are categorical counterparts of the $\beta, \eta$ reduction rules of lambda calculus: given arrows $h : u \otimes v \to w$ and $k : u \to v \multimap w$

$$(\beta)\ \Lambda(h) \otimes id_v; \mathsf{plug}_{v,w} = h$$

$$(\eta)\ \Lambda(k \otimes id_v; \mathsf{plug}_{v,w}) = k$$

The arrow $\mathsf{plug}_{v,w} : (v \multimap w) \otimes v \to w$ represents the counit of the adjunction (i.e. the evaluation function); whilst $\Lambda(-)$ is the usual *currying* operator arising from it, so that $\Lambda(h) : u \to v \multimap w$. As in the case of gs theories, the theory of their closed counterpart differs from that of cartesian closed for the two naturality axioms of duplicators and dischargers. They thus inherit the expressiveness of gs theories and have higher-order features: if arrows of gs theories represent term graphs, those of gs·$\Lambda$ theories can be conceived as *parametric specifications over term graphs.*

We extend the wire-and-box notation of gs theories in a conservative manner to encompass the extensions of the theory. A wire wrapper (representing functional closure), a plugger box and an abstractor meta-operator are then added. Interestingly enough, they provide insights about the nature of the corresponding operators, suggesting both a graph-theoretic interpretation of higher-order and interesting relationships with other (braided, tangled) monoidal models, deserving future research.

### *1.4 A sketchy comparison with sharing theories*

The higher-order acyclic sharing theories by Hasegawa [15] are reflective subcategories of gs·$\Lambda$ theories: they are captured by an equivalent axiomatisation, except for the laws concerning the duplication of *values* (that is, of $\lambda$-abstractions, conceived as finished computations: algebraic values in the sense of Führmann [11]). Dropping those axioms allows for a neat graphical representation, expanding the wire-and-box notation developed for gs theories in a conservative manner. Besides the technical points, it is relevant to stress the different interpretations: while sharing theories are of interest whenever the notion of computational effects enter the game (hence, whenever computations are the focus of the formalism), we insist on focusing on gs·$\Lambda$ theories as a syntactical device for the architectural description: the analysis of the actual behaviour of a system is associated to the graph rewriting mechanism (which is not explored here, and we refer to [3]). Shortly, as for gs theories, the step from specification to computation (from graphs to graph transformations) is captured when moving from a theory to its 2-categorical counterpart [4,13].

## 2 Background on gs theories and term graphs

The categorical approach to (equational) logic permits others fruitful generalisations. The notion of theory itself can be progressively weakened or decomposed obtaining more general notions. This idea, far from being new (see [5,6] for an historical tracing) relaxes cartesianity to possibly transformation-enriched monoidality. A recent proposal is that of gs monoidality [5,12].

**Definition 2.1 (GS categories)** *A gs monoidal category $\underline{C}$ (gs category, for short) is a symmetric strict monoidal category $\langle \underline{C}_0, \otimes, e, \rho \rangle$ together with two transformations $\nabla_a : a \to a \otimes a$ and $!_a : a \to e$ (called respectively duplicators and dischargers), satisfying the axioms of Table 1.*

*A gs functor $F$ is a symmetric strict monoidal functor which preserves duplicators and dischargers, i.e. such that $F\nabla_\_ = \nabla_{F\_}$ and $F!_\_ = !_{F\_}$. The category of small gs categories and gs functors will be denoted with $\underline{\mathtt{gsCat}}$.*

Notice that the first three axioms say that for each $a \in \underline{C}_0$, $\langle a, \nabla_a, !_a \rangle$ is a (chosen) commutative comonoid object in $\underline{C}$ [10]. In that respect, a gs category may be considered as a symmetric strict monoidal category with a chosen comonoid structure on each object, subject to suitable coherence axioms: the final set of axioms states that the choice must respect the monoidal structure, i.e., that the transformations must be monoidal (as seen in e.g. [12]).

The following proposition shows that a gs category and a cartesian one differ for the naturality of the transformations the former is equipped with, providing, as a side effect, an indirect proof of coherence (see also [5,24,25]).

**Proposition 2.2** *A gs category $\underline{C}$ is cartesian if $\nabla$ and $!$ are natural transformations.*

Iterative duplication:

$$\nabla_a; (id_a \otimes \nabla_a) = \nabla_a; (\nabla_a \otimes id_a)$$

Exchange of copies:

$$\nabla_a; \rho_{a,a} = \nabla_a$$

Vacuous duplication:

$$\nabla_a; (id_a \otimes !_a) = id_a$$

Monoidality and coherence:

$$!_{a \otimes b} = !_a \otimes !_b$$

$$\nabla_{a \otimes b} = (\nabla_a \otimes \nabla_b); (id_a \otimes \rho_{a,b} \otimes id_b)$$

$$\nabla_e = !_e = id_e$$

Table 1
Equations for gs categories

The idea now is to consider a weaker notion of theory, based on gs categories. This generalisation implies a series of facts and phenomena alien to the cartesian world. The most important is the distinction between *hypersignature* and (usual) signature, which is trivial in the cartesian case. The notion of hypersignature extends that of signature, representing a natural symmetric variant of the latter: the *result* of (the interpretation of) a given operator (symbol) belonging to it is of a product (or more generally a tensor) type, as for the input. In the traditional setting this distinction reduces to triviality because of the universal property of the categorical products.

**Definition 2.3 (Hypersignatures)** *A* hypersignature $\Sigma$ *is a 4-tuple* $\langle S_\Sigma, F_\Sigma, s_\Sigma, t_\Sigma : F_\Sigma \to S_\Sigma^* \rangle$ *where* $S_\Sigma$ *is the set of sorts,* $F_\Sigma$ *is the set of operator symbols, and* $s_\Sigma, t_\Sigma$ *are functions specifying input and output ports.*

Another trade-off of passing from a cartesian to a monoidal world is that the usual *syntactic* notion of term becomes inadequate, for it is not able to carry the information and the distinctions one has gained: it is then needed a more concrete representation, taking into account the finer structure of output ports (or connectors). Notice that the problem may be easily solved defining a logical system for presenting, given a hypersignature $\Sigma$, its free gs theory $\underline{\mathsf{gs}}(\Sigma)$. In Section 2.1 we will take instead a different point of view: abandoning cartesianity means passing from a world of trees (usual terms for algebraic theories) to a world of graphs (term graphs for gs theories).

**Definition 2.4 (GS theories)** *For any hypersignature* $\Sigma$*, the associated gs theory* $\underline{\mathsf{gs}}(\Sigma)$ *is the category whose set of objects is* $S_\Sigma^*$ *(for* $\lambda$ *the empty string) and whose set of arrows is generated by the following inference rules*

$$\frac{u \in S_\Sigma^*}{id_u : u \to u}, \qquad \frac{u, v \in S_\Sigma^*}{\rho_{u,v} : uv \to vu}, \qquad \frac{t : u \to v,\ t' : u' \to v'}{t \otimes t' : uu' \to vv'},$$

$$\frac{u \in S_\Sigma^*}{!_u : u \to \lambda}, \qquad \frac{u \in S_\Sigma^*}{\nabla_u : u \to uu}, \qquad \frac{f \in F_{u,w}}{f_\Sigma : u \to w}.$$

*with respect to the equations in Table 1 (being string composition the tensor on objects) and Table 2 (with side conditions implicitly stated).*

Functoriality:

$$id_{uv} = id_u \otimes id_v$$
$$(t; t_1) \otimes (t_2; t_3) = (t \otimes t_2); (t_1 \otimes t_3)$$

Monoidality:

$$t \otimes id_\lambda = t = id_\lambda \otimes t$$
$$(t \otimes t_1) \otimes t_2 = t \otimes (t_1 \otimes t_2)$$

Naturality:

$$\rho_{u_1,u}; (t \otimes t_1) = (t_1 \otimes t); \rho_{v_1,v}$$

Simmetry:

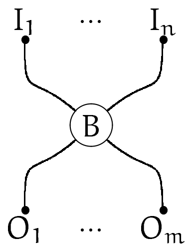
$$\rho_{\lambda,\lambda} = id_\lambda$$
$$\rho_{u,v}; \rho_{v,u} = id_{uv}$$
$$\rho_{uv,w} = (id_u \otimes \rho_{v,w}); (\rho_{u,w} \otimes id_v)$$

Table 2
Equations for gs theories

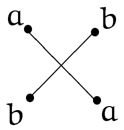
### *2.1 A concrete glance to gs theories*

In this section we introduce a graphical notation for the arrows of gs theories, casting some light on their concrete structure. The correspondence between the arrows of a gs theory and their graphical representations is such that axioms link isomorphic graphs. Thus any calculation becomes just a matter of changing graph representation or, in other terms, of moving wires.

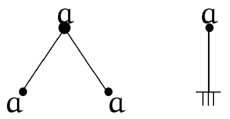
The basic ingredients of the graphical notation have already been put in the foreground by the axiomatic presentation: every theory is made up of objects, arrows coming from the signature and some other auxiliary structure (identities, symmetries, dischargers and so on) satisfying suitable properties. Elements of the signature are depicted as boxes with input/output interfaces made of ports. A port has its own type, specified by an object of the theory.



Boxes may be seen as software/hardware components, systems, gates and so on and may be composed sequentially through their interfaces or put in parallel through the tensor. The auxiliary structure represents the connecting machinery, i.e. special *wires* acting on interfaces. They look like those interface transformers useful to plug an Italian electric device to the Dutch line. Symmetries, for example, swap interfaces and are represented as follows




Duplicators allow the sharing of interfaces whilst dischargers hide them to the environment. They are represented as follows




Let us look now again at the axioms of gs categories, as presented in Table 1. The comonoidality laws of auxiliary transformations gain in clarity when represented pictorially, as already Lafont did in [24,25]. Notice that, accordingly to his convention (and contrary to that of e.g. [2]), we draw arrows downward. We also avoid, in case of auxiliary arrows, to represent explicitly their direction. The two-dimensional equations for gs theories are collected in Table 3.

The graphical notation we introduced simplifies calculations within gs theories. In deciding whether two terms (arrows) are equal, it is often helpful to draw them, being easier to reason directly with them.

## 3 Closing gs categories

The introduction of higher-order is a relatively simple feat: it is just necessary to consider a suitable adjoint situation.

**Definition 3.1 (GS closed categories)** *A* gs closed *category* $\underline{C}$ *(gsC category for short) is a gs category in which the functor* $\_ \otimes a$ *has a right adjoint* $a \multimap \_$*, for each* $a \in \underline{C}_0$*.*

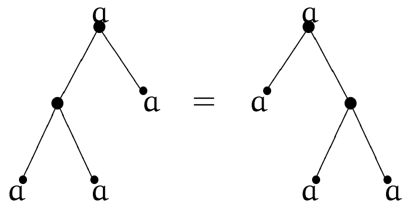
*A gsC functor is a gs functor that in addition commutes with the functor* $a \multimap \_$*, i.e., such that* $F(a \multimap \_) = F(a) \multimap F(\_)$*, for each* $a$*. We shall denote with* $\underline{\mathsf{gsCCat}}$ *the category of gsC categories and gsC functors.*

Again, as in the *first order* framework, the naturality of dischargers and duplicators discriminates between cartesianity and simple monoidality.

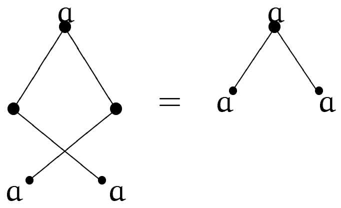
**Proposition 3.2** *A gsC category is cartesian closed if* $\nabla$ *and* $!$ *are natural transformations.*

The additional *higher-order auxiliary structure* does not interfere with the gs one, so the result above is a consequence of the analogous one for $\underline{\mathsf{gsCat}}$. This orthogonality will be clearer after developing the appropriate theory.

Iterative duplication:



Exchange of copies:



Vacuous duplication:

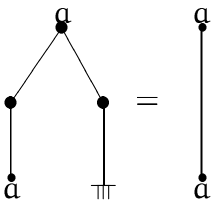


Table 3
Two-dimensional equations for gs theories

In fact, we provide now an axiomatisation of gsC categories in the style of the one given for gs categories. Since the type system of gs·$\Lambda$ theories has another type constructor, namely lollipop, the objects must take it into account. For any hypersignature $\Sigma$, the set $S_\Sigma^{*,\multimap}$ is the closure of $S_\Sigma$ with respect to string composition and $\multimap$.

**Definition 3.3 (GS·$\Lambda$ theories)** *For any hypersignature $\Sigma$, the associated gs $\cdot$ $\Lambda$ theory $\underline{\mathsf{gs}\cdot\Lambda}(\Sigma)$ is the category whose set of objects is $S_\Sigma^{*,\multimap}$ and whose set of arrows is generated by the inference rules and equations (suitably taking into account the new objects) of $\underline{\mathsf{gs}}(\Sigma)$ together with the following rules*

$$\frac{v, w \in S_\Sigma^{*,\multimap}}{\mathsf{plug}_{v,w} : (v \multimap w) \otimes v \to w \in \underline{\mathsf{gs}\cdot\Lambda}(\Sigma)}\ (plugger)$$

$$\frac{h : u \otimes v \to w \in \underline{\mathsf{gs}\cdot\Lambda}(\Sigma)}{\Lambda(h) : u \to v \multimap w \in \underline{\mathsf{gs}\cdot\Lambda}(\Sigma)}\ (abstractor)$$

*and the following (typed) equations:*

$$(\beta)\frac{h : u \otimes v \to w}{\Lambda(h) \otimes id_v ; \mathsf{plug}_{v,w} = h}$$

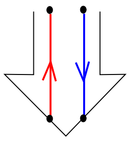
$$(\eta)\frac{k : u \to v \multimap w}{\Lambda(k \otimes id_v ; \mathsf{plug}_{v,w}) = k}$$

Observe that the plugger is the counit of the adjunction (i.e. the evaluation function), while the abstractor is the usual *currying* operator arising from it and the equations are the standard categorical counterparts of $\beta, \eta$ reduction rules of lambda calculus (see e.g. [26]). We have then the following result.

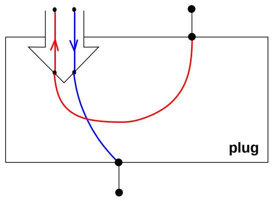
**Proposition 3.4** *For any hypersignature* $\Sigma$*, the associated gs*$\cdot\Lambda$ *theory* $\underline{\mathsf{gs}\cdot\Lambda}(\Sigma)$ *is a small gs closed category.*

### *3.1 Wires, Boxes and Wrappers*

Now that we set up the framework, let us step through a digression. In fact, if definitions are entirely standard, the names we choose for our entities are not. This unusual choice comes from the graph-theoretic intuitions which were under our work: if an arrow $v \to w$ in a gs theory represents an (open) hypergraph with input interface $v$ and output interface $w$, the type $v \multimap w$ corresponds (in some way) to the collection of such graphs. This is the usual interpretation of function types. But in this context, the counit of the adjunction, i.e. the plugger arrow $\mathsf{plug}_{v,w}$, of type $(v \multimap w) \otimes v \to w$, represents an active auxiliary component that connects what it gets from the channel of type $v$ to the interface $v \multimap w$ (i.e., it performs a $\beta$-reduction). This naïve interpretation leads to (and it is supported by) an interesting concrete notation, extending in a conservative way that devised for gs theories. First of all, in passing from gs theories to their closed counterparts we gain a type system's extension: a type constructor, namely lollipop, is added. This is contravariant in its first argument and variant in its second. Graphically, this translates in a *wrapper* packing together two ordered wires inverting the direction of the first.

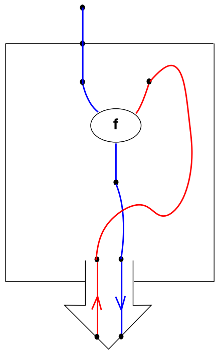


Having inspected this point we can go further exploring the *internal* structure of the box $\mathsf{plug}_{v,w}$. It implements a connecting pattern between two interfaces.




The curry meta operator $\Lambda(-)$ found in this graphical framework has a strikingly simple and suggestive interpretation. It gets an arrow $f$ of type, say, $u \otimes v \to w$, that is a box named $f$ with input interface of type $u \otimes v$ and output interface $w$. It envelops then $f$ into a bigger box forcing forward $v$ into an output higher-interface, with an obvious internal connection of wires. The resulting box $\Lambda(f)$ has $u$ as input interface and $v \multimap w$ as output interface.

The following picture should help the reader




Now, beta and eta axioms transform to the knot-theory game of stretching internal wires, possibly with the additional, further step of getting rid of useless wrappers. Figure 1 and Figure 2 translate reduction rules into our *wrapped wires and boxes* notation.

Fig. 1. $\beta$-equation in wrapped wires and boxes disguise

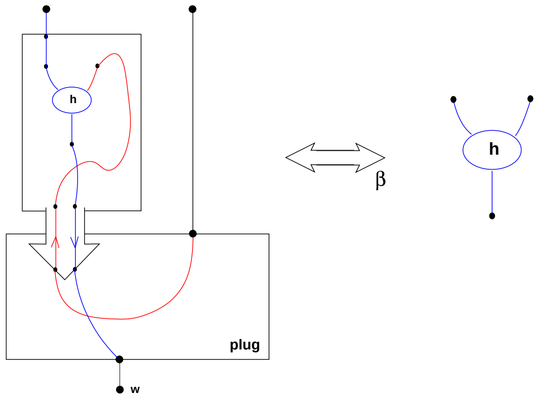
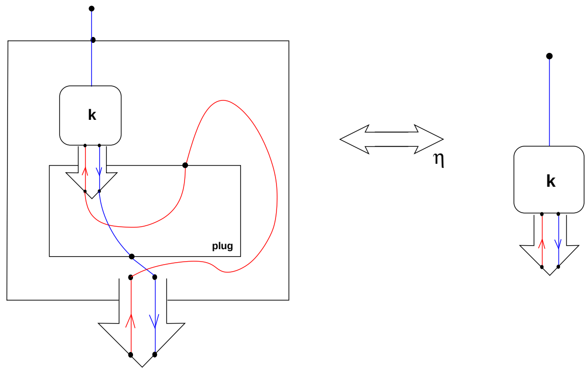



Fig. 2. $\eta$-equation in wrapped wires and boxes disguise



### 3.2 Conservativity Lemma

Consider a hypersignature $\Sigma$ and two elements $u, v \in S_\Sigma^*$. Clearly, each arrow $t : u \to v \in \underline{\mathsf{gs}}(\Sigma)$ is also an arrow of $\underline{\mathsf{gs} \cdot \Lambda}(\Sigma)$, being gs$\cdot\Lambda$ theories extensions of gs theories; thus the following holds

$$\underline{\mathsf{gs}}(\Sigma)[u, v] \hookrightarrow \underline{\mathsf{gs} \cdot \Lambda}(\Sigma)[u, v].$$

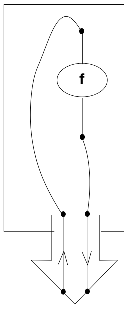
The inclusion map is in fact also surjective.

**Proposition 3.5 (Conservativity)** *For any hypersignature $\Sigma$, $\underline{\mathsf{gs}}(\Sigma)$ is a full subcategory of $\underline{\mathsf{gs} \cdot \Lambda}(\Sigma)$.*

From the previous result we learn that, for any hypersignature $\Sigma$ and lollipop-free objects $u, v$, the following chain of isomorphisms holds

$$\underline{\mathsf{gs}}(\Sigma)[u, v] \leftrightarrow \underline{\mathsf{gs} \cdot \Lambda}(\Sigma)[u, v] \leftrightarrow \underline{\mathsf{gs} \cdot \Lambda}(\Sigma)[\lambda, u \multimap v]$$

Thus, categorically speaking, the inhabitants of higher-order objects (or lollipops) are *wrapped* arrows of the shape




## 4 Sharing Theories as Syntax for GS Theories

Informally, sharing theories [15] are equational presentations of acyclic sharing graphs which in turns are a set-theoretic version of term graphs. They constitute an internal language for gs categories, though the representation of duplicators and dischargers is not explicit. The underlying idea of the combinator syntax for gs theories was to describe systems externally, by means of black-box components and of their parallel and sequential composition, along with special wiring machinery. The idea of term-like syntax is instead to describe systems assuming an internal point of view. Ports are then named through variables and the connectivity is managed through mechanisms for name sharing and discharging. They can be implicit or explicit, depending on the chosen presentation for the associated natural deduction.

**Definition 4.1 (Raw terms)** *For any hypersignature $\Sigma$, the set $ST(\Sigma)$ of raw terms over $\Sigma$ is the language defined by the following grammar*

$$t ::= x \mid * \mid f(t) \mid t_0 \otimes t_1 \mid \mathsf{let}\ (x_1, ..., x_n)\ \mathsf{be}\ s\ \mathsf{in}\ t$$

*where $f \in F_\Sigma$. Terms are subject to the axioms stating the strict associativity of $\otimes$, which in addition has $*$ as its neutral element.*

Often, $\mathsf{let}\ (x_1, ..., x_n)\ \mathsf{be}\ s\ \mathsf{in}\ t$ may be written for convenience $\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t$. Notions of free and bound variables are standard: in a raw term $\mathsf{let}\ (x_1, ..., x_n)\ \mathsf{be}\ s\ \mathsf{in}\ t$, variables $x_i$'s are bound in $t$. We write $s\{^t/_x\}$ for the capture-free substitution of $t$ into free occurrences of $x$ in $s$. Similarly, we use $s\{^{\boldsymbol{t}}/_{\boldsymbol{x}}\}$ for simultaneous substitution.

The system for inferring typing judgments is shown in Table 4. Variable typing assertions are pairs $x : \sigma$ of a variable and a sort, whilst term typing assertions are pairs $t : (\sigma_1, ..., \sigma_n)$ of a term and a finite list of sorts. This is to cope with multiple conclusions output interfaces. It may be abbreviated as $t : (\boldsymbol{\sigma})$. Notice the use of additive contexts that hides sharing and discharging within the let construct.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \quad \text{(var)}$$

$$\frac{f \in \Sigma_{\sigma_1 \cdot \ldots \cdot \sigma_n, \tau_1 \cdot \ldots \cdot \tau_m} \qquad \Gamma \vdash t : (\sigma_1, ..., \sigma_n)}{\Gamma \vdash f(t) : (\tau_1, ..., \tau_m)} \quad \text{(operator)}$$

$$\frac{}{\Gamma \vdash * : ()} \quad \text{(unit)}$$

$$\frac{\Gamma \vdash s : (\sigma_1, ..., \sigma_n) \qquad \Gamma \vdash t : (\tau_1, ..., \tau_m)}{\Gamma \vdash s \otimes t : (\sigma_1, ..., \sigma_n, \tau_1, ..., \tau_m)} \quad \text{(tensor)}$$

$$\frac{\Gamma \vdash s : (\sigma_1, ..., \sigma_n) \qquad \Gamma, x_1 : \sigma_1, ..., x_n : \sigma_n \vdash t : (\tau_1, ..., \tau_m)}{\Gamma \vdash \mathsf{let}\ (x_1, ..., x_n)\ \mathsf{be}\ s\ \mathsf{in}\ t : (\tau_1, ..., \tau_m)} \quad \text{(let)}$$

$$\frac{\Gamma, x : \sigma, x' : \sigma', \Gamma' \vdash t : (\tau_1, ..., \tau_n)}{\Gamma, x' : \sigma', x : \sigma, \Gamma' \vdash t : (\tau_1, ..., \tau_n)} \quad \text{(exchange)}$$

Table 4
Type system for terms

The structural congruence on raw terms is presented in Table 5. The congruence specifies the behaviour of the let construct: it is a sort of explicit substitution operator, and in fact it contains the laws for gs categories. The congruence is actually typed, in the style of equations-in-contexts. Terms are congruent iff they in addition gets the same type. As a consequence we get side conditions on free and bound variables: for instance, in ($\mathrm{ass}_2$), $x'$s cannot be free in $t$ nor $y'$s in $s$.

| | |
|---|---|
| $(\sigma_{var})$ | $\mathsf{let}\ (x)\ \mathsf{be}\ y\ \mathsf{in}\ t \equiv t\{^y/_x\}$ |
| (id) | $\mathsf{let}\ (x)\ \mathsf{be}\ t\ \mathsf{in}\ x \equiv t$ |
| $(\mathrm{ass}_1)$ | $\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ (\mathsf{let}\ (\boldsymbol{y})\ \mathsf{be}\ s\ \mathsf{in}\ t)\ \mathsf{in}\ t' \equiv \mathsf{let}\ (\boldsymbol{y})\ \mathsf{be}\ s\ \mathsf{in}\ (\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ t\ \mathsf{in}\ t')$ |
| $(\mathrm{ass}_2)$ | $\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ (\mathsf{let}\ (\boldsymbol{y})\ \mathsf{be}\ t\ \mathsf{in}\ t') \equiv \mathsf{let}\ (\boldsymbol{x}, \boldsymbol{y})\ \mathsf{be}\ s \otimes t\ \mathsf{in}\ t'$ |
| $(\otimes_1)$ | $t' \otimes (\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t) \equiv \mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ (t' \otimes t)$ |
| $(\otimes_2)$ | $(\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t) \otimes t' \equiv \mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ (t \otimes t')$ |
| (subst) | $\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ f(t) \equiv f(\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t)$ |

Table 5
Structural congruence on raw terms

### *4.1 Higher-Order Acyclic Sharing Theories*

It is quite straightforward to prove that gs theories and sharing theories actually are equivalent categories. Things are a bit more complex when closures are of interests. In [15], a higher-order extension of acyclic sharing theories is developed. As the author notices,

> In some sense it is fairly routine to enrich a theory with lambda terms. [..] The main trouble is to find the *right* axiomatisation for them. It easily turns out that assuming either the full $\beta$ axiom or $\eta$ axiom is sufficient for validating any substitutions in the theory, thus we lose the notion of sharing.

The problem is raised by additive contexts, i.e., by the implicit management of sharing, and the solution chosen by the author resembles Moggi's computational lambda calculus [28] (of which higher-order acyclic sharing theories are a conservative extension). Function closures (i.e. lambda abstractions) are finished computations and *copying them is deemed harmless*. This means that the substitution of function closures is acceptable, or in other words performed. To cope with distinct classes of terms with respect to substitution policies, a notion of value is introduced.

Our perspective is instead different: gs theories are aimed at specifying the structural description of a system (its statics) in opposition to sharing theories, aiming at grasping its evolution (its dinamics). The management of the rewriting mechanism is considered an additional dimension of the framework, separated from system specification, and its representation then deemed to the 2-categorical version of gs closed theories.

**Definition 4.2 (Higher-Order sorts)** *For any hypersignature $\Sigma$, the set of higher-order sorts $S_\Sigma^{(),\Rightarrow}$ is defined by the following grammar*

$$\sigma ::= s \mid (\sigma_1, ..., \sigma_n) \mid \sigma_1 \Rightarrow \sigma_2$$

*where $s$ ranges over $S_\Sigma$, and $\sigma, \sigma_i$'s range over $S_\Sigma^{(),\Rightarrow}$.*

**Definition 4.3** *A* higher-order acyclic sharing theory *over a hypersignature $\Sigma$ is an equational theory on the well-typed terms with sorts $S_\Sigma^{(),\Rightarrow}$ closed under the term construction of acyclic sharing theories plus*

$$\frac{\Gamma, \boldsymbol{x} : \boldsymbol{\sigma} \vdash t : \boldsymbol{\tau}}{\Gamma \vdash \lambda(\boldsymbol{x}).t : (\boldsymbol{\sigma}) \Rightarrow (\boldsymbol{\tau})} \; (abstraction)$$

$$\frac{\Gamma \vdash t : (\boldsymbol{\sigma}) \Rightarrow (\boldsymbol{\tau}) \qquad \Gamma \vdash s : (\boldsymbol{\sigma})}{\Gamma \vdash ts : (\boldsymbol{\tau})} \; (application)$$

*where the equality on terms is a congruence relation containing the axioms for the acyclic sharing theory plus the laws shown in Table 6. Equality must be closed under the new term constructions*

$$\frac{\Gamma, \boldsymbol{x} : \boldsymbol{\sigma} \vdash t \equiv t' : \boldsymbol{\tau}}{\Gamma \vdash \lambda(\boldsymbol{x}).t \equiv \lambda(\boldsymbol{x}).t' : (\boldsymbol{\sigma}) \Rightarrow (\boldsymbol{\tau})} \; (abstraction)$$

$$\frac{\Gamma \vdash t \equiv t' : (\boldsymbol{\sigma}) \Rightarrow (\boldsymbol{\tau}) \qquad \Gamma \vdash s \equiv s' : (\boldsymbol{\sigma})}{\Gamma \vdash ts \equiv t's' : (\boldsymbol{\tau})} \; (application)$$

| | |
|---|---|
| $(\beta)$ | $(\lambda(\boldsymbol{x})t)s \equiv \mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t$ |
| $(\eta_0)$ | $(\lambda(\boldsymbol{x})y(\boldsymbol{x})) \equiv y$ |
| $(\sigma_v)$ | $\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ \lambda(\boldsymbol{y}).s\ \mathsf{in}\ t \equiv t\{^{\lambda(\boldsymbol{y}).s}/_{\boldsymbol{x}}\}$ |
| $(\mathrm{app}_1)$ | $(\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t)t' \equiv \mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ tt'$ |
| $(\mathrm{app}_2)$ | $t'(\mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t) \equiv \mathsf{let}\ (\boldsymbol{x})\ \mathsf{be}\ s\ \mathsf{in}\ t't$ |

Table 6
Axioms for higher-order acyclic sharing theories

The calculus has then a *call by value* flavour. In fact, it is possible to prove the following equations, valid in higher-order sharing theories:

$$(\lambda(\boldsymbol{x})t)v \equiv t\{^v/_x\}$$

$$(\lambda(\boldsymbol{x})v(\boldsymbol{x})) \equiv v \qquad (x \notin fv(v))$$

where $v$ is a *value*, that is a term of the form:

$$v ::= 0 \mid x \mid (\lambda(\boldsymbol{x})t) \mid v \otimes v$$

stating that values may be freely copied and discharged (substitution is performed, and not delayed, for them). The culprit is of course the $(\sigma_v)$ axiom: dropping that, the resulting higher-order sharing theories, and the gs$\cdot\Lambda$ theories, again turn out to be equivalent presentations.

**Proposition 4.4** *For any hypersignature $\Sigma$, let $\underline{\mathsf{st}}(\Sigma)$ be the theory obtained using the inference rules of Definition 4.3 and the laws of Table 6, except for axiom $(\sigma_v)$. Then, $\underline{\mathsf{st}}(\Sigma)$ is equivalent to $\underline{\mathsf{gs} \cdot \Lambda}(\Sigma)$.*

## 5 Conclusions and further works

There is a long tradition in theoretical computer science in finding suitable axiomatic presentations for graph-like structures (see e.g. [7], and [5,6] for an historical tracing), which has received renewed attention after the seminal work of Joyal *et alii* on traced monoidal categories [19]. In fact, gs theories could be alternatively described as premonoidal categories [29]: Our formalism is, among other sources, indebted to the graphical presentation of these categories proposed by Jeffrey [18], and our "closure" can indeed be considered a refinement of the proposal in [31].

In this paper we introduced gs closed categories, presenting a graphical notation for describing the arrows of (freely generated) gs·$\Lambda$ theories. Our results are clearly preliminary: a set-theoretical presentation for our wrapped wire-and-box notation is only hinted at, even if it could be easily given in terms of the hierarchical graphs by Drewes *et alii* [8] (but see also the stratified term graphs by Kahl [20,21]). We are currently investigating for a more polished presentation, along the line of the correspondence between autonomous and compact closed categories [22]: a first proposal can be found in [3].

The correspondence sketched with Hasegawa's higher-order sharing theories is just an useful exercise, needed to highlight our interest toward system specification, instead of its dynamics: a survey of the axiomatics developed for capturing computational effects can be found in [11].

As far as the use of gs closure for rewriting is concerned, we refer the reader again to [3], where an encoding in gs·$\Lambda$ theories of the hyperedge replacement mechanism is presented, and an application to the evolution of software architecture is given, building upon the presentation of an higher-order language for representing derivations on hyperedge replacement systems [16].

Finally, note that the use of higher-order signatures would raise no problem in our framework; hence, we feel confident that a 2-categorical version of gs closure, even if not yet developed, should prove analogous to the presentation in [4,6], and give raise to similar results, as far as graph rewriting is concerned.

## Acknowledgments

We are indebted to Roberto Bruni and Andrea Corradini for the useful comments and suggestions on the topic presented in this paper.

The paper is respectfully dedicated to Matteo Coccia, who passed away a few months ago. This work would have been part of a much richer project, of which his master thesis would have been the first, significant step. A full-fledged draft of the thesis is currently being edited by the second author of this paper, and it will be published as [3].

## References


[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User Guide.* Addison-Wesley, 1997.

[2] R. Bruni, F. Gadducci, and U. Montanari. Normal forms for algebras of connections. *Theoret. Comput. Sci.*, 286:247–292, 2002.

[3] M. Coccia. (Higher order) graphs, (closed) categories and (linear) logics. Technical report, Dipartimento di Informatica, Università di Pisa, 2002. To appear.

[4] A. Corradini and F. Gadducci. A 2-categorical presentation of term graph rewriting. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *Lect. Notes in Comp. Science*, pages 87–105. Springer, 1997.

[5] A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7:299–331, 1999.

[6] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence between the operational and the categorical description. *Informatique Théorique et Applications/Theoretical Informatics and Applications*, 33:467–493, 1999.

[7] V.-E. Căzănescu and Gh. Ştefănescu. A general result on abstract flowchart schemes with applications to the study of accessibility, reduction and minimization. *Theoret. Comput. Sci.*, 99:1–63, 1992.

[8] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph trasformation. *Journal of Computer and System Science*, 64:249–283, 2002.

[9] H. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools.* World Scientific, 1998.

[10] T. Fox. Coalgebras and cartesian categories. *Communications in Algebra*, 4:665–667, 1976.

[11] C. Führmann. Varieties of effects. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lect. Notes in Comp. Science*, pages 144–158. Springer, 2002.

[12] F. Gadducci. *On the Algebraic Approach to Concurrent Term Rewriting.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996.

[13] F. Gadducci, R. Heckel, and M. Llabrés. A bi-categorical axiomatisation of concurrent graph rewriting. In M. Hofmann, D. Pavlovič, and G. Rosolini, editors, *Category Theory and Computer Science*, volume 29 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 1999.

[14] F. Gadducci and U. Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoret. Comput. Sci.*, 285:319–358, 2002.

[15] M. Hasegawa. *Models of Sharing Graphs.* PhD thesis, University of Edinburgh, Department of Computer Science, 1997.

[16] D. Hirsch and U. Montanari. Higher-order hyperedge replacement systems. In Täntzer and Ehrig [33], pages 215–223.

[17] B. Jay. Languages for monoidal categories. *Journal of Pure and Applied Algebra*, 59:61–85, 1989.

[18] A. Jeffrey. Premonoidal categories and a graphical view of programs. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1997. Available at `http://www.cogs.susx.ac.uk/users/alanje/premon/`.

[19] A. Joyal, R.H. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:425–446, 1996.

[20] W. Kahl. Parallel composition and decomposition of specifications. *Information Sciences*, 139:197–220, 1991.

[21] W. Kahl. Stratified term graphs. In Täntzer and Ehrig [33], pages 115–122.

[22] G.M. Kelly. Many variable functorial calculus, I. In G.M. Kelly, M. Laplaza, G. Lewis, and S. MacLane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 66–105. Springer, 1972.

[23] T.W. Koh and C.-H.L. Ong. Explicit substitution internal languages for autonomous and *-autonomous categories. In M. Hofmann, D. Pavlovič, and G. Rosolini, editors, *Category Theory and Computer Science*, volume 29 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1999.

[24] Y. Lafont. Penrose diagrams and 2-dimensional rewriting. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 191–201. Cambridge University Press, 1992.

[25] Y. Lafont. Equational reasoning with 2–dimensional diagrams. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting*, volume 909 of *Lect. Notes in Comp. Science*, pages 170–195. Springer, 1995.

[26] J. Lambek and P.J. Scott. *Introduction to higher-order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, 1986.

[27] I. Mackie, L. Roman, and S. Abramsky. An internal language for autonomous categories. *Applied Categorical Structures*, 1:331–343, 1993.

[28] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[29] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7:453–468, 1998.

[30] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations.* World Scientific, 1997.

[31] R. Schweimeier and A. Jeffrey. A categorical and graphical treatment of closure conversion. In S. Brookes, A. Jung, M. Mislove, and A. Scedrov, editors, *Mathematical Foundations of Programming Semantics*, volume 20 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000.

[32] M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice*. Wiley, 1993.

[33] G. Täntzer and H. Ehrig, editors. *Graph Transformation Systems*. Technical Report, Fachbereich 13 – Informatik, Technische Universität Berlin, 2000.