Diagrammatic logic and effects : the example of exceptions

Dominique Duval, Jean-Claude Reynaud

Diagrammatic logic and effects : the example of exceptions

Dominique Duval*, Jean-Claude Reynaud†

December 21, 2004

Abstract. This paper presents a unified framework for dealing with exceptions in axiomatic specifications and in programming languages. Our framework includes a deduction system and a denotational semantics with respect to a diagrammatic logic [10, 8]. This approach can be seen as an alternative to the monads approach for introducing effects in specifications and in programs [22]; for instance, our denotational semantics is related to the monadic semantics of exceptions by an adjunction result between two different logics. Moreover, in order to build upon a realistic computational model, we use extensive categories as a minimal requirement to express our various logical theories.

Keywords. Exceptions, Diagrammatic Logic, Sketches, Extensive Categories, Specifications, Semantics, Monads.

1 Introduction

This paper presents a unified framework for dealing with the deduction system and the denotational semantics of exceptions, as an instance of a more general framework for formalising various features of actual programming languages. A preliminary version of this work has appeared in [11].

In order to bridge the gap between theoretical approaches and the reality of programming languages, a decisive step has been carried out by Moggi [22], with the basic idea that computational effects can be modelled by the categorical notion of *monad*. In the monadic approach, the distinction between *values* and *computations* is central; this distinction is also one of the starting points of our work.

However, instead of starting from a computational model (the *computational lambda calculus*, or λ_c-calculus), as in [22], here we start from a logical theory in order to describe a deduction system for specifications.

It is well-known that lambda calculus does not fully match actual functional programming languages such as Standard ML: the call-by-value strategy does not match the unrestricted use of beta-reduction, because it matters how often and in which order the arguments of a procedure are evaluated; detailed examples can be found in [15]. Moreover, exponential types are not central for exceptions, since there is an exception mechanism in languages which do not consider the functions as first-class citizens. While the basic type constructor in the lambda calculus is the exponential, in this paper the basic type constructor is the sum. Indeed, sum types are used for case distinctions, which in turn are heavily used in the treatment of exceptions; usually an exception is raised only in some branch of a case distinction, then it must be tested whether the program does raise an exception, and precisely which exception is raised: thus, altogether, the exception mechanism makes use of three kinds of case distinctions.

*LMC-IMAG, Grenoble, Dominique.Duval@imag.fr
†LSR-IMAG, Grenoble, Jean-Claude.Reynaud@imag.fr

A categorical model of the λ_c-calculus is made of a category $\mathcal{C}$ with finite products and a monad T on $\mathcal{C}$ satisfying some additional properties (T is a strong monad with Kleisli exponentials). An object A of $\mathcal{C}$ is viewed as the set of values of type A, and the object TA as the set of computations of type A. A program from A to B can be identified with a morphism from A to TB in $\mathcal{C}$: the programs are the morphisms of the Kleisli category of T. Moggi calls T a *notion of computation* since it abstracts away from the type of values that a computation may produce. Each choice of T corresponds to a notion of computation: side-effects, exceptions, interactive output, etc. The monad of exceptions on the category of sets is such that $TA = A + E$ for every set A, where E is a fixed set of exceptions.

Several denotational semantics have been proposed for the λ_c-calculus; following [15], three denotational semantics can be distinguished. In the *naïve semantics*, a term t of context A and type B simply denotes a morphism $t : A \to B$ in $\mathcal{C}$, which clearly is too crude. In the *monadic semantics*, such a term t denotes a morphism $t : A \to TB$ in $\mathcal{C}$; whenever t is a value, this morphism is obtained from a morphism $t_v : A \to B$ in $\mathcal{C}$, composed with the unit morphism of the monad $\varepsilon : B \to TB$. Several authors have designed other categorical semantics, in order to provide a so-called *direct semantics* for the λ_c-calculus; in [25] it is proved that the approach via *Freyd categories* (based on *premonoidal categories*) is equivalent to the approach via *κ-categories* (based on *indexed categories*); in [15] an approach via *precartesian categories* is proposed. In this paper, an alternative to the monads approach is defined, and another way to get a direct denotational semantics is presented.

Thus, in the monadic approach, the *computational effects* are encapsulated in a monad. In our approach, what we call the *logical effects* are encapsulated in a logic. The direct denotational semantics is defined in this logic, and the monadic semantics can be recovered thanks to a morphism between logics. A logic here is a *diagrammatic logic*, as defined in [10, 8] and reviewed in appendix A: it is a morphism of projective sketches satisfying some property, so that it is endowed with a deduction system and a denotational semantics in a sound way. Two distinct diagrammatic logics are used in this paper: the *decorated logic* for the syntax, the deduction system and the direct denotational semantics of exceptions, and the *explicit logic* for the monadic denotational semantics of exceptions. The explicit logic is a "familiar" kind of logic, which is presented here in the diagrammatic fashion, while the decorated logic is an "unfamiliar" kind of logic. In addition, there is a robust notion of morphism between diagrammatic logics, such that the relation between the direct and the monadic semantics is an easy adjunction result.

Now, let us look more closely at the mechanism of *exceptions*: when an exceptional situation occurs at run-time, then an exception is *raised*, normal program execution is in abandoned, and the exception is *handled*, which means that some actions are performed in response to the arising of the exception. For computing with exceptions, a framework for dealing with non-exceptional situations can be used, at the cost of adding a few keywords (like "`raise`" and "`handle`", or "`throw`", "`try`" and "`catch`") and modifying some of the usual rules. But, for describing a denotational semantics, the set of exceptions has to be explicitly introduced, in order to get a sound interpretation of the computations. Hence, the operational semantics of exceptions on one side, and its denotational semantics on the other side, are described in two fairly distinct frameworks. This paper presents a unified framework for dealing with the axiomatic and the denotational semantics of exceptions, in a Standard ML style [26]. This framework is called the *decorated logic for exceptions*, it is a diagrammatic logic. As every diagrammatic logic, the decorated logic is sound; in addition, its denotational semantics is the direct semantics we are looking for. It is an unfamiliar kind of logic, but there are morphisms (of diagrammatic logics) from this logic to more familiar ones; one such morphism gives rise to the naïve semantics, another one to the monadic semantics.

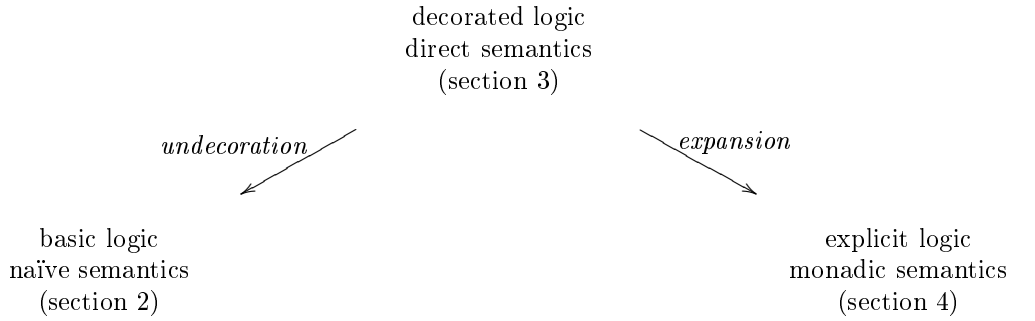
The handling of exceptions is some kind of *case distinction*: either the result of the expression before `handle` is non-exceptional, then this result is returned, or it is exceptional, then the expression following `handle` is executed. In this paper, the handling of exceptions is indeed defined as some kind of case distinction, and the usual properties of the handling of exceptions follow from this definition. For instance, handling exceptions depend on the expressions before and after the keyword `handle` in a natural way. This naturality of handling exceptions does not correspond to the naturality of an *algebraic operation*,

as in [24]: the first one is satisfied, while the second one is not. Hence, this paper also suggests an issue to Plotkin and Power's question in [23]: "*Evident futher work is to consider how other operations such as those for handling exceptions should be modelled. That might involve going beyond monads, as Moggi has suggested to us.*"

Since there is a mechanism of exceptions in various kinds of languages, our basic logical framework is kept as poor as possible: the *basic logic* in this paper corresponds to a language with records and case distinction, so that the content of the paper could be adapted to a functional language like Standard ML as well as to an object-oriented language like Java. This basic logic is defined in section 2, it corresponds to the *extensive categories with products* of [4], which means categories with products and with *well-behaved* sums; it does not deal with exceptions at all. The extensive categories with products correspond to a reasonable fragment of a programming langage, since it has records (thanks to the products) and case distinctions (thanks to the well-behaved sums). In addition, the corresponding syntax is simple enough to be described in a detailed way in a reasonable space.

In section 3, the basic logic is refined in order to classify the terms and, more generally, all the ingredients; in this way we get the *decorated logic*, which determines both the axiomatic semantics and a denotational semantics of exceptions, which is the direct semantics we are looking for. Basically, the decoration of terms corresponds to the distinction between *values* and *computations* in the monads approach [22]: a computation may raise an exception, while a value may not. The crucial point is that all the ingredients in the property that defines a case distinction are also decorated; this gives rise to several notions of case distinctions, which in turn yield a formalisation of the handling of exceptions. A simple *undecoration* morphism, from the decorated logic to the basic logic, allows to look at the proofs with exceptions as a refinement of proofs without exceptions, but it does not preserve the denotational semantics: the undecoration morphism gives rise to the irrelevant naïve semantics.

In section 4, a *logic with explicit exceptions* is defined, as the basic logic together with a distinguished type for exceptions. Thanks to a non-trivial *expansion* morphism, from the decorated logic to the explicit logic, it is proved that our direct semantics gives rise to the monadic semantics, that is, to the usual interpretations of a language with exceptions.



A large part of the paper is devoted to the definition of the three diagrammatic logics and the morphisms between them. The main results are:
– theorems 3.4, 3.18, 3.23 and 3.25 about raising exceptions, handling exceptions, and building records of computations; they assert that our framework does fit with the treatment of exceptions in programming languages;
– theorems 3.5 and 3.19, about the naturality of raising and handling exceptions;
– and theorem 4.6, which establishes the equivalence between our decorated semantics and the monadic semantics, thereby proving that the decorated semantics provides a direct denotational semantics for exceptions.

2 A basic logic

The *diagrammatic* point of view about logic is introduced in [10, 8] and presented in appendix A. Each diagrammatic logic is defined from its *syntax* (the definition of the specifications) and its *axiomatic semantics* (the deduction rules for generating a theory from each specification). A deduction rule is a morphism of specifications, which becomes an isomorphism between the theories generated by these specifications. Both the syntax and the axiomatic semantics of a diagrammatic logic are defined by a *propagator*, that is, by a morphism of projective sketches satisfying some property (see appendix A). Then the *denotational semantics* of the logic defines the *models* of each specification in a systematic way, such that the soundness is guaranted: the models of a specification are preserved by the deduction steps.

In this section, our *basic logic* is defined as a diagrammatic logic, which means that a propagator P_{basic} for this logic is described. This logic does not deal with exceptions; its domains correspond to the *extensive categories with products* of [4], that means, categories with products and sums, where the sums are *well-behaved*, as explained below. It follows that the basic logic corresponds to a language with records and cases. The syntax we use is somewhat similar to the one of Standard ML (or SML) [26]; in SML there is a constructor for products of types, and also a constructor for "datatypes" that combines sums and type recursion. Moreover, in order to deal with computational notions, we use $\equiv$-categories rather than categories, so that for instance terms like $t : X \to Y$ and $\mathrm{id}_Y \circ t : X \to Y$ are not identified, they are only congruent. This basic logic is described here in a detailed way, which should help to understand the decorated logic in section 3.

A *basic specification* is a P_{basic}-specification, a *basic domain* is a P_{basic}-domain.

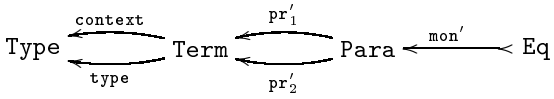
2.1 Types, terms, equations

The propagator P_{basic}, like many usual ones, is based on enrichments of the propagator:

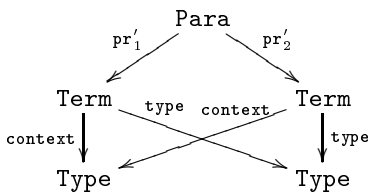
$$P_{\text{comp}} : \mathcal{S}_{\text{comp}} \longrightarrow \overline{\mathcal{S}}_{\text{comp}} ,$$

from example A.16. The P_{comp}-specifications are the compositive graphs, in the sense of appendix A.1; they are made of types and terms with exactly one argument. The P_{comp}-domains are the saturated compositive graphs, in the sense of appendix A.3.

The propagator P_{comp} is enriched, in order to get a propagator $P_{\text{eq}} : \mathcal{S}_{\text{eq}} \to \overline{\mathcal{S}}_{\text{eq}}$ for dealing with unary equations and congruence. Let us say that a pair of terms is *parallel* when both terms share the same type and the same context (that is, type of arguments, as in appendix A). The projective sketch $\mathcal{S}_{\text{eq}}$ is made of a copy of $\mathcal{S}_{\text{comp}}$ together with a point `Para`, which stands for *parallel pairs (of terms)*, a distinguished cone $\Gamma_{\texttt{Para}}$ for the definition of parallel pairs, with two projections $\mathtt{pr}'_1, \mathtt{pr}'_2 : \mathtt{Para} \to \mathtt{Term}$, a point `Eq` for *equations*, and a mono $\mathtt{mon}' : \mathtt{Eq} \rightarrowtail \mathtt{Para}$ for the inclusion:



$\Gamma_{\texttt{Para}}$:



Hence, a P_{eq}-specification can be illustrated as a (directed multi-)graph: a type becomes a point, a term becomes an arrow (the source and target of the arrow are, respectively, the context and the type of the term) and the equations are either mentioned by signs "$\equiv$" added to the graph, or they are written besides the graph.

The propagator $P_{\mathrm{eq}} : \mathcal{S}_{\mathrm{eq}} \to \overline{\mathcal{S}}_{\mathrm{eq}}$ is made of the deduction rules of P_{comp}, together with additional deduction rules, in order to ensure that the equations form a *congruence* relation, in the following sense. The *structural equations* are the usual associativity and unitarity properties of categories, but only up to equivalence. A *congruence* relation is an equivalence relation which contains the structural equations and which is compatible, on both sides, with the composition of terms.

The P_{eq}-domains are called the $\equiv$-*categories* ("equiv-categories"). The $\equiv$-categories where the congruence is the equality are the categories. For instance, the category of sets, with the equality for congruence, is a P_{eq}-domain.

The propagator P_{eq} is now progressively enriched, in order to add products and well-behaved sums of types, which will yield the propagator P_{basic}.

2.2 Products for records

Products of types are used as types for records $(t_1, \ldots, t_n)$. In this way, a n-ary term can be considered as a term with exactly one argument, which is a n-ary record. Here we deal with $\equiv$-*products* ("equiv-products"), which become actual products, in the categorical sense, as soon as the congruence is the equality. Only finite $\equiv$-products are considered. Let n denote any non-negative integer; the illustrations (via the Yoneda functor, as explained in appendix A.4) correspond to $n = 2$.

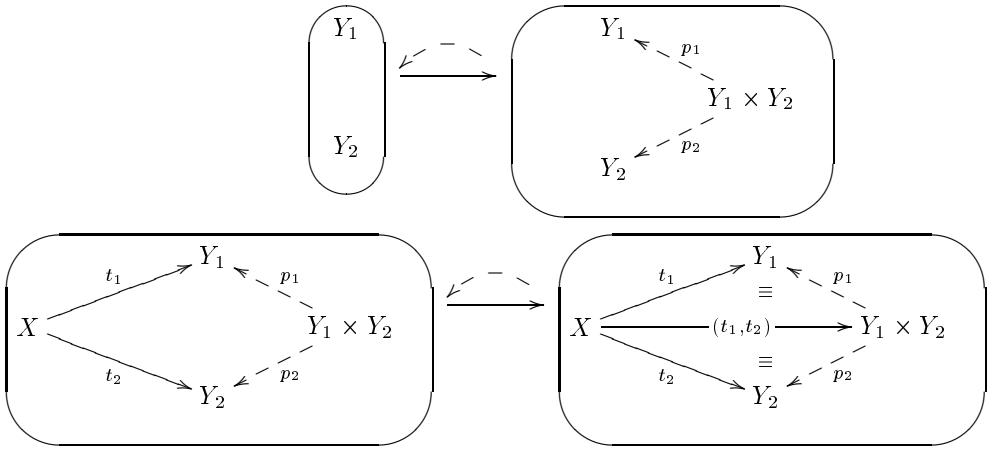
Definition 2.1 An $\equiv$-*product* is a finite discrete cone $(p_i : Y \to Y_i)_{1 \leq i \leq n}$ such that for each cone with the same base $(t_i : X \to Y_i)_{1 \leq i \leq n}$ there is a term $(p_1 : t_1, \ldots, p_n : t_n)$, or simply $(t_1, \ldots, t_n) : X \to Y$, such that for each i:

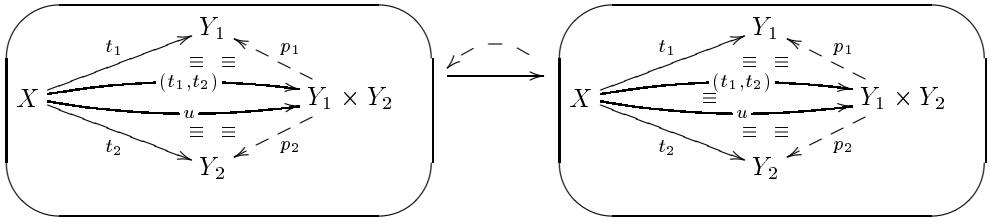
$$p_i \circ (t_1, \ldots, t_n) \equiv t_i \ ,$$

and this term is $\equiv$-*unique*: if a term $u : X \to Y$ is such that $p_i \circ u \equiv t_i$ for each i, then $u \equiv (t_1, \ldots, t_n)$. The *vertex* of the $\equiv$-product is $Y = Y_1 \times \ldots \times Y_n = \prod_{i=1}^{n} Y_i$ and its *projections*, or *labels*, are the p_i's. A shortcut for "the $\equiv$-product with vertex $Y = \prod_{i=1}^{n} Y_i$ and projections $p_i : Y \to Y_i$" is "the $\equiv$-product $\prod_{i=1}^{n}(p_i : Y \to Y_i)$". The term $(t_1, \ldots, t_n) : X \to Y$ is the *record* of $t_1, \ldots, t_n$.

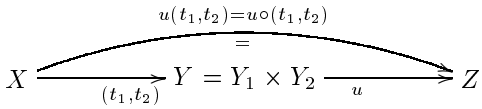
Rules 2.2 *In a basic domain, every tuple of types has a chosen $\equiv$-product, and discrete cone has a chosen record.*

This corresponds to three active deduction rules of P_{basic}: on efor the existence of a chosen $\equiv$-product, a second one for the existence of a chosen record, and the last one for the $\equiv$-unicity of records. With the convention that projections are represented as dashed arrows, these rules can be illustrated as follows; in the last one, there are double symbols "$\equiv$" where one equation involves (t_1, t_2) and the other one involves u.





Now, a n-ary term u with parameters of types $Y_1, \ldots, Y_n$ and with type Z, is considered as a unary term $u : Y \to Z$ with context $Y = \prod_{i=1}^{n} Y_i$. Then, the term $u(t_1, \ldots, t_n) : X \to Z$ is composed from $(t_1, \ldots, t_n) : X \to Y$ and $u : Y \to Z$:



The usual properties of records are easily derived from these properties, for instance below the compatibility of records with congruence and composition.

Proposition 2.3 *Let $t_i : X \to Y_i$ for $i = 1, \ldots, n$.*

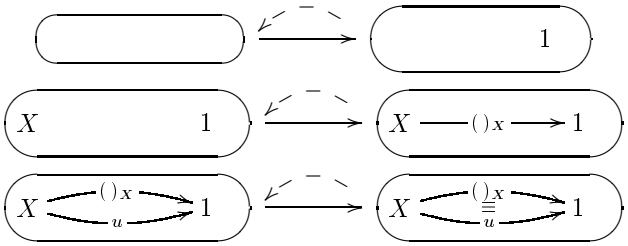
- *Let $t'_i \equiv t_i$ for each i, then:*

$$(t'_1, \ldots, t'_n) \equiv (t_1, \ldots, t_n) : X \to \prod_{i=1}^{n} Y_i \, .$$

- *Let $t' : X' \to X$, then:*

$$(t_1, \ldots, t_n) \circ t' \equiv (t_1 \circ t', \ldots, t_n \circ t') : X' \to \prod_{i=1}^{n} Y_i \, .$$

When $n = 0$, the vertex of the $\equiv$-product is a *terminal point*, denoted 1 (or Unit, or Void). The deduction rules become:



2.3 Well-behaved sums for cases

A case distinction, or simply a *case*, looks like:

$$\text{case } u \text{ of } [j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n] \, ,$$

where $u : X \to Y$ is a term and $\sum_{i=1}^{n} (j_i : Y_i \to Y)$ is a sum. This assumption about the j_i's means that one and only one among the patterns in the match is relevant, for each value of u; this could be weakened. The terms t_i's all have the same type Z. When u is the identity id_Y, the context of t_i is Y_i. But in general, $u : X \to Y$ is any term, and the context of t_i is a type dependent on u, which is denoted

$u^{-1}(Y_i)$; in a set-valued interpretation, it is the set of pairs (x, y_i) where x in X and y_i in Y_i are such that $u(x) = j_i(y_i)$. The case construction can be defined from the notion of *well-behaved sums*. This notion is borrowed from [4], where an *extensive category* is defined as a category with well-behaved sums, and it is proved that an extensive category with products is a *distributive category*, and moreover the relation between the well-behaviour of sums and the existence of pullbacks along coprojections is stated. Here, let us first introduce sums, then the well-behaviour. We deal with $\equiv$-*sums* ("equiv-sums"), which become actual sums as soon as the congruence is the equality. Only finite $\equiv$-sums are considered. The $\equiv$-sums are dual to the $\equiv$-products. As above, n denotes a non-negative integer, and the illustrations correspond to $n = 2$.

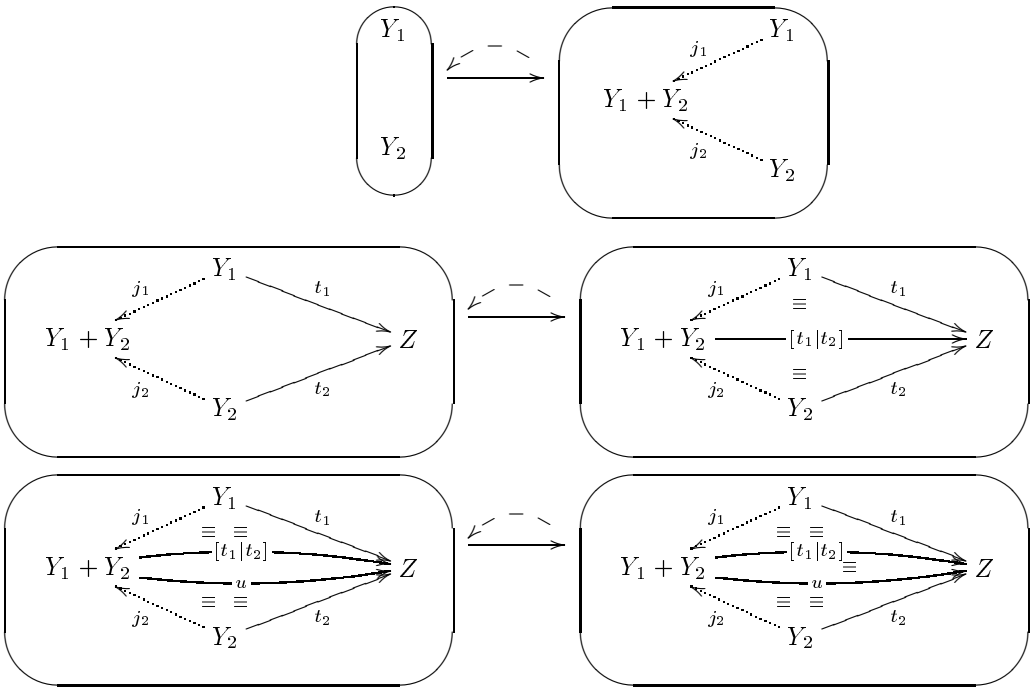
Definition 2.4 An $\equiv$-*sum* is a finite discrete cocone $(j_i : Y_i \to Y)_{1 \leq i \leq n}$ such that for each cocone with the same base $(t_i : Y_i \to Z)_{1 \leq i \leq n}$ there is a term $[j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n]$, or simply $[t_1 \mid \ldots \mid t_n] : Y \to Z$, such that for each i:

$$[t_1 \mid \ldots \mid t_n] \circ j_i \equiv t_i \ ,$$

and this term is $\equiv$-*unique*: if a term $u : Y \to Z$ is such that $u \circ j_i \equiv t_i$ for each i, then $u \equiv [t_1 \mid \ldots \mid t_n]$. The *vertex* of the $\equiv$-sum is $Y = Y_1 + \ldots + Y_n = \sum_{i=1}^{n} Y_i$ and its *coprojections*, or *patterns*, are the j_i's. A shortcut for "the $\equiv$-sum with vertex $Y = \sum_{i=1}^{n} Y_i$ and coprojections $j_i : Y_i \to Y$" is "the $\equiv$-sum $\sum_{i=1}^{n} (j_i : Y_i \to Y)$". The term $[t_1, \ldots, t_n] : Y \to Z$ is the *match* of $t_1, \ldots, t_n$.

Rules 2.5 *In a basic domain, every tuple of types has a chosen $\equiv$-sum, and each tuple of terms with the same type has a chosen match.*

This corresponds to active deduction rules of P_{basic}. With the convention that coprojections are represented as dotted arrows, these rules can be illustrated as:



The usual properties of matches are easily derived from these properties, for instance below the compatibility of matches with congruence and composition.

Proposition 2.6 *Let $t_i : Y_i \to Z$ for $i = 1, \ldots, n$.*

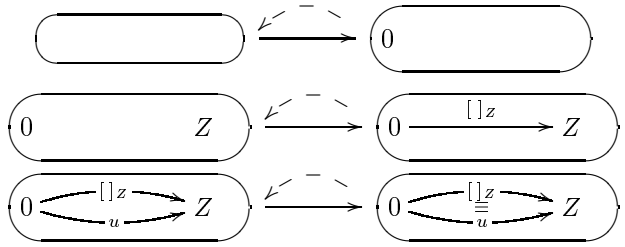
- *Let $t'_i \equiv t_i$ for each i, then:*

$$[t'_1 \mid \ldots \mid t'_n] \equiv [t_1 \mid \ldots \mid t_n] : \sum_{i=1}^{n} Y_i \to Z \ .$$

- *Let $t' : Z \to Z'$, then:*

$$t' \circ [t_1 \mid \ldots \mid t_n] \equiv [t' \circ t_1 \mid \ldots \mid t' \circ t_n] \equiv : \sum_{i=1}^{n} Y_i \to Z' \ .$$

When $n = 0$, the vertex of the $\equiv$-sum is an *initial point*, denoted 0. The deduction rules get simpler:



In order to formalize all kind of cases, as explained above, it is assumed that the $\equiv$-sums are "well-behaved", in the sense of [4], but only up to equivalence, in the following sense.

Definition 2.7 Two $\equiv$-sums $\sum_{i=1}^{n}(k_i : X_i \to X)$ and $\sum_{i=1}^{n}(k'_i : X'_i \to X)$ with the same vertex are *equivalent* if for each i there is an invertible term $s_i : X'_i \to X_i$ such that $k_i \circ s_i \equiv k'_i$. This is denoted:

$$\sum_{i=1}^{n}(k_i : X_i \to X) \cong \sum_{i=1}^{n}(k'_i : X'_i \to X) \ .$$

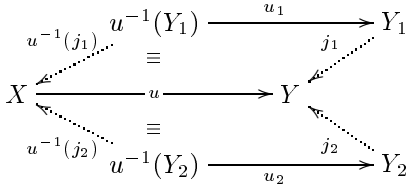
Definition 2.8 Let $\sum_{i=1}^{n}(j_i : Y_i \to Y)$ be an $\equiv$-sum and $u : X \to Y$ a term. An *inverse image of* $(j_i : Y_i \to Y)$ *by* u is an $\equiv$-sum

$$u^{-1}(\sum_{i=1}^{n}(j_i : Y_i \to Y)) = \sum_{i=1}^{n}(u^{-1}(j_i) : u^{-1}(Y_i) \to X)$$

and terms $u_i : u^{-1}(Y_i) \to Y_i$ such that for each i:

$$u \circ u^{-1}(j_i) \equiv j_i \circ u_i \ ,$$

and such that this $\equiv$-sum is $\cong$-unique.

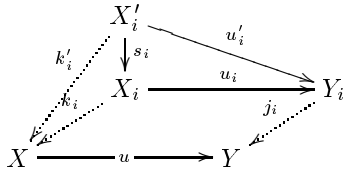


The terms $u_i : u^{-1}(Y_i) \to Y_i$ are called the *restrictions* of u with respect to $\sum_{i=1}^{n}(j_i : Y_i \to Y)$. They are such that $u \equiv \sum_{i=1}^{n} u_i$.

The notation $u^{-1}(\sum_{i=1}^{n}(j_i : Y_i \to Y))$ is ambiguous, but only up to $\cong$-unicity; this ambiguity will disappear as soon as one such inverse image will be chosen. But the notation $u^{-1}(Y_i)$ remains ambiguous as soon as several among the Y_i's share the same name: whenever this occurs, this notation has to be changed.

Proposition 2.9 *Let $\sum_{i=1}^{n}(j_i : Y_i \to Y)$ be an $\equiv$-sum and $u : X \to Y$ a term. Let $X_i = u^{-1}(Y_i)$ and $k_i = u^{-1}(j_i)$. Let an $\equiv$-sum $\sum_{i=1}^{n}(k'_i : X'_i \to X)$ and terms $u'_i : X'_i \to Y_i$ be such that $u \circ k'_i \equiv j_i \circ u'_i$, so that from the $\cong$-unicity there are invertible terms $s_i : X'_i \to X_i$ such that $k_i \circ s_i \equiv k'_i$ for each i. Then, $u_i \circ s_i \equiv u'_i$ for each i.*

Proof.



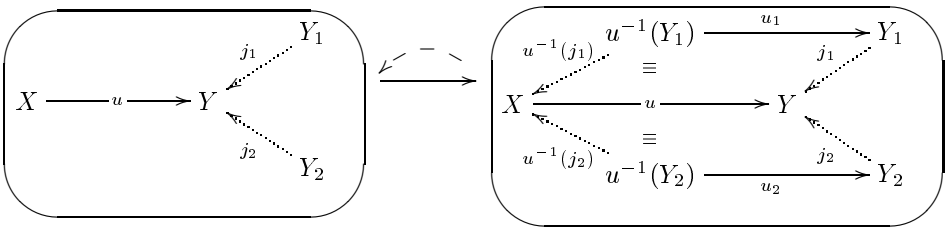
It is known that:

$$j_i \circ u_i \circ s_i \equiv u \circ k_i \circ s_i \equiv u \circ k'_i \equiv j_i \circ u'_i \ .$$

According to [4], an extensive category with products is distributive, and the coprojections in a distributive category are monic. Similarly, in the basic logic, the coprojections are monic up to $\equiv$, so that it can be derived from $j_i \circ u_i \circ s_i \equiv j_i \circ u'_i$ that $u_i \circ s_i \equiv u'_i$, as required. □

Rules 2.10 *In a basic domain, every chosen $\equiv$-sum has a chosen inverse image by every term.*

This corresponds to active deduction rules of P_{basic}, here is an illustration of the existence rule:



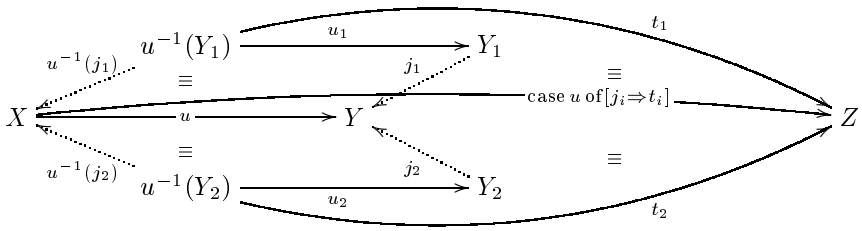
The compatibility of inverse images with respect to congruence and composition, as stated now, easily results from the $\cong$-unicity of inverse images.

Proposition 2.11 *Let Y be the vertex of a sum $\sum_{i=1}^{n}(j_i : Y_i \to Y) = \sum_i j_i$.*

- *Let $u' \equiv u : X \to Y$, then:*
$$u^{-1}(\sum_i j_i) \cong {u'}^{-1}(\sum_i j_i) \ .$$
- *Let $u : X \to Y$ and $u' : X' \to X$, then:*
$${u'}^{-1}(u^{-1}(\sum_i j_i)) \cong (u \circ u')^{-1}(\sum_i j_i) \ .$$

Definition 2.12 Let $u : X \to Y$, with $\sum_{i=1}^{n}(j_i : Y_i \to Y)$, and let $t_i : u^{-1}(Y_i) \to Z$ for each i. Then the term "case u of $[\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,]$", which is called a *case construction*, is defined as:

$$\text{case } u \text{ of } [\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,] \ = \ [\, u^{-1}(j_1) \Rightarrow t_1 \mid \ldots \mid u^{-1}(j_n) \Rightarrow t_n \,] \ : \ X \to Z \ .$$



This means that the term "case u of $[\, j_i \Rightarrow t_i \,]_{1 \le i \le n}$" is characterized, up to $\equiv$, by the equations:

$$\text{case } u \text{ of } [\, j_i \Rightarrow t_i \,]_{1 \le i \le n} \circ (u^{-1}(j_i)) \equiv t_i \quad \text{, for } \; 1 \le i \le n \; .$$

Clearly, when $u = \mathrm{id}_Y : Y \to Y$:

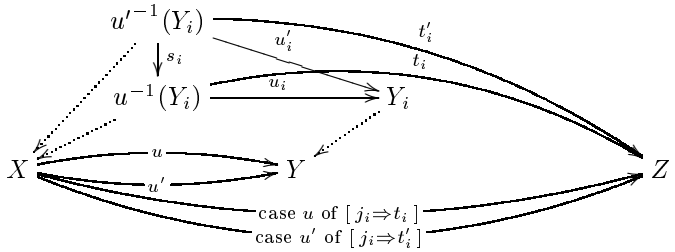
$$\text{case } \mathrm{id}_Y \text{ of } [\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,] \;=\; [\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,] \;:\; Y \to Z \; .$$

The compatibility of cases with congruence and composition (on both sides) is now easily derived from this definition and from the properties of matches (proposition 2.6) and inverse images (proposition 2.11). In the illustrations, for readibility, only one index i is mentioned, and the equations are omitted.

Proposition 2.13 *Let Y be the vertex of a sum $\sum_{i=1}^{n}(j_i : Y_i \to Y)$.*

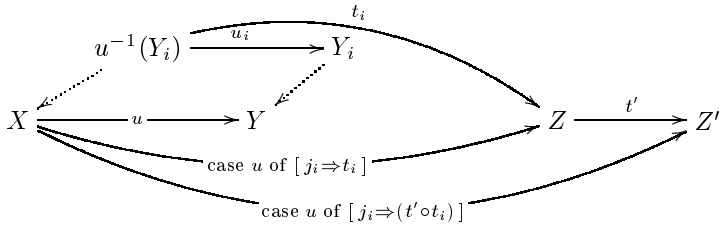
- *Let $u' \equiv u : X \to Y$, let $s_i : u'^{-1}(Y_i) \to u^{-1}(Y_i)$ be the invertible terms arising from proposition 2.11, and let $t_i : u^{-1}(Y_i) \to Z$ and $t'_i : u'^{-1}(Y_i) \to Z$ be such that $t'_i \equiv t_i \circ s_i$ for each i. Then:*

$$\text{case } u' \text{ of } [\, j_i \Rightarrow t'_i \,]_{1 \le i \le n} \equiv \text{case } u \text{ of } [\, j_i \Rightarrow t_i \,]_{1 \le i \le n} : X \to Z \; .$$



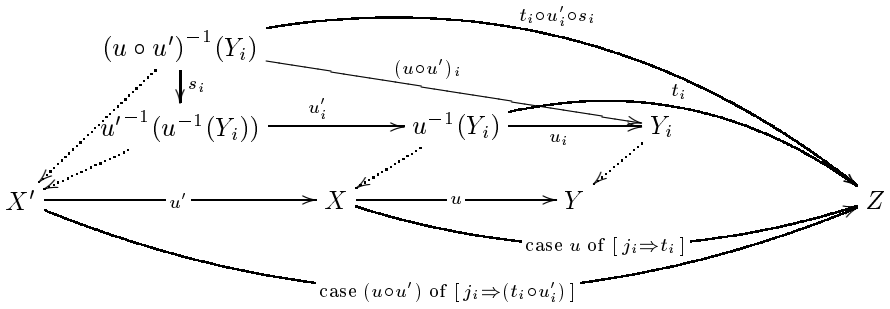
- *Let $u : X \to Y$ and $t' : Z \to Z'$, and let $t_i : u^{-1}(Y_i) \to Z$ for each i. Then:*

$$t' \circ (\, \text{case } u \text{ of } [\, j_i \Rightarrow t_i \,]_{1 \le i \le n} \,) \equiv \text{case } u \text{ of } [\, j_i \Rightarrow (t' \circ t_i) \,]_{1 \le i \le n} : X \to Z' \; .$$



- *Let $u : X \to Y$ and $u' : X' \to X$. Let $s_i : (u \circ u')^{-1}(Y_i) \to u'^{-1}(u^{-1}(Y_i))$ be the invertible terms arising from proposition 2.11, let $u'_i : u'^{-1}(u^{-1}(Y_i)) \to u^{-1}(Y_i)$ be the restrictions of u', and let $t_i : u^{-1}(Y_i) \to Z$ for each i. Then:*

$$(\, \text{case } u \text{ of } [\, j_i \Rightarrow t_i \,]_{1 \le i \le n} \,) \circ u' \equiv \text{case } (u \circ u') \text{ of } [\, j_i \Rightarrow (t_i \circ u'_i \circ s_i) \,]_{1 \le i \le n} : X' \to Z \; .$$



2.4 The basic logic

The *basic propagator* $P_{\text{basic}} : \mathcal{S}_{\text{basic}} \to \overline{\mathcal{S}}_{\text{basic}}$ is the enrichment of P_{eq} with the deduction rules for $\equiv$-products and well-behaved $\equiv$-sums. The *basic logic* is the diagrammatic logic associated to the propagator P_{basic}.

The basic domains are the saturated compositive graphs with congruence, $\equiv$-products and well-behaved $\equiv$-sums. When the congruence is the equality, the basic domains are the extensive categories with products [4]. Each basic specification Σ_{basic} generates a basic domain $\mathrm{F}_{P_{\text{basic}}}(\Sigma_{\text{basic}})$, called *the basic theory of* Σ_{basic}, with all the types which are generated from the types of Σ_{basic} by $\equiv$-products, $\equiv$-sums and inverse images of $\equiv$-sums, all the terms which are generated from the terms of Σ_{basic} by composition, records and cases, and all the equations which are generated from the equations of Σ_{basic} in order to form a congruence relation.

From now on, $\equiv$-products and $\equiv$-sums are called simply *products* and *sums*.

2.5 Basic models

Definition 2.14 Let Σ_{basic} be a basic specification and Δ_{basic} a basic domain. The set of *basic models of* Σ_{basic} *with values in* Δ_{basic} is defined as in appendix A:

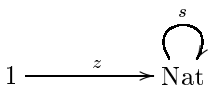
$$\mathrm{Mod}_{\text{basic}}(\Sigma_{\text{basic}}, \Delta_{\text{basic}}) = \mathrm{Hom}_{\mathbf{Dom}(P_{\text{basic}})}(\mathrm{F}_{P_{\text{basic}}}(\Sigma_{\text{basic}}), \Delta_{\text{basic}}) \simeq \mathrm{Hom}_{\mathbf{Spec}(P_{\text{basic}})}(\Sigma_{\text{basic}}, \mathrm{G}_{P_{\text{basic}}}(\Delta_{\text{basic}}))\ .$$

The category of sets with the equality for congruence, the cartesian products for products, the disjoint unions for sums, and the canonical pullbacks for inverse images, is a basic domain, denoted **Set**.

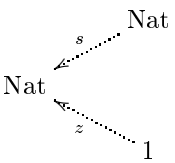
Definition 2.15 Let Σ_{basic} be a basic specification. A *set-valued model of* Σ_{basic} is a basic model of Σ_{basic} with values in the basic domain **Set**.

This simply means that, as usual, a set-valued model of Σ_{basic} interprets types as sets, terms as maps, equations as equalities, products as cartesian products and sums as disjoint unions.

Example 2.16 Let Σ_{nat} be the basic specification:



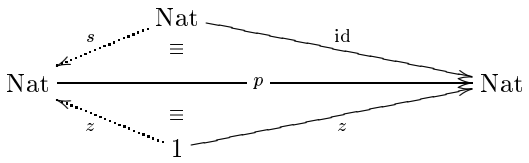
with the product: 1 and the sum:



Hence, Σ_{nat} has types Nat and 1, terms $z : 1 \to \text{Nat}$ and $s : \text{Nat} \to \text{Nat}$, no equation, and empty records $(\ )_X : X \to 1$ can be added, as well as binary matches $[s \Rightarrow u \mid z \Rightarrow t] : \text{Nat} \to X$. We are interested

in the set-valued model M_{nat} of Σ_{nat} that interprets Nat as the set $\mathbb{N}$ of naturals, z as $0 \in \mathbb{N}$ and s as the successor map $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$. Let $\mathrm{pred} : \mathbb{N} \to \mathbb{N}$ denote the predecessor map on the positive integers, extended to the naturals by $\mathrm{pred}(0) = 0$. It is easy to build from Σ_{nat} a term $p : \mathrm{Nat} \to \mathrm{Nat}$ which formalizes pred (the subscript Nat is omitted):

$$p = \text{case id of } [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow z \,] = [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat}$$



3 A decorated logic for exceptions

Now, let us define some available mechanism for exceptions, similar to the one in SML [26]. Then, the predecessor map pred, from example 2.16, can also be formalised in the following way, in pseudo-SML syntax.

- First, an exception e is created:
$$\text{Exception } e$$
- Then a term $p' : N \to N$ is generated, such that $p'(z)$ raises the exception e:
$$p'(x) = \text{case } x \text{ of } [\, s(y) \Rightarrow y \mid z \Rightarrow \text{raise } e \,]$$
- And finally, a term $p'' : N \to N$ is generated, which calls p' and handles the exception e:
$$p''(x) = p'(x) \text{ handle } [\, e \Rightarrow z \,]$$

We are going to modify the basic logic, in order to be able to deal with the mechanism of exceptions, which means to deal with the three keywords:

$$\text{Exception}\,, \quad \text{raise}\,, \quad \text{handle}\,.$$

For this purpose, we use a kind of logic where the terms are associated to symbols, which are called *decorations*, and which appear as superscripts. The main decorations are "v" for "*value*" and "c" for "*computation*", they are borrowed from the monads approach [22]. Moreover, all the ingredients of a specification are decorated: the types (in a trivial way), the equations, and above all, the properties of the products and sums are also decorated, which gives rise to a variety of decorated features. Our point of view about exceptions is that expressions of the form:

$$\text{raise } e \qquad \text{or} \qquad u \text{ handle } t\,,$$

as well as any other expressions, can be considered as decorated terms. In actual programming languages, decorations do not appear, but they can be derived from the use of the keywords and from the rules of the decorated logic: each exception is a computation, then the rules imply that every term involving an exception is also a computation.

The propagator P_{deco} for this *decorated logic* for exceptions is described in this section. It yields an axiomatic semantics and a denotational semantics of exceptions. Another point of view on the denotational semantics of exceptions will be given in section 4.

A *decorated specification* is a P_{deco}-specification, a *decorated domain* is a P_{deco}-domain. In this section, a decorated specification $\Sigma_{\mathrm{nat,deco}}$ is built, in order to define p'' as a decorated term.

3.1 The decoration "v" for "value"

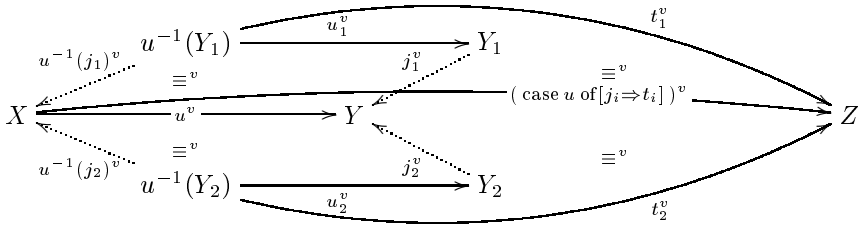
The types are not decorated. All other features are decorated, but some decorations may be omitted in the notations, when they can be easily guessed in a non-ambiguous way.

The terms which have nothing to do with the exceptions are called *values*; they are decorated with the symbol v, as well as the equations between them.

The identities are values, and the composition of values is a value. The value equations generate a congruence. The products and sums of types behave as in the basic logic, with values instead of arbitrary terms: the projections (or labels) and the coprojections (or patterns) are values, a record of values is a value, a match of values is a value, and sums are well-behaved with respect to values, so that cases over values give rise to values. This property means that all the rules of the basic logic give rise to rules of the decorated logic where each term and equation is decorated with v. These products and sums, records, matches and cases are denoted as in the basic logic, so that the terminal and the initial types for values are denoted respectively 1 and 0.

For the case construction, this means that a case like "case u of $[\, j_i \Rightarrow t_i \,]_i$", where u and the t_i's are values, is the value:

$$(\text{case } u \text{ of } [\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,])^v \;=\; [\, u^{-1}(j_1) \Rightarrow t_1 \mid \ldots \mid u^{-1}(j_n) \Rightarrow t_n \,]^v \;:\; X \to Z \;.$$



Example 3.1 In our example, the value part of the decorated specification $\Sigma_{\text{nat,deco}}$ is a copy of the basic specification Σ_{nat} from example 2.16. Hence, $\Sigma_{\text{nat,deco}}$ has types Nat and 1, values $z^v : 1 \to \text{Nat}$ and $s^v : \text{Nat} \to \text{Nat}$, and no value equation. It generates a value:

$$p^v = \text{case id of } [\, s \Rightarrow \text{id} \mid z \Rightarrow z \,] = [\, s \Rightarrow \text{id} \mid z \Rightarrow z \,] : \text{Nat} \to \text{Nat} \;.$$

from which it follows that $p \circ s \equiv^v \text{id}$ and $p \circ z \equiv^v z$.

3.2 The decoration "c" for "computation"

All the terms which may raise exceptions, and which propagate the exceptions, are called *computations*; they are decorated with the symbol c, as well as the equations between them. From this definition, computations *may* (and not *must*) raise exceptions, hence every value can be considered as a computation, so that the decoration c is more general than v: each value t^v may be *coerced* into a computation t^c. Similarly, each value equation may be coerced into a computation equation. Moreover, it is assumed that each computation equation between values comes from a value equation, so that the decoration of equations does not really matter.

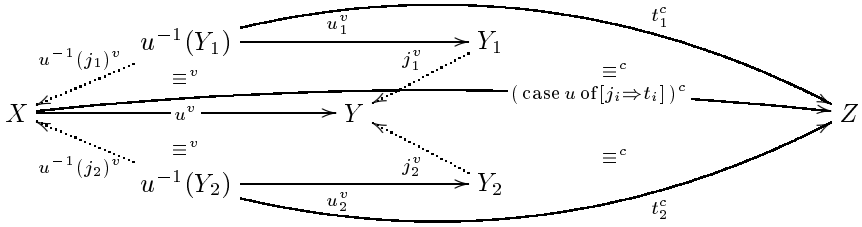
The composition of computations yields a computation: in the composition $(u \circ t)^c = u^c \circ t^c$, any exception which is raised by t^c is propagated by u^c. The computation equations generate a congruence.

As is well-known, there is no straightforward way to build a record of computations: if two terms t^c and u^c raise two distinct exceptions e and e', there is no canonical way to decide what should be the result of a record of t^c and u^c. However, most often, a choice is done; it may correspond to the fact that "t is evaluated before u"; the corresponding record of computations will be defined in section 3.8.

On the contrary, a match of computations is a computation, in a straightforward way. When $n = 0$, this means that the initial type for values is also initial for computations: for every type X, there is a $\equiv^v$-unique value $[]^v_X : 0 \to X$, and its coercion as a computation $[]^c_X$ is the $\equiv^c$-unique computation $[]^c_X : 0 \to X$.

Since a match of computations is a computation, a case like "case u of $[\, j_i \Rightarrow t_i \,]_i$" is defined when the t_i's are computations and u is a value; the same notation "case" is used for this construction:

$$(\text{case } u \text{ of } [\, j_1 \Rightarrow t_1 \mid \ldots \mid j_n \Rightarrow t_n \,])^c \;=\; [\, u^{-1}(j_1) \Rightarrow t_1 \mid \ldots \mid u^{-1}(j_n) \Rightarrow t_n \,]^c \;:\; X \to Z \,.$$



But there is no such definition when u is a computation; indeed, if u raises an exception, there is no canonical way to decide which Y_i the exception "comes from". However, in section 3.5 a special situation is described, where some kind of "case u of ..." can be defined when u is a computation.

Each set $\mathbb{E}$ gives rise to a decorated domain $\mathbf{Set}_{\text{deco}}[\mathbb{E}]$, where:
– a type is a set;
– a value $\varphi^v : A \to B$ is a map $\varphi_v : A \to B$;
– the composition $(\psi \circ \varphi)^v : A \to C$ of values $\varphi^v : A \to B$ and $\psi^v : B \to C$ is the map $(\psi \circ \varphi)_v = \psi_v \circ \varphi_v : A \to C$;
– a computation $\varphi^c : A \to B$ is a map $\varphi_c : A \to B + \mathbb{E}$;
– the composition $(\psi \circ \varphi)^c : A \to C$ of computations $\varphi^c : A \to B$ and $\psi^c : B \to C$ is defined in the Kleisli way: it is the map $(\psi \circ \varphi)_c = [\psi_c \mid r_C] \circ \varphi_c : A \to C + \mathbb{E}$, where $r_C : \mathbb{E} \to C + \mathbb{E}$ is the inclusion;
– the coercion of a value $\varphi^v : A \to B$ to a computation $\varphi^c : A \to B$ is the map $\varphi_c = i_B \circ \varphi_v : A \to B + \mathbb{E}$, where $i_B : B \to B + \mathbb{E}$ is the inclusion
– an equation $\varphi^v \equiv^v \psi^v : A \to B$ is the equality $\varphi_v = \psi_v : A \to B$;
– an equation $\varphi^c \equiv^c \psi^c : A \to B$ is the equality $\varphi_c = \psi_c : A \to B + \mathbb{E}$.
It should be noted that the coercion from values to computations is the reason why the *subscripts* v and c are used.

Whenever $\mathbb{E}$ is the empty set, in the decorated domain $\mathbf{Set}_{\text{deco}}[\emptyset]$, types are sets, values as well as computations are maps, and equations are equalities.

3.3 The keyword Exception

In a decorated specification, the values are generated from some elementary values, which are the operation symbols of a signature, and the computations are generated from some elementary computations, which are the *exceptions*. Recall that a computation $t^c : X \to Y$ in a decorated specification may raise an exception instead of returning a result of type Y, as would be the case with a term $t : X \to Y$ in a basic specification. Following this idea, we consider that a declaration "Exception e of P", for some type P, adds to the decorated specification a computation with context P and with type 0: indeed, such a computation cannot return a result of type 0, since 0 stands for the empty set, hence it has to raise an exception.

$$P \xrightarrow{\;e^c\;} 0$$

As usual, "Exception e" is a shortcut for "Exception e of 1". In this paper, it is assumed that all the exceptions in a given decorated specification are given once and for all; this assumption could be weaken, in order to take into account the extensibility of exceptions, as in SML.

Moreover, the exceptions form the coprojections of a new kind of sum in the decorated specification ; this *exceptional sum* is studied in section 3.6.

Example 3.2 In the decorated specification $\Sigma_{\mathrm{nat,deco}}$, the values are generated from the operation symbols $z^v : 1 \to \mathrm{Nat}$ and $s^v : \mathrm{Nat} \to \mathrm{Nat}$: for instance, the value p^v in example 3.1. The declaration "Exception e" adds a computation:

$$1 \xrightarrow{\ e^c\ } 0$$

from which other computations will be derived in example 3.6.

3.4 The keyword raise

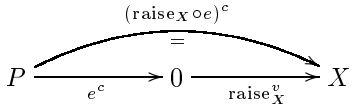
Recall that 0 is an initial type for values and for computations.

Definition 3.3 The keyword raise is the polymorphic value:

$$\mathrm{raise}_X{}^v = [\,]_X{}^v : 0 \longrightarrow X\ .$$

In a decorated specification Σ, let $e^c : P \to 0$ be an exception and X a type. To *raise the exception* e^c *in the type* X is to build the composition:

$$(\mathrm{raise}_X \circ e)^c : P \longrightarrow X\ .$$



The following result proves that the exceptions propagate, as required.

Theorem 3.4 *For every computations* $t^c : X \to Y$ *and* $u^c : P \to 0$*:*

$$t \circ \mathrm{raise}_X \circ u \equiv^c \mathrm{raise}_Y \circ u\ .$$

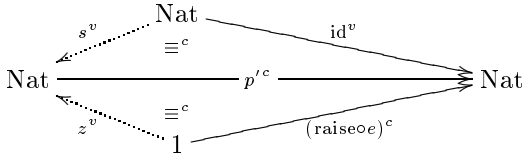
Proof. The initiality property of 0 with respect to the computations proves that $t \circ \mathrm{raise}_X \equiv^c \mathrm{raise}_Y$, and the result follows because $\equiv^c$ is a congruence. □

The next result derives immediately from theorem A.31. Essentially, it has the following meaning. Let $\sigma : \Sigma_1 \to \Sigma_2$ be a morphism of decorated specifications, X_1 a type of Σ_1 and $X_2 = \sigma(X_1)$. Let Σ_1' be made of Σ_1 together with $\mathrm{raise}^v_{X_1} : 0 \to X_1$, and Σ_2' of Σ_2 together with $\mathrm{raise}^v_{X_2} : 0 \to X_2$. Then the unique way to extend σ to $\sigma' : \Sigma_1' \to \Sigma_2'$ is to map $\mathrm{raise}^v_{X_1}$ to $\mathrm{raise}^v_{X_2}$.

Theorem 3.5 *Raising a given exception in a type* X *of a decorated specification* Σ *is natural in* Σ *and* X*.*

Example 3.6 In the decorated specification $\Sigma_{\mathrm{nat,deco}}$, the computation p' is defined as follows (the subscript Nat is omitted):

$$p'^c = (\,\mathrm{case\ id\ of}\ [\,s \Rightarrow \mathrm{id} \mid z \Rightarrow \mathrm{raise} \circ e\,]\,)^c = [\,s \Rightarrow \mathrm{id} \mid z \Rightarrow \mathrm{raise} \circ e\,]^c : \mathrm{Nat} \to \mathrm{Nat}$$

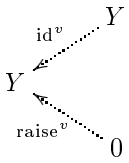


It follows from theorem 3.4 that, for every computation $t^c : \mathrm{Nat} \to \mathrm{Nat}$, the computation $(t \circ p' \circ z)^c$ raises the exception e.

3.5 The case construction over a computation

The case construction over a computation, which is described now, can be used only inside a handle construction (section 3.7), or a record of computations (section 3.8).

Such a construction occurs only with respect to a sum of the form $Y + 0$, for any type Y. It is easy to prove that this sum is isomorphic to Y, with the coprojections raise_Y and id_Y: indeed, the proof involves only values, it is similar to the usual proof, in the basic logic. The subscript Y is often omitted in the illustrations.



As the other sums, this sum $Y = Y + 0$ may be used for building matches of values and matches of computations, and also for building new sums by inverse images of values: the inverse image of the sum $Y = Y + 0$ by a value $u^v : X \to Y$ is simply the sum $X = X + 0$, up to $\sim$, because of the unicity of the inverse images.

Moreover, the inverse image of the sum $Y = Y + 0$ by a *computation* $u^c : X \to Y$ is defined now, with the following meaning: if u raises an exception, then we decide that this exception "comes from" the 0 part of the sum $Y = Y + 0$. More precisely, this inverse image is defined below, using a decorated version of the equivalence of $\equiv$-sums (definition 2.7).

Definition 3.7 Two $\equiv$-sums $\sum_{i=1}^{n}(k_i^v : X_i \to X)$ and $\sum_{i=1}^{n}(k_i'^v : X_i' \to X)$ with the same vertex are *equivalent* if for each i there is an invertible value $s_i^v : X_i' \to X_i$ such that $k_i \circ s_i \equiv^v k_i'$. This is denoted:

$$\sum_{i=1}^{n}(k_i^v : X_i \to X) \sim \sum_{i=1}^{n}(k_i'^v : X_i' \to X) \ .$$

Definition 3.8 Let $u^c : X \to Y$ be a computation. An *inverse image of the sum:*

$$(\mathrm{id}_Y^v : Y \to Y) + (\mathrm{raise}_Y^v : 0 \to Y)$$

by the computation u is an $\equiv$-sum:

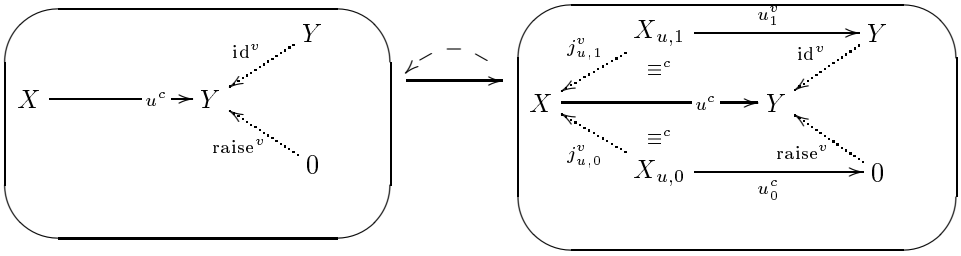
$$(u^c)^{-1}(Y + 0) = (j_{u,1}^v : X_{u,1} \to X) + (j_{u,0}^v : X_{u,0} \to X) \ ,$$

a *value* $u_1^v : X_{u,1} \to Y$ and a *computation* $u_0^c : X_{u,0} \to 0$ such that:

$$u \circ j_{u,1} \equiv^c u_1 \ \text{ and } \ u \circ j_{u,0} \equiv^c \mathrm{raise}_Y \circ u_0 \ .$$

and such that this $\equiv$-sum is $\sim$-unique.

Rules 3.9 *In a decorated domain, every sum* $Y = Y + 0$ *has a chosen inverse image by every computation.*



Some properties of this inverse image are stated now, their proof is easy. The second one shows that there is no ambiguity in our definition: when a computation u^c comes, by coercion, from a value u^v, then the inverse image of $Y = Y + 0$ by the computation u, as defined here, is (up to equivalence) the same as the inverse image of $Y = Y + 0$ by the value u, as defined in section 3.1.

Proposition 3.10

- *Let $u^c, u'^c : X \to Y$ be two computations such that $u \equiv^c u'$, then:*

$$u^{-1}(Y + 0) \sim u'^{-1}(Y + 0) \; .$$

- *Let $u^v : X \to Y$ be a value, then:*

$$(u^v)^{-1}(Y + 0) \sim X + 0 \; .$$

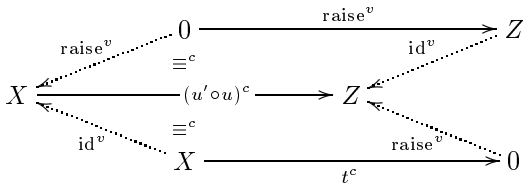
- *Let $u^c : X \to Y$ be a computation and $u'^v : Y \to Z$ a value, then:*

$$(u'^v \circ u)^{-1}(Z + 0) \sim u^{-1}(Y + 0) \; .$$

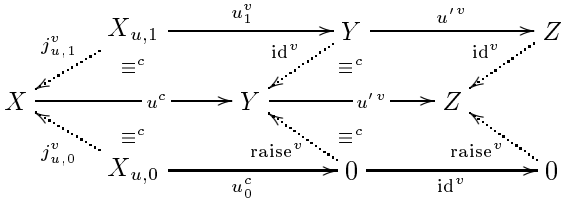
Let us now prove the "back-propagation" of the raising of exceptions, with respect to values.

Proposition 3.11 *Let $u^c : X \to Y$ be a computation and $u'^v : Y \to Z$ a value. If $u' \circ u \equiv^c \mathrm{raise}_Z \circ t$ for some computation $t^c : X \to 0$, then $u \equiv^c \mathrm{raise}_Y \circ t$.*

Proof. On the one hand, from the assumption and from the initiality of 0, we get:



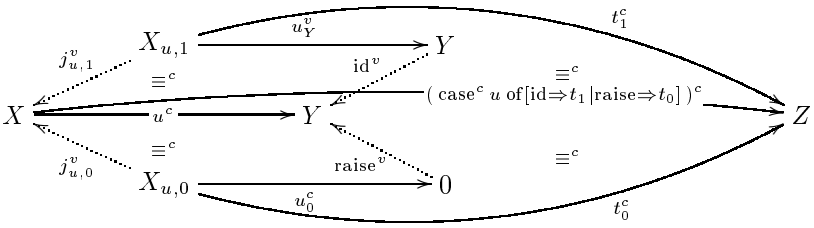
On the other hand:



From the $\sim$-unicity of inverse images, the sums $X = 0 + X$ and $X = X_{u,1} + X_{u,0}$ are equivalent, so that $u_0 \equiv^c t$, hence $u \equiv^c \mathrm{raise}_Y \circ t$, as required. □

Definition 3.12 Let $u^c : X \to Y$ be a computation, and $t_1^c : X_{u,1} \to Z$ and $t_0^c : X_{u,0} \to Z$ two computations. Then the computation "case^c u of $[\,\mathrm{id} \Rightarrow t_1 \mid \mathrm{raise} \Rightarrow t_0\,]$", which is called a *case over computation* construction, is defined as:

$$(\,\mathrm{case}^c\ u^c\ \mathrm{of}\ [\,\mathrm{id} \Rightarrow t_1 \mid \mathrm{raise} \Rightarrow t_0\,]\,)^c \;=\; [\,j_{u,1} \Rightarrow t_1 \mid j_{u,0} \Rightarrow t_0\,]^c \;:\; X \to Z \; .$$



This means that the computation "case^c u of $[\mathrm{id} \Rightarrow t_1 \mid \mathrm{raise} \Rightarrow t_0]$" is characterized, up to $\equiv^c$, by the equations:

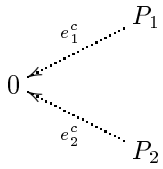
$$\mathrm{case}^c\ u\ \mathrm{of}\ [\mathrm{id} \Rightarrow t_1 \mid \mathrm{raise} \Rightarrow t_0] \circ (j_{u,1} \equiv^c t_1 \quad \text{and} \quad \mathrm{case}^c\ u\ \mathrm{of}\ [\mathrm{id} \Rightarrow t_1 \mid \mathrm{raise} \Rightarrow t_0] \circ (j_{u,0} \equiv^c t_0\ .$$

3.6 The exceptional case construction

Let us come back to the declarations of exceptions. The declarations "Exception e_i of P_i", for $1 \leq i \leq k$, add to the decorated specification a sum of a new kind, called the *exceptional sum*, which allows to test which one among the e_i's is some given exception. In the illustrations, it is assumed that $k = 2$.

Definition 3.13 Let $e^c_i : P_i \to 0$, for $1 \leq i \leq k$, be the exceptions in some given decorated specification. The *exceptional sum* is the cocone with vertex 0 and with coprojections the computations e^c_i's for $1 \leq i \leq k$:

$$\sum_{i=1}^{k} (e^c_i : P_i \to 0)\ .$$



The exceptional sum is quite special: Its coprojections are computations, instead of values; it is used only inside a handle construction (section 3.7). The exceptional sum enjoys a decorated version of only one among the properties of sums, namely the well-behaviour, as follows.

Definition 3.14 Let $u^c : X \to 0$ be a computation, and let $\sum_{i=1}^{k} (e^c_i : P_i \to 0)$ be the exceptional sum. An *inverse image of the exceptional sum by* u^c is an $\sim$-unique $\equiv$-sum:

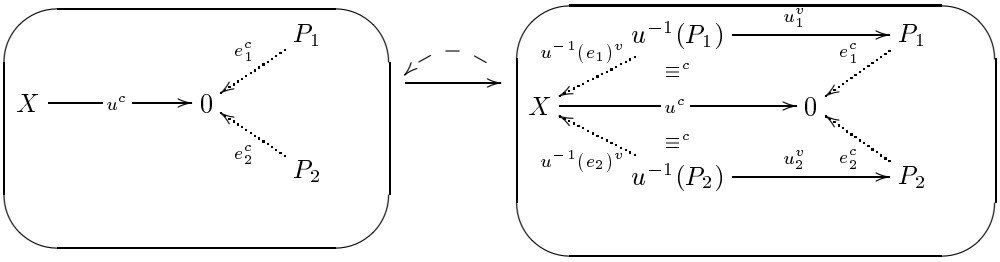
$$\sum_{i=1}^{k} (u^{-1}(e_i)^v : u^{-1}(P_i) \to X)\ ,$$

with *values* $u^v_i : u^{-1}(P_i) \to P_i$ such that for each i:

$$u \circ (u^{-1}(e_i)) \equiv^c e_i \circ u_i\ .$$

Rules 3.15 *In a decorated domain, the exceptional sum has a chosen inverse image by every computation with type* 0.

Here is an illustration of the corresponding existence rule:



Definition 3.16 Let $u^c : X \to 0$ be a computation, $\sum_{i=1}^{k}(e_i^c : P_i \to 0)$ the exceptional sum, and I a subset of $\{1,\ldots,k\}$. For each $i \in I$, let t_i^c be a computation:

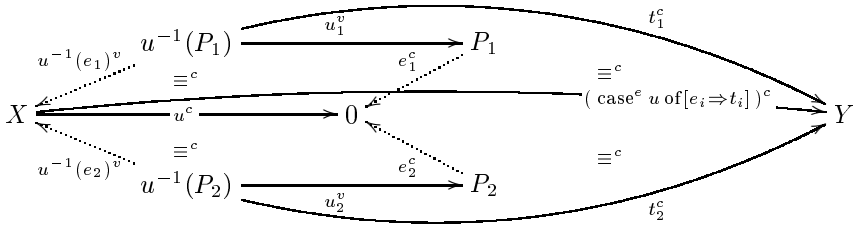
$$t_i^c : u^{-1}(P_i) \to Y \ ,$$

and for each $i \notin I$, let t_i^c be the computation:

$$t_i^c = (\mathrm{raise}_Y \circ u \circ u^{-1}(e_i))^c : u^{-1}(P_i) \to Y \ .$$

Then the computation "case^e u of $[\,e_i \Rightarrow t_i\,]_{i\in I}$", which is called an *exceptional* case construction, is defined as:

$$(\,\mathrm{case}^e\ u^c\ \mathrm{of}\ [\,e_i \Rightarrow t_i\,]_{i\in I}\,)^c \ = \ [\,u^{-1}(e_i) \Rightarrow t_i\,]^c_{1\le i\le k} \ : \ X \to Y \ .$$



This means that the computation "case^e u of $[\,e_i \Rightarrow t_i\,]_{i\in I}$" is characterized, up to $\equiv^c$, by the equations:

$$\mathrm{case}^e\ u\ \mathrm{of}\ [\,e_i \Rightarrow t_i\,]_{i\in I} \circ (u^{-1}(e_i)) \equiv t_i \ \ , \text{for } \ 1 \le i \le k \ .$$

3.7 The keyword handle

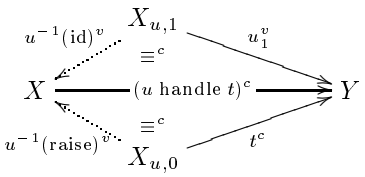
The keyword "handle" has two arguments; for instance, in the term "p' handle $[\,e \Rightarrow z\,]$", the arguments of handle are p' and $[\,e \Rightarrow z\,]$, so that handle behaves, syntactically, as an infix binary operator. Informally, there are two nested kinds of cases in a handling expression "t handle u". The first one tests whether u raises an exception, and when this is true, the second one tests which is the raised exception. The first one is a case distinction over a computation, as in section 3.5, and the second one is an exceptional case distinction, as in section 3.6. Now, the handling construction is easily defined from these two kinds of cases.

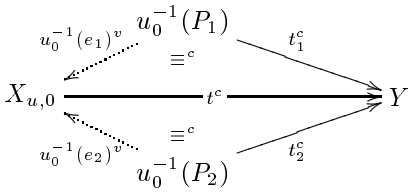
Definition 3.17 Let $u^c : X \to Y$ be a computation, and let $(j^v_{u,1} : X_{u,1} \to X) + (j^v_{u,0} : X_{u,0} \to X)$ be the inverse image of the sum $Y = Y + 0$ by the computation u^c, together with the restrictions $u_1^v : X_{u,1} \to Y$ and $u_0^c : X_{u,0} \to 0$. Let $\sum_{i=1}^{k}(u_0^{-1}(e_i) : u_0^{-1}(P_i) \to X_{u,0})$ be the inverse image of the exceptional sum $\sum_{i=1}^{k} e_i^c : P_i \to 0$ by the computation u_0^c. Let I be a subset of $\{1,\ldots,k\}$ and for each i in I, let $t_i^c : u_0^{-1}(P_i) \to Y$ be a computation. To *handle an exception arising from u^c according to the match* $[\,e_i \Rightarrow t_i\,]_{i\in I}$ is to build the computation:

$$(u\ \mathrm{handle}\ [\,e_i \Rightarrow t_i\,]_{i\in I})^c = (\mathrm{case}^c\ u\ \mathrm{of}\ [\,\mathrm{id}_Y \Rightarrow u_1 \mid \mathrm{raise}_Y \Rightarrow t\,])^c : X \longrightarrow Y \ ,$$

where t^c is the computation:

$$t^c = (\mathrm{case}^e\ u_0\ \mathrm{of}\ [e_i \Rightarrow t_i]_{i\in I})^c : X_{u,0} \longrightarrow Y\ .$$





The following result proves that the exceptions are handled as required; it can be compared to the rules for "handle" in the definition of SML [21].

Theorem 3.18

- *Let $u_1 \equiv^c u_2 : X \to Y$, with (from the $\sim$-unicity of inverse images) the invertible values $s_i^v : u_{1,0}^{-1}(P_i) \to u_{2,0}^{-1}(P_i)$ for $i = 1, \ldots, k$. If $t_{l,i}^c : u_{l,0}^{-1}(P_i) \to Y$ for each $l \in \{1,2\}$ and each $i \in I$ are such that $t_{2,i} \circ s_i \equiv^c t_{1,i}$, then:*

$$u_1\ \mathrm{handle}\ [e_i \Rightarrow t_{1,i}]_{i\in I} \equiv^c u_2\ \mathrm{handle}\ [e_i \Rightarrow t_{2,i}]_{i\in I}\ .$$

- *For every value $u^v : X \to Y$:*

$$u\ \mathrm{handle}\ [e_i \Rightarrow t_i]_{i\in I} \equiv^c u\ .$$

- *For every computation $u^c = \mathrm{raise}_Y \circ u' : X \to Y$ where $u'^c : X \to 0$:*

$$u\ \mathrm{handle}\ [e_i \Rightarrow t_i]_{i\in I} \equiv^c \mathrm{case}^e\ u'\ \mathrm{of}\ [e_i \Rightarrow t_i]_{i\in I}\ .$$

If in addition $u' = e_j \circ u'' : X \to Y$ for some $j \in \{1, \ldots, k\}$ and some value $u''^v : X \to P$, then:

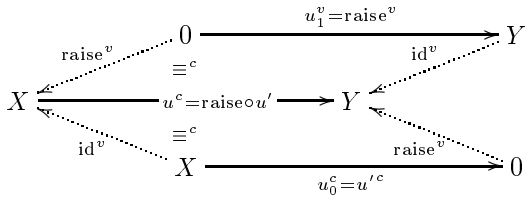
$$u\ \mathrm{handle}\ [e_i \Rightarrow t_i]_{i\in I} \equiv^c t_j \quad \textit{if}\ \ j \in I\ ,$$

$$u\ \mathrm{handle}\ [e_i \Rightarrow t_i]_{i\in I} \equiv^c u \quad \textit{otherwise}\ \ .$$

Proof. The first point is an immediate consequence of the $\equiv$-unicity in the case^c and case^e constructions.

For the second point, since u is a value, the inverse image of $Y = Y + 0$ by u is (up to $\sim$) $X = X + 0$ with $u_1 = u$ and $u_0 = \mathrm{id}_0$. Then, the "raise$_Y$" part of the definition of "u handle $[e_i \Rightarrow t_i]_i$" does not play any rôle, so that the result follows.

For the last point, since $u = \mathrm{raise}_Y \circ u'$, the inverse image of $Y = Y + 0$ by u is (up to $\sim$) $X = 0 + X$ with $u_1^v = \mathrm{raise}_Y^v$ and $u_0^c = u'^c : X \to 0$:



Then, the "id$_Y$" part of the definition of "u handle $[e_i \Rightarrow t_i]_i$" does not play any rôle, so that:

$$u\ \mathrm{handle}\ [e_i \Rightarrow t_i]_i \equiv^c \mathrm{case}^e\ u'\ \mathrm{of}\ [e_i \Rightarrow t_i]_{i\in I}\ .$$

Now, let $u' = e_j \circ u''$, where u''^v is a value. The inverse image of the exceptional sum by u' is such that, up to $\sim$:

$$u'^{-1}(P_j) = X \quad , \quad u'^{-1}(e_j) \equiv \mathrm{id}_X \quad , \quad u'^{-1}(P_i) = 0 \text{ for } i \in I \ , \ i \neq j \ .$$

It follows that:

$$\mathrm{case}^e \ u' \ \mathrm{of} \ [\, e_i \Rightarrow t_i \,]_{i \in I} \equiv^c t_j \ .$$

If $j \in I$ this is the required result. Otherwise, t_j is the default computation:

$$t_j^c = \mathrm{raise}_Y \circ u' \circ (u'^{-1}(e_j)) \equiv^c \mathrm{raise}_Y \circ u' = u \ ,$$

so that, as required:

$$\mathrm{case}^e \ u' \ \mathrm{of} \ [\, e_i \Rightarrow t_i \,]_{i \in I} \equiv^c u \ .$$

□

The next result derives immediately from theorem A.31. Essentially, it has the following meaning. Let $\sigma : \Sigma_1 \to \Sigma_2$ be a morphism of decorated specifications. Let $\sum_{i=1}^k (e_{1,i}^c : P_{1,i} \to 0)$ be the exceptional sum in Σ_1, let $P_{2,i} = \sigma(P_{1,i})$ and $e_{2,i} = \sigma(e_{1,i})$ for each i, so that $\sum_{i=1}^k (e_{2,i}^c : P_{2,i} \to 0)$ is the exceptional sum in Σ_2. Let $u_1^c : X_1 \to Y_1$ in Σ_1 and $u_2^c = \sigma(u_1^c) : X_2 \to Y_2$ in Σ_2. For each i in some subset I of $\{1, \ldots, k\}$, let $t_{1,i}^c : u_1^{-1}(P_{1,i}) \to Y_1$ in Σ_1 and $t_{2,i}^c = \sigma(t_{1,i}^c) : u_2^{-1}(P_{2,i}) \to Y_2$. Let Σ_1' be made of Σ_1 together with $(u_1 \,\mathrm{handle}\, [e_{1,i} \Rightarrow t_{1,i}]_{i \in I})^c : X_1 \to Y_1$, and Σ_2' of Σ_2 together with $(u_2 \,\mathrm{handle}\, [e_{2,i} \Rightarrow t_{2,i}]_{i \in I})^c : X_2 \to Y_2$. Then the unique way to extend σ to $\sigma' : \Sigma_1' \to \Sigma_2'$ is to map $(u_1 \,\mathrm{handle}\, [e_{1,i} \Rightarrow t_{1,i}]_{i \in I})^c$ to $(u_2 \,\mathrm{handle}\, [e_{2,i} \Rightarrow t_{2,i}]_{i \in I})^c$.

Theorem 3.19 *Handling a given exception arising from a computation u^c according to a match $[e_i \Rightarrow t_i]_{i \in I}$ of a decorated specification Σ is natural in Σ, u and the t_i's.*

Example 3.20 In our example, there is only one exception e, and it has no parameter, so that $k = 1$ and $P_1 = 1$. As an example of a proof in the decorated logic, let us now prove that:

$$p'' \equiv^c p \ .$$

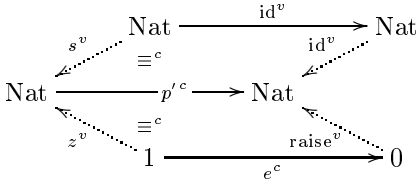
From example 3.1, p^v is the value:

$$p^v = \mathrm{case}\ \mathrm{id}\ \mathrm{of}\ [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow z \,] = [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat} \ .$$

On the other hand, from example 3.6, p'^c is the computation:

$$p'^c = \mathrm{case}\ \mathrm{id}\ \mathrm{of}\ [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow \mathrm{raise} \circ e \,] = [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow \mathrm{raise} \circ e \,] : \mathrm{Nat} \to \mathrm{Nat} \ .$$

It follows that:



Hence, up to $\sim$, the inverse image of the sum $\mathrm{Nat} = \mathrm{Nat} + 0$ by the computation p' is $\mathrm{Nat} = \mathrm{Nat} + 1$, with coprojections s and z, and with $p_1'^v = \mathrm{id}$ and $p_0'^c = e$. Thus:

$$p''^c = p' \,\mathrm{handle}\, [\, e \Rightarrow z \,] \equiv^c \mathrm{case}^c \ p' \ \mathrm{of} \ [\, \mathrm{id} \Rightarrow \mathrm{id} \mid \mathrm{raise} \Rightarrow u \,] \equiv^c [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow u \,] : \mathrm{Nat} \to \mathrm{Nat} \ ,$$

where:

$$u^c = \mathrm{case}^e \ e \ \mathrm{of} \ [\, e \Rightarrow z \,] \equiv^c z : 1 \to \mathrm{Nat} \ .$$

It follows that:

$$p'' \equiv^c [\, s \Rightarrow \mathrm{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat} \ ,$$

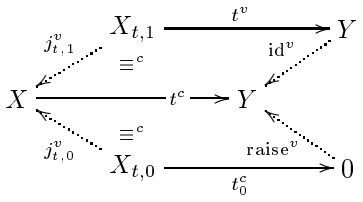
thus, from the $\equiv$-unicity of matches, $p'' \equiv^c p$. Since p is a value, it follows that the computation p'', actually, never raises an exception.

3.8 Records of computations

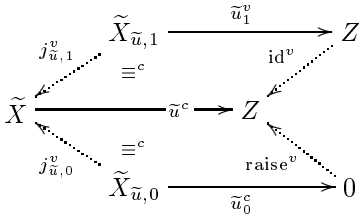
As mentioned in section 3.2, the product of types behaves nicely with respect to values, not with respect to computations. However, usually, some kind of records of computations are used, but, from an operational point of view, the value of these records may depend on the evaluation strategy. Here, we describe this kind of record in a way which corresponds to the SML strategy: record expressions are evaluated from left to right. For simplicity, let us focus on the binary case. let $t^c : X \to Y$ and $u^c : X \to Z$ be two computations, then the computation $(t,u)^c : X \to Y \times Z$ should behave as follows: if t raises an exception e, then (t,u) also raises e, otherwise if u raises an exception e', then (t,u) also raises e', and otherwise (t,u) returns the pair of the results of t and u.

It follows from this description that the definition of $(t,u)^c$ involves two nested cases over computations (as defined in section 3.5), according to t and u respectively, and a record of values.

More precisely, let us consider the inverse image of the sum $Y = Y + 0$ by the computation $t^c : X \to Y$:



Then, let $\widetilde{X} = X_{t,1}$ and $\widetilde{u}^c = u \circ j_{t,1} : \widetilde{X} \to Z$, and let us consider the inverse image of the sum $Z = Z + 0$ by the computation $\widetilde{u}^c$:



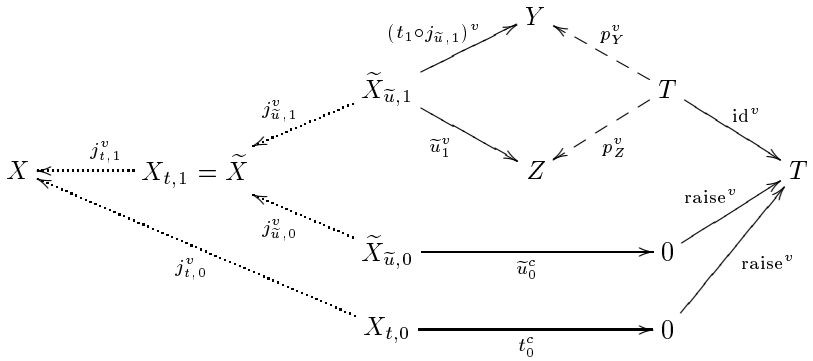
Definition 3.21 Let $t^c : X \to Y$ and $u^c : X \to Z$ be two computations, and let $T = X \times Z$. With the notations above, the *record* of t^c and u^c is the computation $(t,u)^c : X \to T$:

$$(t,u)^c = \mathrm{case}^c\ t\ \mathrm{of}\ [\,\mathrm{id}_Y \Rightarrow w \mid \mathrm{raise}_Y \Rightarrow \mathrm{raise}_T \circ t_0\,]\ :\ X \longrightarrow T\ ,$$

where $w^c : \widetilde{X} \to T$ is the computation:

$$w^c = \mathrm{case}^c\ \widetilde{u}\ \mathrm{of}\ [\,\mathrm{id}_Z \Rightarrow (t \circ j_{\widetilde{u},1}, \widetilde{u}_1)^v \mid \mathrm{raise}_Z \Rightarrow \mathrm{raise}_T \circ \widetilde{u}_0\,]\ :\ \widetilde{X} \longrightarrow T\ .$$

Omitting the records and matches, this definition can be illustrated as:



The equations $p_Y \circ (t,u) \equiv t$ and $p_Z \circ (t,u) \equiv u$ do *not* hold here, in any relevant decorated way. This means, as is well known, that the usual rules for eliminating products do *not* hold for the products of computations.

However, some version of the usual compatibility properties still hold for records of two computations.

Proposition 3.22 *Let $t^c : X \to Y$ and $u^c : X \to Z$ be two computations, and let $T = Y \times Z$.*

- *Let $t' \equiv^c t$ and $u' \equiv^c u$, then:*
$$(t',u') \equiv^c (t,u) : X \to T \ .$$
- *Let $w^v : X' \to X$ be a* value*, then:*
$$(t,u) \circ w \equiv^c (t \circ w, u \circ w) : X' \to T \ .$$

The following result proves that the record $(t,u)^c$ has the properties which are required for formalising a left-to-right evaluation strategy.

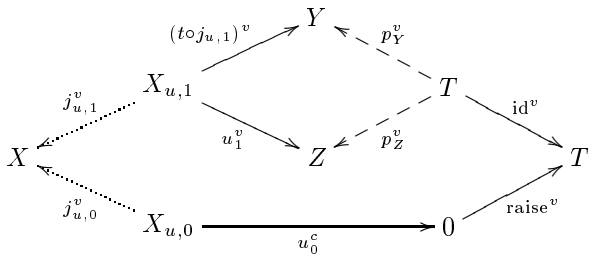
Theorem 3.23 *Let $t^c : X \to Y$ and $u^c : X \to Z$ be two computations, and let $T = Y \times Z$.*

- *When t and u are values, then $(t,u)^c \equiv^c (t,u)^v$.*
- *When $t^c \equiv^c \mathrm{raise}_Y \circ t'$ for some computation t'^c, then $(t,u)^c \equiv^c \mathrm{raise}_T \circ t'$.*
- *When t is a value and $u^c \equiv^c \mathrm{raise}_Z \circ u'$ for some computation u'^c, then $(t,u)^c \equiv^c \mathrm{raise}_T \circ u'$.*

Proof. For the first point, since t and u are values, the "raise_Y" and "raise_Z" parts of the definition of $(t,u)^c$ do not play any rôle and (up to $\sim$) $j_{t,1} = \mathrm{id}_X$, $t_1 = t$, $\widetilde{X} = X$, $\widetilde{u} = u$, $j_{\widetilde{u},1} = \mathrm{id}_X$, $\widetilde{u}_1 = u$, so that:
$$(t,u)^c \equiv^c (t_1 \circ j_{\widetilde{u},1}, \widetilde{u}_1)^v \equiv^v (t,u)^v \ .$$
For the second point, since $t \equiv^c \mathrm{raise}_Y \circ t'$, the "$\mathrm{id}_Y$" part of the definition of $(t,u)^c$ does not play any rôle and (up to $\sim$) $j_{t,0} = \mathrm{id}_X$ and $t_0 = t'$, so that:
$$(t,u)^c \equiv^c \mathrm{raise}_T \circ t_0 \equiv^c \mathrm{raise}_T \circ t' \ .$$
For the last point, since t is a value and $u \equiv^c \mathrm{raise}_Z \circ u'$, the "$\mathrm{raise}_Y$" and "$\mathrm{id}_Z$" parts of the definition of $(t,u)^c$ do not play any rôle and (up to $\sim$) $j_{t,1} = \mathrm{id}_X$, $t_1 = t$, $\widetilde{X} = X$, $\widetilde{u} = u$, $j_{\widetilde{u},0} = \mathrm{id}_X$, $\widetilde{u}_0 = u'$, so that:
$$(t,u)^c \equiv^c \mathrm{raise}_T \circ \widetilde{u}_0 \equiv^c \mathrm{raise}_T \circ u' \ .$$
□

3.9 Records of a value and a computation

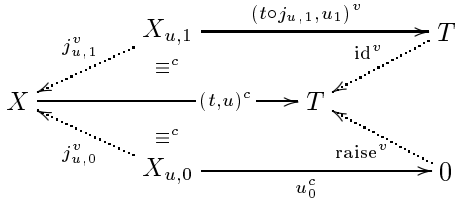
Let us now focus on an intermediate situation, where $t^v : X \to Y$ is a *value* and $u^c : X \to Z$ is a *computation*. Then the record $(t,u)^c$ gets simpler, since the sum $t^{-1}(Y+0)$ is $X+0$, so that $\widetilde{X} = X$, $t_1 = t$ and $\widetilde{u} = u$:
$$(t^v, u^c)^c = \mathrm{case}^c\ u \text{ of } [\,\mathrm{id}_Z \Rightarrow (t \circ j_{u,1}, u_1)^v \mid \mathrm{raise}_Z \Rightarrow \mathrm{raise}_T \circ u_0\,] \ :\ X \longrightarrow T \ .$$



Proposition 3.24 *Let $t^v : X \to Y$ be a value and $u^c : X \to Z$ a computation, then:*

$$((t,u)^c)^{-1}(Y+0) \sim u^{-1}(Z+0)\ .$$

Proof. From the definition of $(t,u)^c$ when t is a value, we get:



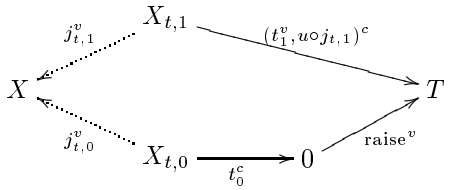
Then, the result follows from the $\sim$-unicity of inverse images. □

Clearly, the other way round, the record $(t^v, u^c)^c$ could be first defined as above:

$$(t^v, u^c)^c = \mathrm{case}^c\ u\ \mathrm{of}\ [\,\mathrm{id}_Z \Rightarrow (t \circ j_{u,1}), u_1)^v \mid \mathrm{raise}_Z \Rightarrow \mathrm{raise}_T \circ u_0\,]\ :\ X \longrightarrow T\ .$$

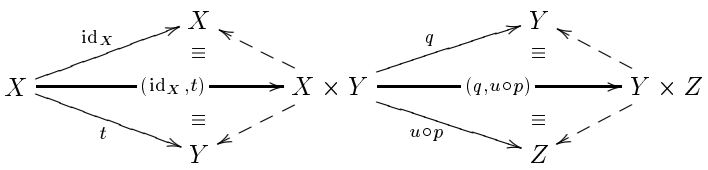
and then $(t^c, u^c)^c$ could be defined as:

$$(t,u)^c = [j_{t,1} \Rightarrow (t_1^v, u \circ j_{t,1})^c) \mid j_{t,0} \Rightarrow \mathrm{raise}_T \circ t_0\,]^c\ .$$



There is still another way to define a record of computations from a record of a value of a computation, which does not involve any sum. It is based on the remark that, in the basic logic, a record (t,u) can be decomposed as follows, where p and q are the projections from $X \times Y$:

$$(t,u) \equiv (q, u \circ p) \circ (\mathrm{id}_X, t)$$



In the decorated logic, the following result proves that a similar property allows to express a record of computations as the composition of two records of a value of a computation.

Theorem 3.25 *Let $t^c : X \to Y$ and $u^c : X \to Z$ be two computations, then:*

$$(t,u)^c \equiv^c (q^v, u \circ p)^c \circ (\mathrm{id}_X^v, t)^c : X \to Y \times Z \ .$$

Proof. Since (omitting some subscripts):

$$(t,u)^c = [\, j_{t,1} \Rightarrow (t_1^v, u \circ j_{t,1})^c) \mid j_{t,0} \Rightarrow \mathrm{raise} \circ t_0 \,]^c \ ,$$

the result will follow from:

$$(q, u \circ p) \circ (\mathrm{id}_X, t) \circ j_{t,1} \equiv^c (t_1^v, u \circ j_{t,1}) \ \text{ and } \ (q, u \circ p) \circ (\mathrm{id}_X, t) \circ j_{t,0} \equiv^c \mathrm{raise} \circ t_0 \ .$$

On the one hand, since $j_{t,1}$ is a value:

$$(\mathrm{id}_X, t) \circ j_{t,1} \equiv^c (j_{t,1}, t \circ j_{t,1})$$

and since $t \circ j_{t,1} \equiv t_1$, which is a value, we get:

$$(\mathrm{id}_X, t) \circ j_{t,1} \equiv^c (j_{t,1}, t_1)^v \ .$$

Again, since $(j_{t,1}, t_1)$ is a value:

$$(q, u \circ p) \circ (\mathrm{id}_X, t) \circ j_{t,1} \equiv^c (q \circ (j_{t,1}, t_1), u \circ p \circ (j_{t,1}, t_1)) \equiv^c (t_1, u \circ j_{t,1}) \ ,$$

which proves the first equation.

On the other hand, since $j_{t,0}$ is a value:

$$(\mathrm{id}_X, t) \circ j_{t,0} \equiv^c (j_{t,0}, t \circ j_{t,0}) \ .$$

Since $t \circ j_{t,0} \equiv^c \mathrm{raise}_Y \circ t_0$, according to theorem 3.23:

$$(\mathrm{id}_X, t) \circ j_{t,0} \equiv^c \mathrm{raise}_{X \times Y} \circ t_0 \ .$$

Then:

$$(q, u \circ p) \circ (\mathrm{id}_X, t) \circ j_{t,0} \equiv^c (q, u \circ p) \circ \mathrm{raise}_{X \times Y} \circ t_0 \ ,$$

and since exceptions propagate (theorem 3.4), this yields:

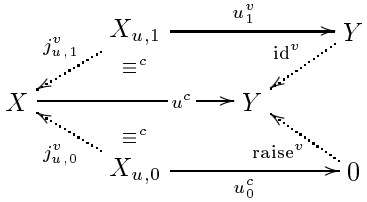
$$(q, u \circ p) \circ (\mathrm{id}_X, t) \circ j_{t,0} \equiv^c \mathrm{raise}_T \circ t_0 \ ,$$

which proves the second equation. □

The values in the records that occur in theorem 3.25 are respectively an identity id_X and a projection q. Thus, from an operational point of view, each of these records correspond to keeping some information in memory, while dealing with some computation.

Moreover, the record of a value and a computation enjoys a decorated property for the elimination of products, in contrast with a record of two computations. First, let us introduce a new decoration *ne* for an "equation when the results are non-exceptional".

Definition 3.26 Let $u^c : X \to Y$ be a computation and $t^v : X \to Y$ a value. Let us consider the inverse image of $Y + 0$ by u:



Then an *equation when results are non-exceptional* between u and t is written $u \equiv^{ne} t$ and it means that:

$$u^c \equiv^{ne} t^v \iff u \circ j_{u,1} \equiv^v t \circ j_{u,1} \iff u_1 \equiv^v t \circ j_{u,1} \ .$$

This decoration *ne* for equations has to be handled with care: it does not generate a congruence, not even an equivalence relation. Here are some of its few properties.

Proposition 3.27

- *An equation when results are non-exceptional between values is equivalent to a value equation:*

$$u^v \equiv^{ne} t^v \iff u^v \equiv^v t^v \ .$$

- *Values can be substituted in an equation when results are non-exceptional: if t'^v is a value, and t' and t are consecutive, then:*

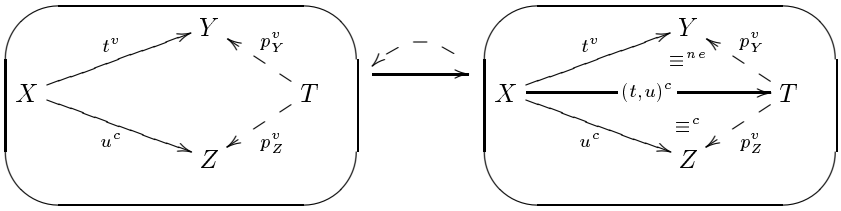
$$u^c \equiv^{ne} t^v \Rightarrow (u \circ t')^c \equiv^{ne} (t \circ t')^v \ .$$

Now, the decoration *ne* can be used for decorating the elimination property of products, as follows.

Proposition 3.28 *For every value $t^v : X \to Y$ and every computation $u^c : X \to Z$, the record $(t^v, u^c)^c : X \to T$, where $T = Y \times Z$, is the $\equiv^c$-unique computation from X to T such that:*

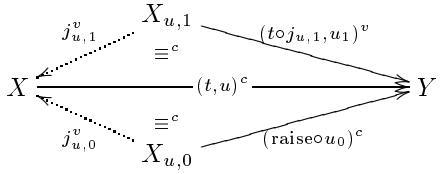
$$p_Y \circ (t,u) \equiv^{ne} t \quad \textit{and} \quad p_Z \circ (t,u) \equiv^c u \ .$$

The existence part of this property is illustrated as follows.

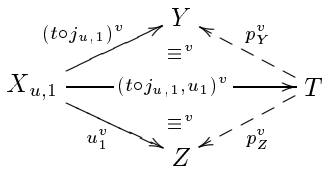


Proof. Since t is a value, as above:

$$(t^v, u^c)^c = \mathrm{case}^c\ u\ \mathrm{of}\ [\,\mathrm{id} \Rightarrow (t \circ j_{u,1}, u_1)^v \mid \mathrm{raise} \Rightarrow \mathrm{raise} \circ u_0\,] \ : \ X \longrightarrow T \ .$$



where



Thus, on the one hand:

$$p_Y \circ (t,u)^c \circ j_{u,1} \equiv^c p_Y \circ (t \circ j_{u,1}, u_1)^v \equiv^c t \circ j_{u,1} \ .$$

Since t is a value, from proposition 3.24, $((t,u)^c)^{-1}(T+0) \sim u^{-1}(Z+0)$.
Since p_Y is a value, from proposition 3.10, $(p_Y \circ (t,u)^c)^{-1}(Y+0) \sim ((t,u)^c)^{-1}(T+0)$.
Let $w^c = p_Y \circ (t,u)^c$, we have proved that $w^{-1}(Y+0) \sim u^{-1}(Z+0)$, so that $j_{u,1}$ can be replaced by $j_{w,1}$ in the equation above, leading to:

$$w \circ j_{w,1} \equiv^c t \circ j_{w,1} \ ,$$

which means that $w \equiv^{ne} t$: this is the first congruence to prove.

On the other hand:

$$p_Z \circ (t,u)^c \circ j_{u,1} \equiv^c p_Z \circ (t \circ j_{u,1}, u_1)^v \equiv^c u_1 \equiv^c u \circ j_{u,1}$$

and:

$$p_Z \circ (t,u)^c \circ j_{u,0} \equiv^c p_Z \circ \mathrm{raise}_T \circ u_0 \equiv^c \mathrm{raise}_Z \circ u_0 \equiv^c u \circ j_{u,0} \ ,$$

so that $p_Z \circ (t,u)^c \equiv^c u$, which is the second congruence to prove.

Now, let us prove the $\equiv^c$-unicity. From its definition, $(t,u)^c$ is the $\equiv^c$-unique computation such that:

$$p_Y \circ (t,u)^c \circ j_{u,1} \equiv^c t \circ j_{u,1} \ \text{ and } \ p_Z \circ (t,u)^c \circ j_{u,0} \equiv^c u_1 \ \text{ and } \ (t,u)^c \circ j_{u,1} \equiv^c \mathrm{raise}_T \circ u_0 \ .$$

Now, let $w^c : X \to Y$ be such that:

$$p_Y \circ w \equiv^{ne} t \ \text{ and } \ p_Z \circ w \equiv^c u \ .$$

From proposition 3.10, since $p_Z \circ w \equiv^c u$ where p_Z is a value, we get $w^{-1}(T+0) \sim u^{-1}(Z+0)$. Thus, the first assumption on w means that:

$$p_Y \circ w \circ j_{u,1} \equiv^c t \circ j_{u,1} \ .$$

From the second assumption on w, and the fact that $u \circ j_{u,1} \equiv^c u_1$, we get:

$$p_Z \circ w \circ j_{u,1} \equiv^c u_1 \ .$$

From the second assumption on w, and the fact that $u \circ j_{u,0} \equiv^c \mathrm{raise} \circ u_0$, we get:

$$p_Z \circ w \circ j_{u,0} \equiv^c \mathrm{raise}_Z \circ u_0 \ .$$

According to proposition 3.11, the raising of exceptions can be "propagated back" along values, so that:

$$w \circ j_{u,0} \equiv^c \mathrm{raise}_T \circ u_0 \ .$$

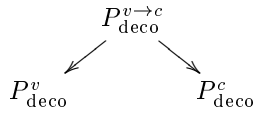
Finally, the three equations that characterise $(t,u)^c$ up to $\equiv^c$ are satisfied by w^c, which proves that $w \equiv^c (t,u)^c$, as required. □

3.10 The decorated logic

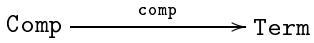
The decorated propagator P_{deco} takes into account all these properties. First, in order to deal with values, P_{deco} contains a copy P^v_{deco} of the basic propagator P_{basic}.

Then, in order to build computations and equations between them, P_{deco} also contains a copy P^c_{deco} of the part (called P_{eq} in section 2.1) of the basic propagator P_{basic} which deals with terms and equations.

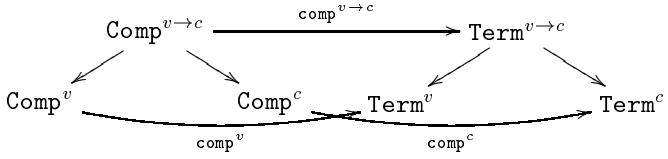
Since every value can be considered as a computation, in a decorated specification every value *may* (and not *must*) be coerced to a computation, there is in P_{deco} another copy $P^{v\to c}_{\mathrm{deco}}$ of P_{eq}, with a span:



The morphisms in this span are called *coercions*. For instance, the composition in $\mathcal{S}_{\mathrm{basic}}$:



gives rise in $\mathcal{S}_{\mathrm{deco}}$ to the commutative diagram:



Moreover, since the types are not decorated, the points $\texttt{Type}^c$, $\texttt{Type}^{v\to c}$ and $\texttt{Type}^c$ are identified in a unique point Type.

In a decorated domain, every value *is* a computation, so that the coercion from $S^{v\to c}$ to S^v gets invertible in $\overline{\mathcal{S}}_{\text{deco}}$.

The construction "case u of $[\,j_i \Rightarrow t_i\,]_i$" where the t_i's are computations and u is a value builds another link between P^v_{deco} and P^c_{deco}. The constructions case^c and case^e, for cases over a computation and exceptional cases, are also added. Then, the raise and handle constructions, as well as the records of computations, are defined from the existing decorated sums and products, following the previous sections.

If the sums were forgotten, from section 3.9, the records of computations could be defined from the records of a value and a computation, which in turn satisfy a decorated product property (proposition 3.28). Hence, roughly speaking, when the congruence is the equality, a decorated domain corresponds to a morphism, that is the identity on objects, from a category with products to a category with some "kind of" products; more precisely, it corresponds to a *Freyd category*, in the sense of [25].

The *decorated logic* is the diagrammatic logic associated to the propagator P_{deco}.

3.11 Decorated models

Definition 3.29 Let Σ_{deco} be a decorated specification and Δ_{deco} a decorated domain. The set of *decorated models of* Σ_{deco} *with values in* Δ_{deco} is defined as in appendix A:

$$\text{Mod}_{\text{deco}}(\Sigma_{\text{deco}}, \Delta_{\text{deco}}) = \text{Hom}_{\mathbf{Dom}(P_{\text{deco}})}(\text{F}_{P_{\text{deco}}}(\Sigma_{\text{deco}}), \Delta_{\text{deco}}) \simeq \text{Hom}_{\mathbf{Spec}(P_{\text{deco}})}(\Sigma_{\text{deco}}, \text{G}_{P_{\text{deco}}}(\Delta_{\text{deco}}))\ .$$

For each set $\mathbb{E}$, the decorated domain $\mathbf{Set}_{\text{deco}}[\mathbb{E}]$ is defined in section 3.2: it is some kind of "decorated domain of sets", which depends on $\mathbb{E}$; hence, a model of a decorated specification with values in some $\mathbf{Set}_{\text{deco}}[\mathbb{E}]$ is called a "set-valued" model.

Definition 3.30 Let Σ_{deco} be a decorated specification. For each set $\mathbb{E}$, a *set-valued decorated model of* Σ_{deco} *with set of exceptions* $\mathbb{E}$ is a decorated model of Σ_{deco} with values in the decorated domain $\mathbf{Set}_{\text{deco}}[\mathbb{E}]$.

Let M be a set-valued model of Σ_{deco} with set of exceptions $\mathbb{E}$, then it must interpret 0 as the empty set, and each exception $e^c_i : P_i \to 0$ as a map $\varepsilon_i : M(P_i) \to \mathbb{E}$, since $\emptyset + \mathbb{E} = \mathbb{E}$. The exceptional sum must be interpreted as a sum $\varepsilon_i : M(P_i) \to \mathbb{E}$.

3.12 The undecoration morphism

The decorated propagator P_{deco} has been built by classifying the features of the basic propagator P_{basic}. The other way round, by forgetting the decorations, one gets a morphism from P_{deco} to P_{basic}.

Definition 3.31 The *undecoration morphism*:

$$\delta : P_{\text{deco}} \longrightarrow P_{\text{basic}}$$

forgets about the decorations, which means that S^d is mapped to S for every point S^d of $\mathcal{S}_{\text{basic}}$, and s^d is mapped to s for every arrow s^d of $\mathcal{S}_{\text{basic}}$.

Since it is a morphism of propagators, the undecoration morphism gives rise to an adjunction between the basic specifications and the decorated specifications, and to an adjunction between the basic domains and the decorated domains. Each decorated specification Σ_{deco} freely generates a basic specification $\Sigma_{\text{basic}} = \text{F}_\delta(\Sigma_{\text{deco}})$, simply by forgetting the decorations.

Decorated products and sums in Σ_{deco} give rise to ordinary products and sums in Σ_{basic}. It follows that decorated records and cases give rise to ordinary records and cases. The types 1 and 0 with respect to values, in Σ_{deco}, give rise to the types 1 and 0 in Σ_{basic}. The sum $Y = Y + 0$ gives rise to the sum $Y = Y + 0$, every computation $u^c : X \to Y$ gives rise to a term $u : X \to Y$, so that the sum $X = X_{u,1} + X_{u,0}$ gives rise to the sum $X = X + 0$. The exceptional sum in Σ_{deco} gives rise in Σ_{basic} to a sum with vertex 0. Hence, the undecoration morphism allows to get a simplified view on the terms and equations, by forgetting all the decorations. It allows to get a simplified view on the proofs, since the image of a proof in the decorated logic is a proof in the basic logic. This can be stated as:

"A proof in Σ_{deco} is a proof in Σ_{basic} which can be decorated".

This yields a two-step method for checking a proof in the decorated logic: first, the proof without its decorations must be valid in the basic logic, then it must be feasible to add the decorations in a way that is valid in the decorated logic.

However, this simplified view "does not preserve the models": for instance a constant exception $e^c : 1 \to 0$ in Σ_{deco} gives rise in Σ_{basic} to a term $e : 1 \to 0$, which has no set-valued interpretation. More precisely, it follows from proposition A.34 that, for each decorated specification Σ_{deco} and each basic domain Δ_{basic}, there is a natural bijection:

$$\text{Mod}_{\text{deco}}(\Sigma_{\text{deco}}, \text{G}_\delta(\Delta_{\text{basic}})) \simeq \text{Mod}_{\text{basic}}(\Sigma_{\text{basic}}, \Delta_{\text{basic}}) \ .$$

When $\Delta_{\text{basic}} = \mathbf{Set}$, it is easy to check that $\text{G}_\delta(\mathbf{Set})$ is the trivial decorated domain of sets $\mathbf{Set}_{\text{deco}}[\emptyset]$, as described in section 3.2. But the models of interest are the models of Σ_{deco} with values in $\mathbf{Set}_{\text{deco}}[\mathbb{E}]$ for some non-empty set $\mathbb{E}$.

The models of Σ_{basic} form the *naïve semantics* of Σ_{deco}, in the sense of [15]. In section 4, another morphism from the decorated propagator to a "classical" (non-decorated) logic is defined, which "does preserve the models".

Example 3.32 In $\Sigma_{\text{nat,basic}} = \text{F}_\delta(\Sigma_{\text{nat,deco}})$, there is a term $e : 1 \to 0$, so that this basic specification has no set-valued model.
In $\Sigma_{\text{nat,deco}}$, the computations (involving the three kinds of decorated cases):

$$p'^c = \text{case id of } [\, s \Rightarrow \text{id} \mid z \Rightarrow \text{raise} \circ e \,] : \text{Nat} \to \text{Nat}$$

and:

$$p''^c = p' \text{ handle } [\, e \Rightarrow z \,] = \text{case}^c \ p' \text{ of } [\, \text{id} \Rightarrow \text{id} \mid \text{raise} \Rightarrow u \,] : \text{Nat} \to \text{Nat}$$

where:

$$u^c = \text{case}^e \ e \text{ of } [\, e \Rightarrow z \,] : 1 \to \text{Nat}$$

give rise in $\Sigma_{\text{nat,basic}}$ to the terms (involving three times the basic case distinction):

$$p' = \text{case id of } [\, s \Rightarrow \text{id} \mid z \Rightarrow \text{raise} \circ e \,] : \text{Nat} \to \text{Nat}$$

and:

$$p'' = \text{case } p' \text{ of } [\, \text{id} \Rightarrow \text{id} \mid \text{raise} \Rightarrow u \,] : \text{Nat} \to \text{Nat}$$

where:

$$u = \text{case } e \text{ of } [\, e \Rightarrow z \,] : 1 \to \text{Nat} \ .$$

It can be proved that $p'' \equiv p$ in $\Sigma_{\text{nat,basic}}$, by undecorating the proof from example 3.20. This means that the inverse image of the sum $\text{Nat} = \text{Nat} + 0$ by p' has to be chosen as $\text{Nat} = \text{Nat} + 1$ with coprojections s and z, instead of $\text{Nat} = \text{Nat} + 0$ with coprojections id and $[\ ]$. Indeed, both sums are equivalent in $\Sigma_{\text{nat,basic}}$, but only the first one can be properly decorated.

4 A logic with explicit exceptions

In this section, exceptions are considered in an *explicit* way, which means that there is a type of exceptions E which formalizes the set of exceptions, and that E appears in the type of a term, as soon as this term may raise an exception. This corresponds to the *explicit* logic, which has no decorations. It is an enrichment of the basic logic with a distinguished type E. The *expansion* morphism χ from the decorated logic to the explicit logic is described in this section. It is proved in theorem 4.6 that, for any decorated specification Σ_{deco}, the freely generated explicit specification $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$ makes *explicit* the meaning of the exception mechanism, in the sense that the set-valued decorated models of Σ_{deco}, can be identified with the set-valued basic models of $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$. Thus, our direct semantics is equivalent to the monadic semantics.

An *explicit specification* is a P_{expl}-specification, an *explicit domain* is a P_{expl}-domain.

4.1 The explicit logic

Definition 4.1 The *explicit propagator* $P_{\mathrm{expl}} : \mathcal{S}_{\mathrm{expl}} \to \overline{\mathcal{S}}_{\mathrm{expl}}$ is the basic propagator P_{basic} together with an arrow:

$$E : \mathbb{I} \to \mathtt{Type}$$

The *explicit logic* is the diagrammatic logic associated to the propagator P_{expl}.

Hence, an explicit specification (resp. an explicit domain) is a basic specification (resp. a basic domain) together with a distinguished type, still denoted E.

The *expansion* morphism:

$$\chi : P_{\mathrm{deco}} \to P_{\mathrm{expl}}$$

is defined below. Actually, χ is a generalized morphism, in the sense of section A.2: it maps P_{deco} to some enrichment of P_{expl}, which is equivalent to P_{expl}.

From corollary A.13, χ is characterized by a generalized morphism of projective sketches:

$$\chi : \mathcal{S}_{\mathrm{deco}} \to \mathcal{S}_{\mathrm{expl}}$$

such that, for each active deduction rule s^{-1} of $P_{\mathrm{deco}}(s)$, the arrow $\chi(s)$ in $\mathcal{S}_{\mathrm{expl}}$ becomes invertible in $\overline{\mathcal{S}}_{\mathrm{expl}}$, via P_{expl}. Moreover, as explained in corollary A.20, since our goal is to describe the explicit specification $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$ for any decorated specification Σ_{deco}, we do not have to describe directly χ, but rather the contravariant functor (where $\mathrm{Yon}_{\mathrm{expl}} = \mathrm{Yon}_{\mathcal{S}_{\mathrm{expl}}}$ and $\mathrm{Yon}_{\mathrm{deco}} = \mathrm{Yon}_{\mathcal{S}_{\mathrm{deco}}}$):

$$\mathrm{W} = \mathrm{Yon}_{\mathrm{expl}} \circ \chi \sim \mathrm{F}_\chi \circ \mathrm{Yon}_{\mathrm{deco}} : \mathcal{S}_{\mathrm{deco}} \nrightarrow \mathbf{Real}(\mathcal{S}_{\mathrm{expl}}) \ ,$$

since $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$ is isomorphic to a colimit of images of W. The contravariant functor W must be such that, for each active deduction rule s^{-1} of $P_{\mathrm{deco}}(s)$, the morphism $\mathrm{W}(s)$ of P_{expl}-specifications becomes an isomorphism between the freely generated P_{expl}-domains.

Now, the contravariant functor W is described, in a progressive way. Each explicit specification contains one copy of the distinguished type E, but in the illustrations below, for clarity, any number of copies of E may appear: from no copy, when E is not used, to several copies, when this improves the readability. Let Σ_{deco} denote a decorated specification, and $\Sigma_{\mathrm{expl}} = \mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$ the freely generated explicit specification.

4.2 Types, terms, equations

A type X in Σ_{deco} remains a type X in $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$, so that the explicit specification $\mathrm{W}(\mathtt{Type})$ is simply made of one type:

$$\mathrm{W}(\mathtt{Type}) = \mathrm{Yon}_{\mathrm{expl}}(\mathtt{Type}) \;=\; \boxed{X}$$

For each type X distinct from E, the coprojections from X and E to the sum $X + E$ are denoted respectively i_X and r_X:

$$X \xrightarrow{\;i_X\;} X + E \xleftarrow{\;r_X\;} E$$

A decorated term $t^d : X \to Y$ in Σ_{deco} remains a term in $\mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$, but its type and context depend on its decoration d.

A value $t^v : X \to Y$ becomes a term $t_v : X \to Y$, so that the explicit specification $\mathrm{W}(\mathtt{Term}^v)$ is:

$$\mathrm{W}(\mathtt{Term}^v) = \mathrm{Yon}_{\mathrm{expl}}(\mathtt{Term}) \;=\; \boxed{\begin{array}{c} X \\ \downarrow{\scriptstyle t_v} \\ Y \end{array}}$$

A computation $t^c : X \to Y$ becomes a term $t_c : X \to Y + E$, so that the explicit specification $\mathrm{W}(\mathtt{Term}^c)$ is:

$$\mathrm{W}(\mathtt{Term}^c) \;=\; \boxed{\begin{array}{ccccc} X & & & & \\ & \searrow^{t_c} & & & \\ Y & \xrightarrow[i_Y]{} & Y + E & \xleftarrow[r_Y]{} & E \end{array}}$$

Moreover:

$$\mathrm{W}(\mathtt{Term}^{v\to c}) \;=\; \boxed{\begin{array}{ccccc} X & & & & \\ \downarrow{\scriptstyle t_v} & \searrow^{t_c} & & & \\ Y & \xrightarrow[i_Y]{} & Y + E & \xleftarrow[r_Y]{} & E \end{array}}$$

The coercions from $\mathtt{Term}^{v\to c}$ to $\mathtt{Term}^v$ and to $\mathtt{Term}^c$ are mapped by W on the injections of $\mathrm{W}(\mathtt{Term}^v)$ and $\mathrm{W}(\mathtt{Term}^c)$ in $\mathrm{W}(\mathtt{Term}^{v\to c})$. The injection of $\mathrm{W}(\mathtt{Term}^v)$ in $\mathrm{W}(\mathtt{Term}^{v\to c})$ becomes an isomorphism on the freely generated explicit domains, as required: every value may be seen as a computation.

The morphisms $\mathrm{W}(\mathtt{type}^d) : \mathrm{W}(\mathtt{Type}) \to \mathrm{W}(\mathtt{Term}^d)$, for each decoration d, map X to X.

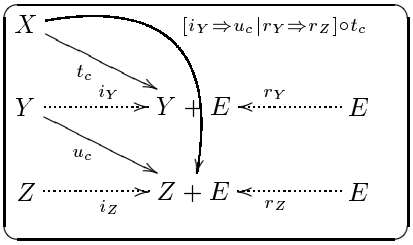
The morphisms $\mathrm{W}(\mathtt{context}^d) : \mathrm{W}(\mathtt{Type}) \to \mathrm{W}(\mathtt{Term}^d)$, for each decoration d, map X to Y.

The values are composed in the usual way:

$$\mathrm{W}(\mathtt{Comp}^v) = \mathrm{Yon}_{\mathrm{expl}}(\mathtt{Comp}) \;=\; \boxed{\begin{array}{ccc} X & & \\ \downarrow{\scriptstyle t_v} & \searrow^{u_v \circ t_v} & \\ Y & & \\ \downarrow{\scriptstyle u_v} & & \\ Z & & \end{array}}$$

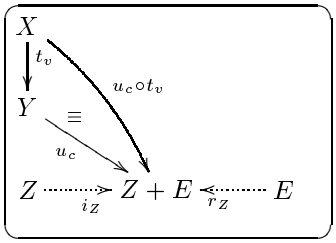
The computations are composed in the Kleisli way (in the monads approach, this composition is part of the description of the monad of exceptions):

$\mathrm{W}(\mathtt{Comp}^c) \;=$



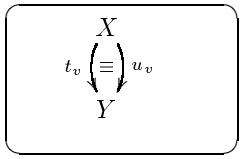
It follows from the description of $\mathrm{W}(\mathtt{Term}^{v \to c})$ and $\mathrm{W}(\mathtt{Comp}^c)$ that the composition of a value t^v and a computation u^c is such that:

$$(u \circ t)^c = [i_Y \Rightarrow u_c \mid r_Y \Rightarrow r_Z] \circ t_c \equiv [i_Y \Rightarrow u_c \mid r_Y \Rightarrow r_Z] \circ i_Y \circ t_v \equiv u_c \circ t_v$$

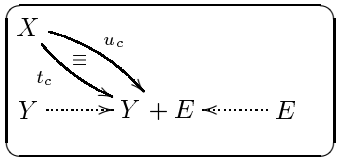


About equations, the description of $\mathrm{W}(\mathtt{Eq}^v)$ and $\mathrm{W}(\mathtt{Eq}^c)$ is straightforward.

$\mathrm{W}(\mathtt{Eq}^v) = \mathrm{Yon}_{\mathrm{expl}}(\mathtt{Eq}) \;=$



$\mathrm{W}(\mathtt{Eq}^c) \;=$



According to [4], an extensive category with products is distributive, and the coprojections are monic in a distributive category. Similarly, the coprojections are monic, up to $\equiv$, in the explicit logic. Hence, the expansion preserves the property that a computation equation between values is equivalent to a value equation.

4.3 Products and sums

Every product $\prod_{i=1}^n (p_i^v : Y \to Y_i)$ in Σ_{deco} is expanded as a product $\prod_{i=1}^n (p_{i,v} : Y \to Y_i)$ in Σ_{expl}, so that the terminal type 1 for values in Σ_{deco} is expanded as the terminal type 1 in Σ_{expl}. In this way, the properties of products of values in the decorated logic get satisfied by their images in the explicit logic.

Similarly, every non-exceptional sum $\sum_{i=1}^n (j_i^v : Y_i \to Y)$ in Σ_{deco} is expanded as a sum $\sum_{i=1}^n (j_{i,v} : Y_i \to Y)$ in Σ_{expl}, so that the initial type for values 0 in Σ_{deco} is expanded as the initial type 0 in Σ_{expl}, and the value $\mathrm{raise}_Y^v = [\,]^v : 0 \to Y$ gets expanded as $[\,]_v : 0 \to Y$, for each type Y. In this

way, the properties of sums of values in the decorated logic get satisfied by their images in the explicit logic. This includes the existence and unicity, up to $\equiv$, of the inverse image of any value u^v, which gets expanded as the inverse image of the term u_v. This also includes the property that there are matches of computations ; indeed let $(t_i^c : Y_i \to Z)_{1 \leq i \leq n}$ be computations in Σ_{deco}, they get expanded as terms $(t_{i,c} : Y_i \to Z + E)_{1 \leq i \leq n}$, and the computation $[t_1 \mid \ldots \mid t_n]^c : Y \to Z$ gets expanded as the term $[t_1 \mid \ldots \mid t_n]_c = [t_{1,c} \mid \ldots \mid t_{n,c}] : Y \to Z + E$.

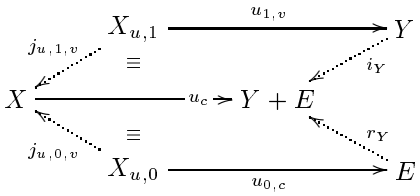
Let us now look at the cases over computations. Let $u^c : X \to Y$ be a computation in Σ_{deco}, then the inverse image of the sum $Y = Y + 0$ by u^c is a sum $X = X_{u,1} + X_{u,0}$ together with a value $u_1^v : X_{u,1} \to Y$ and a computation $u_0^c : X_{u,0} \to 0$ satisfying the equations $u \circ j_{u,1} \equiv^c u_1$ and $u \circ j_{u,0} \equiv^c \text{raise}_Y \circ u_0$. From section 4.2, the expansion of the equation $u \circ j_{u,1} \equiv^c u_1$ is the equation:

$$u_c \circ j_{u,1,v} \equiv u_{1,c} \quad \text{where} \quad u_{1,c} \equiv i_Y \circ u_{1,v} \ ,$$

and the expansion of the equation $u \circ j_{u,0} \equiv^c \text{raise}_Y \circ u_0$ is the equation:

$$u_c \circ j_{u,0,v} \equiv ([\,]_Y)_c \circ u_{0,c} \quad \text{where} \quad ([\,]_Y)_c \circ u_{0,c} \equiv r_Y \circ u_{0,c} \ .$$

This means that the expansion of this inverse image in Σ_{deco} is the inverse image of the sum $Y + E$ by the term $u_c : X \to Y + E$ in Σ_{expl}.



Now, let us look at the exceptional cases. The exceptions $e_i^c : P_i \to 0$ are computations, so that they get expanded as $e_{i,c} : P_i \to E$. The image of each exceptional sum in Σ_{deco} is a sum, with vertex E, in Σ_{expl}. The expansion of an inverse image of an exceptional sum in Σ_{deco} is the inverse image of a sum with vertex E in Σ_{expl}.

Since the raising and handling of exceptions, as well as the records of computations, have been defined in terms of these sums and products, they get expanded accordingly.

4.4 The expansion morphism

In sections 4.2 to 4.3, the contravariant functor:

$$\mathrm{W} = \mathrm{Yon}_{\text{expl}} \circ \chi : \mathcal{S}_{\text{deco}} \nrightarrow \mathbf{Real}(\mathcal{S}_{\text{expl}})$$

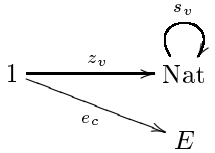
is defined, from which the generalized morphism $\chi : P_{\text{deco}} \to P_{\text{expl}}$ follows, by the properties of the Yoneda functor.

Definition 4.2 The *expansion morphism*:

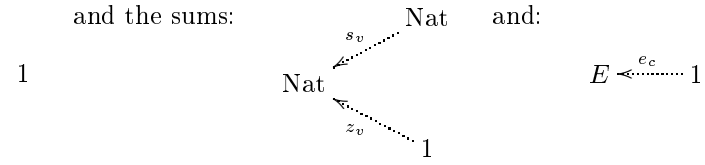
$$\chi : P_{\text{deco}} \to P_{\text{expl}}$$

is the unique generalized morphism from P_{deco} to P_{expl}, such that $\mathrm{W} = \mathrm{Yon}_{\text{expl}} \circ \chi$.

Example 4.3 Let $\Sigma_{\text{nat,expl}} = \mathrm{F}_\chi(\Sigma_{\text{nat,deco}})$: it is made of a copy of $\Sigma_{\text{nat},0}$ from example 2.16, with the index v, together with $e_c : 1 \to E$, which has to be a sum, which means that e_c has to be an isomorphism.



with the product: and the sums: and:



4.5 Explicit models

Definition 4.4 Let Σ_{expl} be an explicit specification and Δ_{expl} an explicit domain. The set of *explicit models of* Σ_{expl} *with values in* Δ_{expl} is defined as in appendix A:

$$\mathrm{Mod}_{\mathrm{expl}}(\Sigma_{\mathrm{expl}}, \Delta_{\mathrm{expl}}) = \mathrm{Hom}_{\mathbf{Dom}(P_{\mathrm{expl}})}(\mathrm{F}_{P_{\mathrm{expl}}}(\Sigma_{\mathrm{expl}}), \Delta_{\mathrm{expl}}) \simeq \mathrm{Hom}_{\mathbf{Spec}(P_{\mathrm{expl}})}(\Sigma_{\mathrm{expl}}, \mathrm{G}_{P_{\mathrm{expl}}}(\Delta_{\mathrm{expl}}))\ .$$

Each set $\mathbb{E}$ gives rise to an explicit domain $\mathbf{Set}_{\mathrm{expl}}[\mathbb{E}]$, made of the basic domain **Set** together with the distinguished set $\mathbb{E}$.

Definition 4.5 Let Σ_{expl} be an explicit specification. For each set $\mathbb{E}$, a *set-valued explicit model of* Σ_{expl} *with set of exceptions* $\mathbb{E}$ is an explicit model of Σ_{expl} with values in the explicit domain $\mathbf{Set}_{\mathrm{expl}}[\mathbb{E}]$.

This means that it is a set-valued explicit domain is a set-valued model (in the basic sense) such that the interpretation of the type E is the set $\mathbb{E}$.

Since it is a morphism of propagators, the expansion morphism gives rise to an adjunction between the explicit specifications and the decorated specifications, and to an adjunction between the explicit domains and the decorated domains. Each decorated specification Σ_{deco} freely generates an explicit specification $\Sigma_{\mathrm{expl}} = \mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$. By corollary A.20, the freely generated specification Σ_{expl} can be built from the images of W:

$$\Sigma_{\mathrm{expl}} = \mathrm{F}_\chi(\Sigma_{\mathrm{deco}}) \sim \mathrm{colim}_{(\mathcal{S}_{\mathrm{deco}} \backslash \Sigma_{\mathrm{deco}})^{op}}(\mathrm{W}(S))\ .$$

It follows from proposition A.34 that, for each decorated specification Σ_{deco} and each explicit domain Δ_{expl}, there is a natural bijection:

$$\mathrm{Mod}_{\mathrm{deco}}(\Sigma_{\mathrm{deco}}, \mathrm{G}_\delta(\Delta_{\mathrm{expl}})) \simeq \mathrm{Mod}_{\mathrm{expl}}(\Sigma_{\mathrm{expl}}, \Delta_{\mathrm{expl}})\ .$$

For each set $\mathbb{E}$, when $\Delta_{\mathrm{expl}} = \mathbf{Set}_{\mathrm{expl}}[\mathbb{E}]$, it is easy to check that $\mathrm{G}_\chi(\mathbf{Set}_{\mathrm{expl}}[\mathbb{E}]) \sim \mathbf{Set}_{\mathrm{deco}}[\mathbb{E}]$, so that the expansion morphism "does preserve the models".

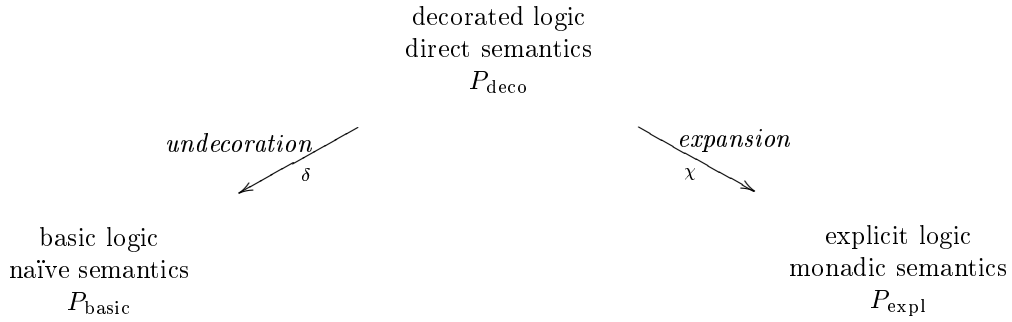
Theorem 4.6 (Denotational semantics of exceptions) *Let* Σ_{deco} *be a decorated specification, and let* $\Sigma_{\mathrm{expl}} = \mathrm{F}_\chi(\Sigma_{\mathrm{deco}})$. *For each set* $\mathbb{E}$, *there is a natural bijection:*

$$\mathrm{Mod}_{\mathrm{deco}}(\Sigma_{\mathrm{deco}}, \mathbf{Set}_{\mathrm{deco}}[\mathbb{E}]) \simeq \mathrm{Mod}_{\mathrm{expl}}(\Sigma_{\mathrm{expl}}, \mathbf{Set}_{\mathrm{expl}}[\mathbb{E}])\ .$$

The explicit models of Σ_{expl} with values in $\mathbf{Set}_{\mathrm{expl}}[\mathbb{E}]$ form the *monadic semantics* of exceptions, in the sense of [15]. Thanks to theorem 4.6, they can be identified to the decorated models of Σ_{deco} with values in $\mathbf{Set}_{\mathrm{deco}}[\mathbb{E}]$, so that these decorated models provide a *direct semantics* of exceptions.

5 Conclusion and further work

We have defined three propagators, for three diagrammatic logics, related by two morphisms:



The decorated logic provides a syntax, a deduction system and a direct denotational semantics for exceptions. The undecoration morphism gives a simplified view on the syntax and the deductions, while the expansion morphism provides the monadic semantics of exceptions.

This work could be improved by taking into account the extensibility of exceptions, as in SML.

This work could also be completed in the operational direction. It has been stated in section 3.12 that "a proof in Σ_{deco} is a proof in Σ_{basic} which can be decorated". In order to go further, it should be possible to extend this from proofs to computations, so that "a computation in Σ_{deco} is a computation in Σ_{basic} which can be decorated". This could rely on [17] and on [12, 13].

Another direction for future research is to use a similar approach, via morphisms of diagrammatic logics, in order to study other effects, like the side-effects due to the state in imperative languages. The combination of various effects should run smoothly in our diagrammatic framework. Other computational features, like the overloading of terms, can also be studied from this point of view; this looks quite promising, as suggested by [9].

References

[1] A. Asperti, G. Longo. *Categories, Types and Structures. An introduction to Category Theory for the working computer scientist.* M.I.T. Press (1991). http://www.di.ens.fr/users/longo/download.html

[2] M. Barr, C. Wells. *Category Theory for Computing Science*, Prentice Hall (1990).

[3] J. Bénabou. Les Distributeurs. *Rapport de l'Université Catholique de Louvain, Institut de Mathématique Pure et Appliquée* **33** (1973).

[4] A. Carboni, S. Lack, R.F.C. Walters. Introduction to extensive and distributive categories, *J. Pure Appl. Algebra* **84**145-158 (1993) .

[5] L. Coppey. Théories algébriques et extensions de pré-faisceaux, *Cahiers de Topologie et Géométrie Différentielle* **13** (1972).

[6] L. Coppey, C. Lair. Leçons de Théorie des esquisses (I), *Diagrammes* **12** (1984).

[7] L. Coppey, C. Lair. Leçons de Théorie des esquisses (II), *Diagrammes* **19** (1988).

[8] D. Duval. Diagrammatic specifications. *Mathematical Structures in Computer Science* **13** 857-890 (2003). This is the journal version of [10].

[9] D. Duval, C. Kirchner, C. Lair. Subtypes and Subsorts in Overloaded Specifications *Rapport de Recherche IMAG-LMC* **1058** (2003) `http://www-lmc.imag.fr/lmc-cf/Dominique.Duval/`

[10] D. Duval, C. Lair. Diagrammatic specifications. *Rapport de recherche IMAG-LMC* **1043** (2002). This is a preliminary version of [8]. `http://www-lmc.imag.fr/lmc-cf/Dominique.Duval/`

[11] D. Duval, C. Lair, C. Oriat, J.-C. Reynaud. A zooming process for specifications, with an application to exceptions *Rapport de Recherche IMAG-LMC* **1055** (2003) `http://www-lmc.imag.fr/lmc-cf/Dominique.Duval/`

[12] D. Duval, J.-C. Reynaud. Sketches and computation (Part I): Basic Definitions and Static Evaluation *Mathematical Structures in Computer Science* **4** 185-238 (1994)

[13] D. Duval, J.-C. Reynaud. Sketches and computation (Part II): Dynamic Evaluation and Applications *Mathematical Structures in Computer Science* **4** 239-271 (1994)

[14] C. Ehresmann. Introduction to the theory of structured categories. Report **10**, University of Kansas, Lawrence (1966).

[15] C. Führmann. The structure of call-by-value. PhD thesis, Division of Informatics, University of Edinburgh (2000).

[16] M. Hébert, J. Adámek, J. Rosický. More on orthogonality in locally presentable categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* **XLII-1**, 51–80 (2001).

[17] C.B. Jay. Tail recursion through universal invariants. *Theoretical Computer Science* **115** 151–189 (1993).

[18] C. Lair. Trames et sémantiques catégoriques des systèmes de trames, *Diagrammes* **18**, Paris, CL1–CL47 (1987).

[19] C. Lair, D. Duval. Esquisses et spécifications. Manuel de référence, 4ème partie : Fibrations et Eclatements, Lemmes de Yoneda et Modèles Engendrés. *Rapport de recherche du LACO, Université de Limoges* **2001-03**. http://www.unilim.fr/laco/rapports/.

[20] S. Mac Lane. *Categories for the working mathematician*, Springer-Verlag (1971).

[21] R. Milner, M. Tofte, R. Harper. *The definition of Standard ML*, MIT Press (1990).

[22] E. Moggi. Notions of computation and monads, *Information and Computation* **93**, 55–92 (1991).

[23] G. Plotkin, J. Power. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science* **45**, 1–14 (2001).

[24] G. Plotkin and J. Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures* **11**, 69–94 (2003). This is the journal version of [23].

[25] J. Power, H. Thielecke. Closed Freyd- and κ-Categories *Proc. ICALP'99, LNCS* **1644** (1999).

[26] M. Tofte. *Tips for Computer Scientists On Standard ML.* Department of Computer Science, University of Copenhagen (1993). http://www.diku.dk/users/tofte/publ/publ.html

A Diagrammatic logic

A diagrammatic logic is made of a syntax, a deduction system and a sound notion of models. The syntax tells which features can be defined (typically, axioms and theorems). The deduction system is such that some features (typically, theorems) can be generated from some elementary ones (typically, axioms), and any generated feature can be added to the elementary ones, changing neither the generated features nor the models.

The papers [10, 8] introduce a framework for dealing with *diagrammatic logic*, which is based on *projective sketches*, more precisely on *propagators*, which are quite simple:

> a propagator is a morphism of projective sketches such that the associated omitting functor is full and faithful[1].

Any morphism of projective sketches gives rise to an adjunction, so that "features can be generated from elementary ones". The additionnal property of propagators ensures that "any generated feature can be added to the elementary ones". For instance, a propagator for a logic with records and case distinctions is built in section 2.

In this appendix, the main definitions and results from [10, 8] are presented, together with a simplified definition of a deduction step and a new result about the naturality of deduction steps (theorem A.31). Some knowledge of category theory is assumed, which can be found in many textbooks, like [20] or [1]. As usual, the definitions and results in this appendix are valid only under relevant size assumptions.

A.1 Projective sketches

As explained in [10, 8], *projective sketches* are used at the meta-level in order to define a diagrammatic logic. Sketches appear in [14], and an introduction to sketch theory can be found in [6, 7] or in [2]. Some basic definitions are given below, together with the theorems which are used for defining diagrammatic logic.

Definition A.1 A *(directed multi-)graph* is made of *points* and *arrows*, with a *source* and a *target* for each arrow. A *morphism of graphs* is made of two maps, one for points and one for arrows, which preserve the sources and targets. Graphs with their morphisms form a category **Gr**.

Definition A.2 A *compositive graph* is a graph where some points X have an *identity arrow* $\mathrm{id}_X : X \to X$ and some consecutive pairs of arrows $(f : X \to Y, g : Y \to Z)$ have a *composed arrow* $g \circ f : X \to Z$ (there is no assumption about associativity and unitary of composition and identities). A *morphism of compositive graphs* is a morphism of graphs which preserves the identities and the composed arrows. Compositive graphs with their morphisms form a category **Comp**.

In a compositive graph $\mathcal{G}$, a *cone* is made of a *vertex* point G, a *base* morphism of compositive graphs $b : \mathcal{I} \to \mathcal{G}$, and *projection* arrows $p_I : G \to b(I)$ for each point I in $\mathcal{I}$, such that $b(i) \circ p_I = p_{I'}$ for each arrow $i : I \to I'$ in $\mathcal{I}$.

Definition A.3 A *projective sketch* is a compositive graph where some cones are called *distinguished cones*. A *morphism of projective sketches* is a morphism of compositive graphs which preserves the distinguished cones. Projective sketches with their morphisms form a category **PSk**.

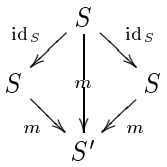
The vertex of a distinguished cone with an empty base is denoted $\mathbb{1}$.

[1] This is slightly different from [8], where any morphism of projective sketches is called a "propagator".

Definition A.4 A *(set-valued) realization* Σ of a projective sketch $\mathcal{S}$, also called a $\mathcal{S}$*-realization*, maps each point S of $\mathcal{S}$ to a set $\Sigma(S)$ and each arrow $s : S_1 \to S_2$ of $\mathcal{S}$ to a map $\Sigma(s) : \Sigma(S_1) \to \Sigma(S_2)$, in such a way that each identity loop becomes an identity map, each composed arrow becomes a composed map, and each distinguished cone of $\mathcal{S}$ becomes a limit cone in the category of sets. A *morphism of realizations* of $\mathcal{S}$ is a natural transformation $\sigma : \Sigma \to \Sigma'$; this means that it is made of a map $\sigma_S : \Sigma(S) \to \Sigma'(S)$ for each point S of $\mathcal{S}$, such that $\Sigma'(s) \circ \sigma_{S_1} = \sigma_{S_2} \circ \Sigma(s)$ for each arrow $s : S_1 \to S_2$ of $\mathcal{S}$. The realizations of $\mathcal{S}$ with their morphisms form a category $\mathbf{Real}(\mathcal{S})$.

A realization is often called a *model* of $\mathcal{S}$. But here we speak about the *realizations* of a projective sketch $\mathcal{S}$ at the meta-level, then we will speak about the *models* of a diagrammatic specification Σ at the level of specifications.

Definition A.5 An arrow $m : S \to S'$ in a projective sketch $\mathcal{S}$ is called a *mono*, and is represented as $S \overset{m}{\rightarrowtail} S'$ when there is in $\mathcal{S}$ a distinguished cone of the following form:



It follows that a mono becomes an injective map in each realization of $\mathcal{S}$.

A.2 Adjunction

Let us consider a morphism of projective sketches:

$$M : \mathcal{S} \to \mathcal{S}' \ .$$

The *omitting functor* $\mathrm{G}_M : \mathbf{Real}(\mathcal{S}') \to \mathbf{Real}(\mathcal{S})$ is such that $\mathrm{G}_M(\Sigma') = \Sigma' \circ M$ for all realization Σ' of $\mathcal{S}'$. The following major theorem about projective sketches is due to Ehresmann [14].

Theorem A.6 (adjunction theorem) *The functor* $\mathrm{G}_M : \mathbf{Real}(\mathcal{S}') \to \mathbf{Real}(\mathcal{S})$ *has a left adjoint* $\mathrm{F}_M : \mathbf{Real}(\mathcal{S}) \to \mathbf{Real}(\mathcal{S}')$.

$$\mathbf{Real}(\mathcal{S}) \underset{\mathrm{G}_M}{\overset{\mathrm{F}_M}{\rightleftarrows}} \mathbf{Real}(\mathcal{S}')$$

The adjunction determines two natural transformations: the *unit* $\eta : \mathrm{id}_{\mathbf{Real}(\mathcal{S})} \Rightarrow \mathrm{G}_M \circ \mathrm{F}_M$ and the *counit* $\varepsilon : \mathrm{F}_M \circ \mathrm{G}_M \Rightarrow \mathrm{id}_{\mathbf{Real}(\mathcal{S}')}$.

Definition A.7 The *freely generating functor* with respect to M is the left adjoint functor F_M of G_M.

A morphism of projective sketches $Q : \mathcal{S} \to \mathcal{S}'$ is called an *equivalence* if both F_Q of G_Q are full and faithful. The *equivalence* of projective sketches, as well as the *equivalence* of morphisms of projective sketches (both denoted $\sim$), are the equivalence relations generated by the fact that, if there is an equivalence morphism $Q : \mathcal{S} \to \mathcal{S}'$, then:
– $\mathcal{S} \sim \mathcal{S}'$,
– for every $M_1 : \mathcal{S}_1 \to \mathcal{S}$, $Q \circ M_1 \sim M_1$,
– and for every $M'_1 : \mathcal{S}' \to \mathcal{S}'_1$, $M'_1 \circ Q \sim M'_1$.

It is easy to check that a morphism of projective sketches $Q : \mathcal{S} \to \mathcal{S}'$ that adds identity arrows to points in $\mathcal{S}$, composed arrows for consecutive pairs of arrows in $\mathcal{S}$, and distinguished cones with their base in $\mathcal{S}$, is an equivalence.

The notion of morphism of projective sketches can be generalized as follows: a *generalized morphism of projective sketches* from $\mathcal{S}_0$ to $\mathcal{S}$ is a morphism $M : \mathcal{S}_0 \to \mathcal{S}'$ together with an equivalence morphism $Q : \mathcal{S} \to \mathcal{S}'$.

Equivalent sketches have equivalent categories of realizations (the equivalence of categories is also denoted $\sim$): if $\mathcal{S} \sim \mathcal{S}'$, then $\mathbf{Real}(\mathcal{S}) \sim \mathbf{Real}(\mathcal{S}')$.

Definition A.8 A *projective prototype* is a projective sketch such that its underlying compositive graph is a category and all its distinguished cones are limit cones. A *morphism of projective prototypes* is a morphism of projective sketches.

Theorem A.9 (prototype theorem) *Each projective sketch $\mathcal{S}$ freely generates a projective prototype* $\mathrm{Proto}(\mathcal{S})$.

The prototype theorem can be derived from the adjunction theorem A.6, since the categories of projective sketches and of projective prototypes can themselves be identified to the categories of realizations of two projective sketches.

A.3 Propagators and distributors

Definition A.10 A *propagator* is a morphism of projective sketches $P : \mathcal{S} \to \mathcal{S}'$ such that the functor G_P is full and faithful [2].

Equivalently, P is such that the counit $\varepsilon : \mathrm{F}_P \circ \mathrm{G}_P \Rightarrow \mathbf{Real}(\mathcal{S}')$ is an isomorphism.

The following theorem is proven in [16].

Theorem A.11 (inversion theorem) *A morphism of projective sketches is a propagator if and only if, up to equivalence, it consists of adding inverses to arrows.*

Definition A.12 A *morphism of propagators* $\alpha : P_1 \to P_2$, where $P_1 : \mathcal{S}_1 \to \mathcal{S}'_1$ and $P_2 : \mathcal{S}_2 \to \mathcal{S}'_2$ is a pair of morphism of projective sketches, both denoted α:

$$\alpha : \mathcal{S}_1 \to \mathcal{S}_2 \quad \text{and } \alpha : \mathcal{S}'_1 \to \mathcal{S}'_2$$

such that $\alpha \circ P_1 = P_2 \circ \alpha$:

$$\begin{array}{ccc} \mathcal{S}_1 & \xrightarrow{P_1} & \mathcal{S}'_1 \\ \downarrow{\scriptstyle \alpha} & = & \downarrow{\scriptstyle \alpha} \\ \mathcal{S}_2 & \xrightarrow{P_2} & \mathcal{S}'_2 \end{array}$$

Propagators with their morphisms form a category **Prop**.

Theorem A.11 has the following corollary.

Corollary A.13 (morphisms of propagators) *Let $P_1 : \mathcal{S}_1 \to \mathcal{S}'_1$ and $P_2 : \mathcal{S}_2 \to \mathcal{S}'_2$ be two propagators. A morphism of projective sketches $\alpha : \mathcal{S}_1 \to \mathcal{S}_2$ determines a morphism of propagators $\alpha : P_1 \to P_2$ if and only if, for each arrow s in $\mathcal{S}_1$ such that $P_1(s)$ is invertible in $\mathcal{S}'_1$, the arrow $P_2(\alpha(s))$ in $\mathcal{S}_2$ is invertible in $\mathcal{S}'_2$*

[2] In [8], this is called a "fractionning propagator".

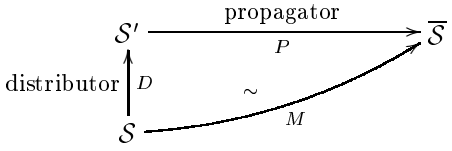
Definition A.14 A *distributor* is a morphism of projective sketches $D : \mathcal{S} \to \mathcal{S}'$ such that the functor F_D is full and faithful. [3].

Equivalently, D is such that the unit $\eta : \mathrm{id}_{\mathbf{Real}(\mathcal{S})} \Rightarrow \mathrm{G}_D \circ \mathrm{F}_D$ is an isomorphism.

The following theorem is the main result of [10, 8]; it states that, up to equivalence, any morphism of projective sketches can be decomposed as a distributor followed by a propagator.

Theorem A.15 (decomposition theorem) *Let $M : \mathcal{S} \to \overline{\mathcal{S}}$ be a morphism of projective sketches. There are a projective sketch $\mathcal{S}'$, a distributor $D : \mathcal{S} \to \mathcal{S}'$ and a propagator $P : \mathcal{S}' \to \overline{\mathcal{S}}$, such that:*

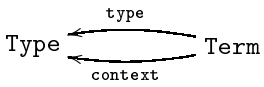
$$M \sim P \circ D \ .$$



This decomposition is not unique. The proof which is given in [8] is an effective one: it builds explicitely D and P in such a way that, in addition, the functor F_D is very easy to compute.

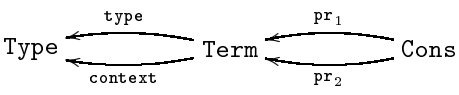
Example A.16 In the examples, "sketch" means "projective sketch". Graphs are used at the meta-level, for building sketches; at the meta-level, we speak about *points* and *arrows*, and about the *source* and the *target* of an arrow. Graphs are also used at the level of specifications, for building various kinds of specifications; at this level, we rather speak about *types* and *terms*, and about the *context* and the *type* of a term. Usually, types are built from elementary types, called *sorts*, and terms are built from elementary terms, called *operations*, or *operation symbols*. In this example, there is no type constructor, and the unique constructor for terms is the composition. In section 2, other constructors will be introduced.

Let $\mathcal{S}_{\mathrm{gr}}$ be the sketch made of two points `Type` and `Term` for *types* and *terms*, two arrows `type` and `context` for *context* and *type*, and no distinguished cone:

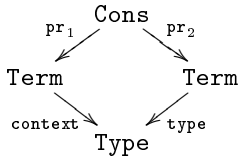


The sketch $\mathcal{S}_{\mathrm{gr}}$ is a *sketch of graphs*, in the sense that its category of realizations is isomorphic to the category of graphs: $\mathbf{Real}(\mathcal{S}_{\mathrm{gr}}) \sim \mathbf{Gr}$.

The sketch $\mathcal{S}_{\mathrm{gr}}$ can be enriched in order to deal with consecutive terms. The resulting sketch $\mathcal{S}'_{\mathrm{gr}}$ has a point `Cons` for *consecutive pairs (of terms)*, two arrows $\mathtt{pr}_1, \mathtt{pr}_2 : \mathtt{Cons} \to \mathtt{Term}$ for the projections, and a distinguished cone $\Gamma_{\mathtt{Cons}}$, which formalizes the definition of consecutive pairs:



$\Gamma_{\mathtt{Cons}}$:



Since the basis of $\Gamma_{\mathtt{Cons}}$ is in $\mathcal{S}_{\mathrm{gr}}$, the inclusion of $\mathcal{S}_{\mathrm{gr}}$ in $\mathcal{S}'_{\mathrm{gr}}$ is an equivalence. Hence, $\mathcal{S}'_{\mathrm{gr}}$ is another *sketch of graphs*: $\mathbf{Real}(\mathcal{S}'_{\mathrm{gr}}) \sim \mathbf{Gr}$.

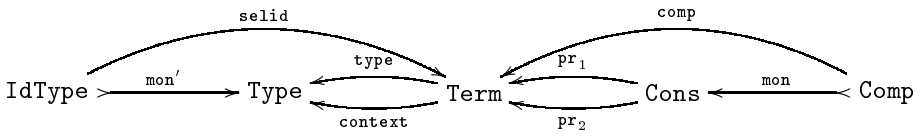
[3] In [8], this is called a "filling propagator", and a "distributor" is a special kind of "filling propagator"; the name "distributor" is inspired by [3].

In order to get a *sketch of compositive graphs* $\mathcal{S}_{\text{comp}}$, let us enrich $\mathcal{S}'_{\text{gr}}$ with a point $\mathtt{Comp}$ for *composable pairs (of terms)*, a mono $\mathtt{mon} : \mathtt{Comp} \rightarrowtail \mathtt{Cons}$ for the inclusion, and an arrow $\mathtt{comp} : \mathtt{Comp} \to \mathtt{Term}$ for the *composition* of composable terms, such that:

$$\mathtt{type} \circ \mathtt{comp} = \mathtt{type} \circ \mathtt{pr}_1 \circ \mathtt{mon} \quad \text{and} \quad \mathtt{context} \circ \mathtt{comp} = \mathtt{context} \circ \mathtt{pr}_2 \circ \mathtt{mon} \,,$$

and also with a point $\mathtt{IdType}$ for *types with identity*, a mono $\mathtt{mon}' : \mathtt{IdType} \rightarrowtail \mathtt{Type}$ for the inclusion, and an arrow $\mathtt{selid} : \mathtt{IdType} \to \mathtt{Term}$ for the *selection of identity*, such that:

$$\mathtt{type} \circ \mathtt{selid} = \mathtt{mon}' \quad \text{and} \quad \mathtt{context} \circ \mathtt{selid} = \mathtt{mon}' \,.$$



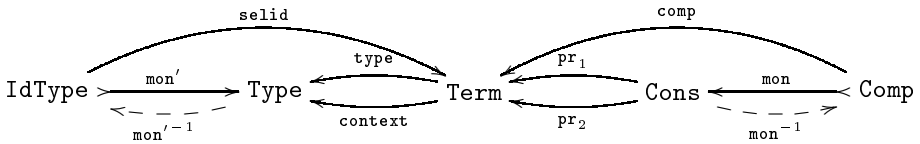
Then, $\mathcal{S}_{\text{comp}}$ is a *sketch of compositive graphs*: $\mathbf{Real}(\mathcal{S}_{\text{comp}}) \sim \mathbf{Comp}$. Moreover, it is easy to check that the inclusion of $\mathcal{S}_{\text{gr}}$ in $\mathcal{S}_{\text{comp}}$ is a distributor.

Let us say that a compositive graph is *saturated* if each consecutive pair of terms can be composed and each type has an identity. The saturated compositive graphs form a full subcategory **SComp** of **Comp**. A sketch $\overline{\mathcal{S}}_{\text{comp}}$ of saturated compositive graphs is built by enriching $\mathcal{S}_{\text{comp}}$ with an inverse for the arrows $\mathtt{mon} : \mathtt{Comp} \to \mathtt{Cons}$ and $\mathtt{mon}' : \mathtt{IdType} \to \mathtt{Type}$. This means that two arrows $\mathtt{mon}^{-1} : \mathtt{Cons} \to \mathtt{Comp}$ and $\mathtt{mon}'^{-1} : \mathtt{Type} \to \mathtt{IdType}$ are added, such that:

$$\mathtt{mon}^{-1} \circ \mathtt{mon} = \mathrm{id}_{\mathtt{Cons}} \quad \text{and} \quad \mathtt{mon} \circ \mathtt{mon}^{-1} = \mathrm{id}_{\mathtt{Comp}} \,,$$

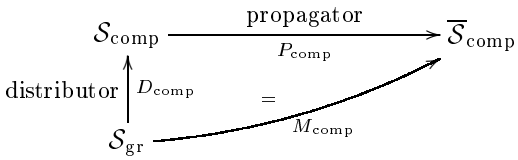
$$\mathtt{mon}'^{-1} \circ \mathtt{mon}' = \mathrm{id}_{\mathtt{IdType}} \quad \text{and} \quad \mathtt{mon}' \circ \mathtt{mon}'^{-1} = \mathrm{id}_{\mathtt{Type}} \,.$$

Then the arrow $\mathtt{comp} \circ \mathtt{mon}^{-1} : \mathtt{Cons} \to \mathtt{Term}$ stands for the *composition* of consecutive terms, and the arrow $\mathtt{selid} \circ \mathtt{mon}'^{-1} : \mathtt{Type} \to \mathtt{Term}$ for the *selection of identity* of every type:



The inclusion of $\mathcal{S}_{\text{comp}}$ in $\overline{\mathcal{S}}_{\text{comp}}$ is a propagator, which is the starting point for all the propagators in this paper.

This construction gives rise to an instance of the decomposition theorem: the sketch of compositive graphs $\mathcal{S}_{\text{comp}}$ can be used in order to decompose the morphism $M_{\text{comp}} : \mathcal{S}_{\text{gr}} \to \overline{\mathcal{S}}_{\text{comp}}$ as a distributor followed by a propagator:



A sketch $\mathcal{S}_{\text{cat}}$ of categories can easily be obtained as an enrichment of $\overline{\mathcal{S}}_{\text{comp}}$ with additional features, in order to deal with the associativity and unitarity axioms. The inclusion of $\overline{\mathcal{S}}_{\text{comp}}$ in $\mathcal{S}_{\text{cat}}$ is, up to equivalence, a propagator.

A.4 The Yoneda morphism

Let $\mathcal{S}$ be a projective sketch. The Yoneda morphism associated to $\mathcal{S}$ relates the projective sketch $\mathcal{S}$ to the category $\mathbf{Real}(\mathcal{S})$, in a contravariant way [19]. Basically, when $\mathcal{S}$ is a projective prototype (as in definition A.8) its Yoneda morphism maps each point S of $\mathcal{S}$ to the set of arrows with source S in $\mathcal{S}$. More precisely, the definition and some of the properties of the Yoneda morphism are summarized below. Then, some of its applications are reviewed. As above, the symbol "$\nrightarrow$" is used to denote contravariance.

Definition A.17 The *Yoneda morphism* of $\mathcal{S}$ is a contravariant morphism:

$$\mathrm{Yon}_{\mathcal{S}} : \mathcal{S} \nrightarrow \mathbf{Real}(\mathcal{S}) \ .$$

Whenever $\mathcal{S}$ is a projective prototype (hence, a category), $\mathrm{Yon}_{\mathcal{S}}$ is the contravariant functor such that, for all point S of $\mathcal{S}$:

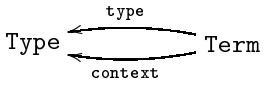
$$\mathrm{Yon}_{\mathcal{S}}(S) = \mathrm{Hom}_{\mathcal{S}}(S, -)$$

and for all arrow $s : S \to S'$ of $\mathcal{S}$:

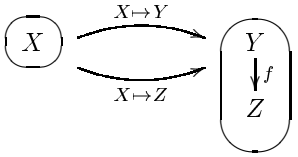
$$\mathrm{Yon}_{\mathcal{S}}(s) = \mathrm{Hom}_{\mathcal{S}}(s, -) : \mathrm{Yon}_{\mathcal{S}}(S') \to \mathrm{Yon}_{\mathcal{S}}(S) \ .$$

When $\mathcal{S}$ is any projective sketch, it freely generates a projective prototype $\mathrm{Proto}(\mathcal{S})$, and the Yoneda morphism $\mathrm{Yon}_{\mathcal{S}}$ is composed of the canonical morphism from $\mathcal{S}$ to $\mathrm{Proto}(\mathcal{S})$, followed by $\mathrm{Yon}_{\mathrm{Proto}(\mathcal{S})}$.

Example A.18 The sketch of graphs $\mathcal{S}_{\mathrm{gr}}$, from example A.16:



is mapped, via the Yoneda morphism, to the following diagram in the category of graphs:



The following theorem states some of the properties of the Yoneda morphism, it deserves some preliminary remarks. The notion of a set-valued realization of a projective sketch, as given in definition A.4, is generalized here in the obvious way. The density property, in this theorem, deals with a colimit indexed by the compositive graph $\mathcal{S}\backslash\Sigma$, where Σ is a realization of a projective sketch $\mathcal{S}$. The compositive graph $\mathcal{S}\backslash\Sigma$ is made of a point S_x for all point S of $\mathcal{S}$ and all $x \in \Sigma(S)$, an arrow $s_x : S_x \to S'_{\Sigma(s)(x)}$ for all arrow $s : S \to S'$ of $\mathcal{S}$ and all $x \in \Sigma(S)$, together with the identities $\mathrm{id}_{S_x} = (\mathrm{id}_S)_x$ when id_S exists in $\mathcal{S}$, and the composites $(s' \circ s)_x = s'_{\Sigma(s)(x)} \circ s_x$, when $s' \circ s$ exists in $\mathcal{S}$. In addition, as usual, $(\mathcal{S}\backslash\Sigma)^{op}$ is the compositive graph *opposite* of $(\mathcal{S}\backslash\Sigma)$, with all the arrows in the opposite direction.

Theorem A.19 (Yoneda theorem) *The Yoneda morphism* $\mathrm{Yon}_{\mathcal{S}}$ *is a contravariant realization of* $\mathcal{S}$*, which means that it maps each distinguished cone in* $\mathcal{S}$ *to a colimit cone in* $\mathbf{Real}(\mathcal{S})$*. In addition:*

- ***Compatibity:*** *Let* $M : \mathcal{S} \to \mathcal{S}'$ *be a morphism of projective sketches, then:* $\mathrm{F}_M \circ \mathrm{Yon}_{\mathcal{S}} \sim \mathrm{Yon}_{\mathcal{S}'} \circ M$.
- ***Yoneda property:*** *For all realization* Σ *of* $\mathcal{S}$*:* $\Sigma \sim \mathrm{Hom}_{\mathbf{Real}(\mathcal{S})}(\mathrm{Yon}_{\mathcal{S}}(-), \Sigma)$.
- ***Density property:*** *For all realization* Σ *of* $\mathcal{S}$*:* $\Sigma \sim \mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{Yon}_{\mathcal{S}}(S))$.

The Yoneda property means that for all point S of $\mathcal{S}$ there are "$\Sigma(S)$" ways to map $\mathrm{Yon}_{\mathcal{S}}(S)$ to Σ. The density property means that, up to isomorphism, Σ can be recovered by gluing together "in the right way" one copy of $\mathrm{Yon}_{\mathcal{S}}(S)$ for each element of $\Sigma(S)$. This gives rise, in the next corollary, to a construction of the image of the freely generating functor F_M associated to M.

Corollary A.20 (construction of freely generated realizations) *Let $M : \mathcal{S} \to \mathcal{S}'$ be a morphism of projective sketches, and Σ a realization of $\mathcal{S}$. Let:*

$$\mathrm{W}_M = \mathrm{Yon}_{\mathcal{S}'} \circ M : \mathcal{S} \nrightarrow \mathbf{Real}(\mathcal{S}') \ .$$

Then:

$$\mathrm{F}_M(\Sigma) \sim \mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{W}_M(S)) \ .$$

Proof. From the compatibity property, $\mathrm{W}_M = \mathrm{Yon}_{\mathcal{S}'} \circ M \sim \mathrm{F}_M \circ \mathrm{Yon}_{\mathcal{S}}$, so that:

$$\mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{W}_M(S)) \sim \mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{F}_M \circ \mathrm{Yon}_{\mathcal{S}}) \ .$$

In addition, it is well-known that the left adjoint F_M commutes to colimits, so that:

$$\mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{F}_M \circ \mathrm{Yon}_{\mathcal{S}}) \sim \mathrm{F}_M(\mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{Yon}_{\mathcal{S}}(S))) \ .$$

Moreover, from the density property:

$$\mathrm{F}_M(\mathrm{colim}_{(\mathcal{S}\backslash\Sigma)^{op}}(\mathrm{Yon}_{\mathcal{S}}(S))) \sim \mathrm{F}_M(\Sigma) \ ,$$

hence the result follows. □

A consequence of corollary A.20 is that, in order to build $\mathrm{F}_M(\Sigma)$ for some realization Σ of $\mathcal{S}$, instead of describing $M : \mathcal{S} \to \mathcal{S}'$, it can be more convenient to describe $\mathrm{W}_M = \mathrm{Yon}_{\mathcal{S}'} \circ M : \mathcal{S} \nrightarrow \mathbf{Real}(\mathcal{S}')$.

A.5 Propagators and logics

Diagrammatic logic associates, to each propagator, simple and coherent notions of specification, models and deduction rules: that is, a syntax, a denotational semantics and an axiomatic semantics. For instance, our basic logic is described as a diagrammatic logic in section 2. However, in general, a diagrammatic logic can be quite "exotic", for instance it may happen that there is no notion of terms or formulas in such a logic. The notion of diagrammatic logic stems from Lair's trames [18].

From now on, let us consider a propagator:

$$P : \mathcal{S} \to \overline{\mathcal{S}} \ .$$

A *realization* of a projective sketch now means a set-valued realization.

Definition A.21 The *category of P-specifications* is the category of realizations of $\mathcal{S}$, and the *category of P-domains* is the category of realizations of $\overline{\mathcal{S}}$:

$$\mathbf{Spec}(P) = \mathbf{Real}(\mathcal{S}) \ \text{ and } \ \mathbf{Dom}(P) = \mathbf{Real}(\overline{\mathcal{S}}) \ .$$

Definition A.22 The *P-theory* of a P-specification Σ is the freely generated P-domain $\mathrm{F}_P(\Sigma)$.

Denotational semantics.

Definition A.23 Let Σ be a P-specification and Δ a P-domain. The *set of P-models of Σ with values in Δ* is the set of morphisms of P-domains from $\mathrm{F}_P(\Sigma)$ to Δ:

$$\mathrm{Mod}_P(\Sigma, \Delta) = \mathrm{Hom}_{\mathbf{Dom}(P)}(\mathrm{F}_P(\Sigma), \Delta) \ .$$

Hence, the adjunction property yields:

Proposition A.24 (another definition of models) *There is a natural bijection between the set of P-models of Σ with values in Δ and the set of morphisms of P-specifications from Σ to $\mathrm{G}_P(\Delta)$:*

$$\mathrm{Mod}_P(\Sigma, \Delta) \simeq \mathrm{Hom}_{\mathbf{Spec}(P)}(\Sigma, \mathrm{G}_P(\Delta)) \ .$$

Let Δ denote a P-domain. For each morphism $\sigma : \Sigma \to \Sigma'$, the map $\mathrm{Mod}_P(\sigma, \Delta) : \mathrm{Mod}_P(\Sigma', \Delta) \to \mathrm{Mod}_P(\Sigma, \Delta)$ is defined as $M' \mapsto M' \circ \mathrm{F}_P(\sigma)$. So, $\mathrm{Mod}_P(-, \Delta)$ is a contravariant functor from the category of P-specifications to the category **Set** of sets:

$$\mathrm{Mod}_P(-, \Delta) : \mathbf{Spec}(P) \nrightarrow \mathbf{Set} \ .$$

Definition A.25 A *P-consequence with respect to Δ* is a morphism of P-specifications σ such that $\mathrm{Mod}_P(\sigma, \Delta)$ is a bijection.

In addition, it can happen, and it does happen in many usual cases, that there is a natural notion of morphisms of P-models, such that the P-models form a category. Then, the functor $\mathrm{Mod}_P(-, \Delta)$ is a contravariant functor from the category of P-specifications to the category **Cat** of categories:

$$\mathrm{Mod}_P(-, \Delta) : \mathbf{Spec}(P) \nrightarrow \mathbf{Cat} \ .$$

Axiomatic semantics.

Definition A.26 An *P-entailment* is a morphism of P-specifications σ such that $\mathrm{F}_P(\sigma)$ is an isomorphism of P-domains. Two P-specifications are *equivalent* when there is a zig-zag of entailments between them.

Elementary P-entailments are the *P-deduction steps*, as described now from *active P-deduction rules*. Roughly speaking, a P-deduction rule is a property of P-domains; it is called passive if it is, more generally, a property of all P-specifications, otherwise it is called active. The active P-deduction rules are used for generating a P-domain from a P-specification.

Definition A.27 A *P-deduction rule* is an arrow in $\overline{\mathcal{S}}$. It is *passive* if it is the image, via P, of an arrow in $\mathcal{S}$, otherwise it is *active*.

It follows from this definition that the P-deduction rules can easily be composed, as arrows in the prototype of $\overline{\mathcal{S}}$.

From the inversion theorem A.11, up to equivalence, an active deduction rule is the inverse of an arrow in $\mathcal{S}$.

The Yoneda morphism can be used for visualizing the active deduction rules. Let $s : C \to H$ be an arrow in $\mathcal{S}$ which gets invertible in $\overline{\mathcal{S}}$:

The points H and C stand respectively for "hypotheses" and "conclusion", while the arrow s^{-1} can be seen as a property which has to be satisfied by any P-domain Δ. This rule is illustrated:

$$\text{either as} \quad (r = s^{-1})\ \frac{H}{C} \qquad \text{or as} \qquad H \overset{r=s^{-1}}{\underset{s}{\leftarrow}} C \ .$$

Moreover, the arrow s of $\mathcal{S}$ gives rise, in a contravariant way, to a morphism $\mathrm{Yon}_{\mathcal{S}}(s)$ in $\mathbf{Real}(\mathcal{S})$. According to the compatibility of the Yoneda morphism, the functor F_P maps $\mathrm{Yon}_{\mathcal{S}}(s)$ to $\mathrm{Yon}_{\overline{\mathcal{S}}}(s)$ in $\mathbf{Real}(\overline{\mathcal{S}})$. Since s gets invertible in $\overline{\mathcal{S}}$, the morphism $\mathrm{Yon}_{\overline{\mathcal{S}}}(s)$ has an inverse $\mathrm{Yon}_{\overline{\mathcal{S}}}(s)^{-1} = \mathrm{Yon}_{\overline{\mathcal{S}}}(s^{-1})$.

$$\text{in } \mathbf{Real}(\mathcal{S}): \quad \mathrm{Yon}_{\mathcal{S}}(H) \xrightarrow{\mathrm{Yon}_{\mathcal{S}}(s)} \mathrm{Yon}_{\mathcal{S}}(C) \qquad\qquad \text{in } \mathbf{Real}(\overline{\mathcal{S}}): \quad \mathrm{Yon}_{\overline{\mathcal{S}}}(H) \underset{\mathrm{Yon}_{\overline{\mathcal{S}}}(r)=\mathrm{Yon}_{\overline{\mathcal{S}}}(s)^{-1}}{\overset{\mathrm{Yon}_{\overline{\mathcal{S}}}(s)}{\rightleftarrows}} \mathrm{Yon}_{\overline{\mathcal{S}}}(C)$$

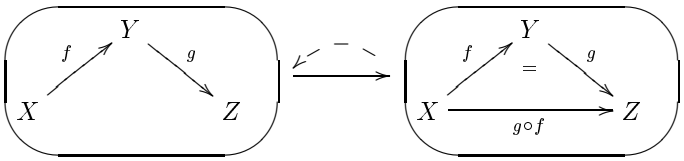
In the spirit of *D-algebras* [5], this is represented by a dashed arrow $\mathrm{Yon}_{\mathcal{S}}(s)^{-1}$ which is added to $\mathbf{Real}(\mathcal{S})$, although it does *not* correspond to any actual morphism in $\mathbf{Real}(\mathcal{S})$:

$$\mathrm{Yon}_{\mathcal{S}}(H) \underset{\mathrm{Yon}_{\mathcal{S}}(s)}{\overset{\mathrm{Yon}_{\mathcal{S}}(s)^{-1}}{\rightleftarrows}} \mathrm{Yon}_{\mathcal{S}}(C)$$

Example A.28 With respect to the propagator P_{comp}, from example A.16, the rule $\mathtt{mon}^{-1} : \mathtt{Cons} \to \mathtt{Comp}$, which states that each pair of consecutive terms can be composed, can be illustrated as:

$$\mathrm{Yon}(\mathtt{Cons}) \rightleftarrows \mathrm{Yon}(\mathtt{Comp})$$

which is:



From the Yoneda property, $x \in \Sigma(H)$ means $x : \mathrm{Yon}(H) \to \Sigma$. This results in a simple definition of a P-deduction step (simpler than the one in [8]):

Definition A.29 Let Σ be a P-specification. The *P-deduction step* associated to an active deduction rule $r = s^{-1} : H \to C$, applied to an element $x \in \Sigma(H)$, is the following morphism $\tau_s(x)$ in the pushout of $\mathrm{Yon}(s)$ and x:

$$\begin{array}{ccc} \mathrm{Yon}(H) & \xrightarrow{\mathrm{Yon}(s)} & \mathrm{Yon}(C) \\ \downarrow{\scriptstyle x} & & \downarrow{\scriptstyle c_s(x)} \\ \Sigma & \xrightarrow{\tau_s(x)} & \Sigma_s(x) \end{array}$$

Proposition A.30 *The P-deduction step $\tau_s(x)$ is a P-entailment.*

Proof. This is easily derived from the fact that $\mathrm{Yon}(s)$ is a P-entailment. □

The P-deduction step $\tau_s(x)$, for a fixed active deduction rule $s^{-1} : H \to C$, is polymorphic: it can be applied to each P-specification Σ and each element $x \in \Sigma(H)$. Let us check that there is some compatibility between the various applications of this deduction rule, which can be expressed as a naturality property.

The elements of $\Sigma(H)$ are identified, thanks to the Yoneda property, with the morphisms from $\mathrm{Yon}(H)$ to Σ. Hence, the elements $x \in \Sigma(H)$, for all the P-specifications Σ, are the arrows of $\mathbf{Spec}(P)$ with source $\mathrm{Yon}(H)$. They are the objects of the *category of objects under* $\mathrm{Yon}(H)$, denoted $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$, as in [20]. The deduction step applied to $x : \mathrm{Yon}(H) \to \Sigma$ yields a morphism:

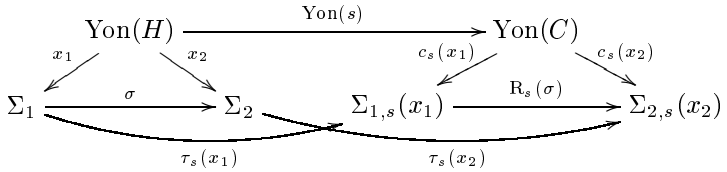
$$\mathrm{R}_s(x) = c_s(x) \circ \mathrm{Yon}(s) = \tau_s(x) \circ x : \mathrm{Yon}(H) \to \Sigma_s(x) \ .$$

So, R_s is a map from the set of objects of $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$ to itself. Moreover, since $\tau_s(x) \circ x = \mathrm{R}_s(x)$, the morphism $\tau_s(x)$ is an arrow from x to $\mathrm{R}_s(x)$ in the category $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$.

Theorem A.31 (naturality of deduction) *The map* R_s *gives rise to an endofunctor of* $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$*, and* τ_s *is a natural transformation from the identity endofunctor to* R_s*.*

Proof. Let $x_1 : \mathrm{Yon}(H) \to \Sigma_1$, $x_2 : \mathrm{Yon}(H) \to \Sigma_2$, and let $\sigma : x_1 \to x_2$ be an arrow in the category $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$, that is, $\sigma : \Sigma_1 \to \Sigma_2$ in $\mathbf{Spec}(P)$ such that $\sigma \circ x_1 = x_2$. Then the property of the pushout with vertex $\Sigma_{1,s}(x_1)$ proves that there is a unique arrow $\mathrm{R}_s(\sigma) : \Sigma_{1,s}(x_1) \to \Sigma_{2,s}(x_2)$ in $\mathbf{Spec}(P)$ such that:

$$\mathrm{R}_s(\sigma) \circ c_s(x_1) = c_s(x_2) \ \text{ and } \ \mathrm{R}_s(\sigma) \circ \tau_s(x_1) = \tau_s(x_2) \circ \sigma \ .$$



From $\mathrm{R}_s(\sigma) \circ c_s(x_1) = c_s(x_2)$, it follows that $\mathrm{R}_s(\sigma)$ is an arrow in $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$, so that R_s is an endofunctor of $\mathrm{Yon}(H) \downarrow \mathbf{Spec}(P)$. From $\mathrm{R}_s(\sigma) \circ \tau_s(x_1) = \tau_s(x_2) \circ \sigma$, it follows that τ_s is a natural transformation:

$$\tau_s : \mathrm{Id}_{\mathrm{Yon}(H)\downarrow\mathbf{Spec}(P)} \Longrightarrow \mathrm{R}_s \ .$$

□

Soundness of the diagrammatic logics.

The following result derives immediately from the definitions; it means that each diagrammatic logic is sound.

Theorem A.32 (soundness theorem) *Every* P*-entailment is a* P*-consequence with respect to* Δ*, for every* P*-domain* Δ*.*

Example A.33 The morphism of projective sketches $P_{\mathrm{comp}} : \mathcal{S}_{\mathrm{comp}} \to \overline{\mathcal{S}}_{\mathrm{comp}}$, from example A.16, is a propagator. The P_{comp}-specifications are the compositive graphs. For instance, the graph:

$$\Sigma_{\mathrm{nat}} \ = \ \overset{s}{\circlearrowright}\ \mathrm{Nat}$$

is a P_{comp}-specification such that:

$$\Sigma_{\mathrm{nat}}(\mathtt{Type}) = \{\mathrm{Nat}\} \ , \ \Sigma_{\mathrm{nat}}(\mathtt{Term}) = \{s\} \ , \ \Sigma_{\mathrm{nat}}(\mathtt{Comp}) = \emptyset \ .$$

The P_{comp}-domains are the saturated compositive graphs, hence each category is a P_{comp}-domain. For instance, the category of sets is a P_{comp}-domain **Set**. One of the models of Σ_{nat} with values in **Set** interprets the sort Nat as the set $\mathbb{N}$ of natural numbers and the operation s as the successor map.

A.6 Morphisms of propagators and links between logics

Let $\alpha : P_1 \to P_2$ be a morphism of propagators, as in definition A.12. Then each P_2-specification Σ_2 determines by omission a P_1-specification $\mathrm{G}_\alpha(\Sigma_2)$, and each P_1-specification Σ_1 generates a P_2-specification $\mathrm{F}_\alpha(\Sigma_1)$, and similarly for morphisms of specifications. This holds also for domains.

Denotational semantics.

The denotational semantics of P_1 and P_2 are related by the following result, which is an easy consequence of adjunction.

Proposition A.34 (links between denotational semantics) *Let Σ_1 be a P_1-specification and Δ_2 a P_2-domain. Then there is a natural bijection:*

$$\mathrm{Mod}_{P_1}(\Sigma_1, \mathrm{G}_\alpha(\Delta_2)) \simeq \mathrm{Mod}_{P_2}(\mathrm{F}_\alpha(\Sigma_1), \Delta_2) \ .$$

Proof. On the one hand, from the definition of P_1-models:

$$\mathrm{Mod}_{P_1}(\Sigma_1, \mathrm{G}_\alpha(\Delta_2)) = \mathrm{Hom}_{\mathbf{Dom}(P_1)}(\mathrm{F}_{P_1}(\Sigma_1), \mathrm{G}_\alpha(\Delta_2)) \ .$$

On the other hand, from the definition of P_2-models:

$$\mathrm{Mod}_{P_2}(\mathrm{F}_\alpha(\Sigma_1), \Delta_2) = \mathrm{Hom}_{\mathbf{Dom}(P_2)}(\mathrm{F}_{P_2}(\mathrm{F}_\alpha(\Sigma_1)), \Delta_2) \ ,$$

and from $\alpha \circ P_1 = P_2 \circ \alpha$:

$$\mathrm{F}_{P_2}(\mathrm{F}_\alpha(\Sigma_1)) \sim \mathrm{F}_\alpha(\mathrm{F}_{P_1}(\Sigma_1)) \ ,$$

hence:

$$\mathrm{Mod}_{P_2}(\mathrm{F}_\alpha(\Sigma_1), \Delta_2) \simeq \mathrm{Hom}_{\mathbf{Dom}(P_2)}(\mathrm{F}_\alpha(\mathrm{F}_{P_1}(\Sigma_1)), \Delta_2) \ .$$

Now, the adjunction with respect to α concludes the proof. □

Axiomatic semantics.

The axomatic semantics of P_1 and P_2 are related by the following result.

Proposition A.35 (links between axiomatic semantics) *Let $\sigma : \Sigma \to \Sigma'$ be a morphism of P_1-specifications. If σ is a P_1-entailment, then $\mathrm{F}_\alpha(\sigma)$ is a P_2-entailment.*

Proof. The assumption means that $\mathrm{F}_{P_1}(\sigma)$ is an isomorphism of P_1-domains, from which it follows that $\mathrm{F}_\alpha(\mathrm{F}_{P_1}(\sigma))$ is an isomorphism of P_2-domains. But from $\alpha \circ P_1 = P_2 \circ \alpha$, it follows that:

$$\mathrm{F}_\alpha \circ \mathrm{F}_{P_1} \sim \mathrm{F}_{P_2} \circ \mathrm{F}_\alpha \ .$$

Hence $\mathrm{F}_{P_2}(\mathrm{F}_\alpha(\sigma))$ is an isomorphism of P_2-domains, which means that $\mathrm{F}_\alpha(\sigma)$ is a P_2-entailment. □