# Twisted Systems and the Logic of Imperative Programs

Lindsay Errington

October 28, 1998

**Abstract**

Following Burstall, a flow diagram can be represented by a pair consisting of a graph and a functor from the free category to the category of sets and relations. A program is verified by incorporating the assertions of the Floyd-Naur proof method into a second functor and exhibiting a natural transformation to the program.

A broader range of properties is obtained by substituting spans for relations and introducing oplaxness into both the functors representing programs and the natural transformations in the morphisms between programs. The apparent complexity of this generalization is overcome by the observation that an oplax functor $\mathsf{J} \xrightarrow{\circ} \mathsf{Sp}(\mathsf{C})$ is essentially the same as a functor $\widetilde{\mathsf{J}} \longrightarrow \mathsf{C}$ where $\widetilde{\mathsf{J}}$ is the twisted arrow category of $\mathsf{J}$. Thus, a program is a presheaf $\widetilde{F(G)} \longrightarrow \mathsf{Set}$ as are the properties of the program.

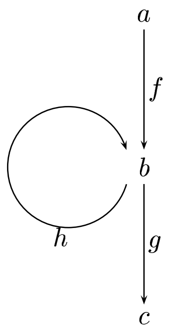
By analogy with categorical models of first-order logic, a program and the properties which pertain to it are subobjects of a suitably chosen base object. In this setting safety and liveness properties are dual in a fibre of subobjects.

## 1 Introduction

In [Bur72] Rod Burstall describes how a flow-chart can be represented by a functor from a free category to the category of sets and partial functions, or more generally, sets and relations. A program is a diagram or *system* in the sense used by Goguen [Gog91] consisting of a pair:

$$(G, F(G) \xrightarrow{S} \mathsf{Rel})$$

where $G$ is a graph and $F(G)$ is the free category. Diagram 1 illustrates this with the factorial program. The vertices of $G$ are the states or *program points*. The vertex $a$ is the start state and paths in $G$ are computation paths. The image of each vertex is a cartesian product with a component for each variable in scope. Tuples in a product are the possible variable assignments at a particular program point. Transitions are labelled with relations. In the example we use terms from a typed $\lambda$-calculus extended with predicates to denote partial functions.



```
fact(n : ℕ) : ℕ
    var x, y : ℕ
    x := 1
    y := 1
    do x < n  →
        x := x + 1
        y := x × y
    end
    return y
```

$$\begin{tikzcd}
N \arrow[d, "\lambda n.(n,1,1)"] \\
N \times N \times N \arrow[loop left, "\lambda(n,x,y).x < n.(n,x+1,x\times y)"] \arrow[d, "\lambda(n,x,y).x \geq n.y"] \\
N
\end{tikzcd}$$

Diagram 1.1.

A morphism of programs:

$$(G, F(G) \xrightarrow{S} \mathsf{Rel}) \longrightarrow (H, F(H) \xrightarrow{T} \mathsf{Rel})$$

is a pair $(P, \eta)$ where $P : G \longrightarrow H$ is a graph homomorphism and $\eta : S \Longrightarrow T \circ F(P)$ is an injective natural transformation. A morphism is a *simulation* when the components of $\eta$ are inclusions.

Less explicit in Burstall's paper is that such a functor is not only a representation of the program, but also its operational semantics. Operationally, each action is a conditional rewrite rule which can "fire" only for those tuples which satisfy the precondition.

As a specification of the operational semantics of a program, this representation has some advantages over a conventional structural operational semantics. First, variables are typed and the product at each program point correctly reflects the locality and scoping of variables. Equally important is that a system preserves the *coherence* of the program. By this we mean that it faithfully expresses the program as a schedule of operations. A conventional operational semantics is a relation between states. A state is a pair $\langle C, s \rangle$ where $C$ is a program fragment and $s$ is a particular variable assignment. The relation does not record the fact that some transitions are instances of the same action in the program. Thus a considerable amount of information concerning the structure of the program is lost. In this respect a functor like the above is more intensional and more faithfully represents the structure of the program. Preserving this information may prove to be significant within the context of model checking as a way of avoiding the state explosion problem.

Burstall goes on to cast the Floyd-Naur proof method into categorical terms. Given a program $(G, S : F(G) \longrightarrow \mathsf{Rel})$ and assertions $P_a$ for each vertex $a \in G$, he constructs a second functor $S' : F(G) \longrightarrow \mathsf{Rel}$ by restricting the relations in $S$ to the sets satisfying the assertions at each vertex. Verifying a program via the Floyd-Naur proof method amounts to exhibiting a simulation $\eta : S' \Longrightarrow S$.

Subsequently Goguen developed the theory further [Gog74] (See also Goguen and Meseguer [GM83]). Amongst other contributions, he allows a more general class of morphisms, $(P, \eta)$, whereby $P : F(G) \longrightarrow F(H)$ can be a functor (rather than a graph homomorphism). In this way it becomes possible to relate programs having different shapes and paths of different length.

This paper considers the relationship between programs and properties by analogy with categorical models of predicate logic. Given a context of variables, $\Gamma = [x_1 : X_1, \ldots, x_n : X_n]$, then a formula $\phi$ in the context $\Gamma$ is interpreted by a subobject of the product interpreting $\Gamma$. Similarly, a program and the properties which pertain to it are subobjects of a suitably chosen base object. In this setting, *safety* and *liveness* properties are dual to one another in a fibre of subobjects. A program has a safety property when it factors through the property. Conversely, a program has a liveness property when the property factors through the program.

Unfortunately, the range of properties which can be expressed using the categories proposed by Burstall and Goguen is limited. The constraints imposed by functorality and naturality are too strong. For instance, suppose $f : a \longrightarrow b \in F(G)$ is a transition and $Sf$ and $Tf$ are relations associated with $f$ from a program and safety property respectively. Naturality implies that $Sf$ and $Tf$ must be the same for all $x \in Sa$ and that $Sf$ can only be a restriction of $Tf$ to the smaller domain and codomain.

$$\begin{tikzcd}
Sa \arrow[r, tail, "\eta_a"] \arrow[d, "Sf"'] & Ta \arrow[d, "Tf"] \\
Sb \arrow[r, tail, "\eta_b"'] & Tb
\end{tikzcd}$$

This amounts to an over specification since is means that the relations in a property can specify little more than the input/output relation of the corresponding steps in the program. The relations in a property cannot be made arbitrarily larger than those in the program. It is not possible, therefore, to express only the postcondition of a transition. Nor can one assert a property concerning a single program variable from the many which may be in scope.

Functorality implies that it is not possible to specify the property of a composite transition without imposing constraints for the steps along the path. For example, one cannot assert that a particular computation path must realize some input/output relation without also specifying how that relation is realized.

To address these issues we move to a more general setting. Intuitively we substitute *spans* in place of relations. Then, in place of functors, we use *oplax functors* and in place of natural transformations we use *oplax map natural transformations*. Thus, a system is a pair

$$(G, F(G) \overset{S}{\mathrel{-\!\!\circ\!\!\rightarrow}} \mathsf{Sp}(\mathsf{C}))$$

in which $S$ is oplax and a morphism is pair $(P, \eta)$ where $\eta$ is an oplax map natural transformation. These terms will be defined shortly.

Bicategories, oplax functors and oplax natural transformations are all subject to coherence conditions so it may seem like we have introduced a considerable degree of complexity for a relatively modest improvement in expressiveness. However, the functor category just outlined has a surprisingly simple characterization in which the coherence conditions are not needed. An oplax functor $\mathsf{J} \mathrel{-\!\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ is essentially "the same" as a functor $\widetilde{\mathsf{J}} \longrightarrow \mathsf{C}$ where $\widetilde{\mathsf{J}}$ is the *twisted arrow category* of $\mathsf{J}$. This gives *twisted systems*, *ie.* pairs:

$$(\mathsf{J}, \widetilde{\mathsf{J}} \longrightarrow \mathsf{C})$$

In particular, a program with shape $G$ is a presheaf $\widetilde{F(G)} \longrightarrow \mathsf{Set}$ as are the properties of that program. Oplax map natural transformations become simply natural transformations. Thus, by introducing twisted arrow categories, we not only obtain a richer range of properties, but we also find ourselves working in a presheaf topos.

The question of what is a suitable logic for describing properties of programs is not addressed in this paper. However, the theory ensures that properties can be combined using the standard propositional connectives. Moreover, it becomes possible to quantify over both local and global program variables, recursive instances of variables or individual occurrences of variables.

Twisted systems also appear in [Err96] where they are used to construct categories of processes. The thesis [Err] gives categorical semantics to a message passing parallel language which is an extension of the sequential language defined later in this paper.

The remainder of the paper is structured as follows. In the next section we review results concerning the relationship between twisted arrow categories and spans. This includes some basic results concerning twisted arrow categories. The reader is referred to [Err] for proofs and further details. Next we introduce categorical transition systems and twisted systems. From the latter we construct a rudimentary bicategory of algorithms which is then used to give a categorical (and compositional) semantics to a simple imperative language. We digress briefly to outline the relationship between the categorical semantics and a conventional structural operational semantics. Finally we consider programs and properties (both safety and liveness) in a setting akin to categorical models of predicate logic.

## 2 Bicategories, spans and oplaxness

The following is adapted from Borceux [Bor94]. See also Benabou [Bén67].

DEFINITION 2.1 *A* bicategory $\mathsf{B}$ *consists of the following data:*

*(i) a collection of objects* $A, B, C, \ldots$ *called* 0*-cells.*

*(ii) for each pair of objects,* $A$ *and* $B$*, a small category* $\mathsf{B}(A,B)$*. The objects of* $\mathsf{B}(A,B)$ *are called morphisms or* 1*-cells of* $\mathsf{B}$ *and the morphisms of* $\mathsf{B}(A,B)$ *are the* 2*-cells.*

*(iii) For each triple of objects* $A$*,* $B$ *and* $C$*, a composition functor*

$$c_{ABC} : \mathsf{B}(A,B) \times \mathsf{B}(B,C) \longrightarrow \mathsf{B}(A,C)$$

*where we write* $g \circ f$ *for the composition of* 1*-cells* $f : A \longrightarrow B$ *and* $g : B \longrightarrow C$*.*

*(iv) for each object* $A$*, an identity arrow* $id_A : A \longrightarrow A$*.*

*The usual axioms for identity and associativity are relaxed to isomorphisms:*

*(v) Associativity: for each quadruple $A$, $B$, $C$, and $D$ a natural isomorphism*

$$\mathsf{assoc}_{ABCD} : c_{ACD} \circ (c_{ABC} \times Id_{CD}) \Longrightarrow c_{ABD} \circ (Id_{AB} \times c_{BCD})$$

*(vi) Unit: for each pair of objects $A$ and $B$, two natural isomorphisms*

$$\begin{aligned} \mathsf{unitl}_{AB} &: Id_{AB} \Longrightarrow c_{AAB} \circ id_A \times Id_{AB} \\ \mathsf{unitr}_{AB} &: Id_{AB} \Longrightarrow c_{ABB} \circ Id_{AB} \times id_B \end{aligned}$$

*These must satisfy coherence conditions which we omit.*

One can quotient the morphisms of a bicategory to obtain a category. Following Bénabou, the category obtained by identifying all 1-cells which are 2-isomorphic and forgetting the 2-cells is the *classifying category*. We write $\mathsf{B}^\flat$ for the classifying category of $\mathsf{B}$.

Given a category $\mathsf{C}$ with pullbacks, the *bicategory of spans* $\mathsf{Sp}(\mathsf{C})$ has as 0-cells the objects of $\mathsf{C}$. A 1-cell $f : A \longrightarrow B$ is a *span*; a diagram in $\mathsf{C}$ of the form:

$$\begin{tikzcd}
& P_f \arrow[ld, "\lambda_f"'] \arrow[rd, "\rho_f"] & \\
A & & B
\end{tikzcd}$$

where $\lambda_f$ and $\rho_f$ are respectively the left and right *legs* and $P_f$ is the *apex* of $f$. Composition of 1-cells is by pullback. A 2-cell $\alpha : f \Longrightarrow g$ is a morphism in $\mathsf{C}$ such that the two triangles commute:

$$\begin{tikzcd}
& P_f \arrow[ld] \arrow[rd] \arrow[dd, "\alpha"'] & \\
A & & B \\
& P_g \arrow[lu] \arrow[ru] &
\end{tikzcd}$$

Some familiar 2-categories are obtained by restricting the choice of morphisms in spans and quotienting by 2-isomorphisms. For example, requiring that $\lambda_f$ be monic for all $f$ yields the 2-category $\mathsf{Part}(\mathsf{C})$ of partial maps over $\mathsf{C}$. Note that the 2-cells in $\mathsf{Part}(\mathsf{C})$ are all monomorphisms. See Robinson and Rosolini [RR88] for other notions of categories of partial maps.

Choosing $\mathsf{C}$ to be a regular category and requiring that $\lambda_f$ and $\rho_f$ are jointly monic yields the category $\mathsf{Rel}(\mathsf{C})$ of relations over $\mathsf{C}$. The composite of

$f$ and $g$ in $\mathsf{Rel}(\mathsf{C})$ is defined by first constructing $gf$ in $\mathsf{Sp}(\mathsf{C})$ and then taking the image factorization of $(\lambda_{gf}, \rho_{gf})$ as a regular epimorphism followed by a jointly monic pair. As with $\mathsf{Part}(\mathsf{C})$, the 2-cells in $\mathsf{Rel}(\mathsf{C})$ are monomorphisms. See Freyd and Scedrov [FS90].

The following is taken from Carboni *et al* [CKS84].

DEFINITION 2.2 *A morphism $f : A \longrightarrow B$ in a bicategory $\mathsf{B}$ is a* map *when it has a right adjoint $f^* : B \longrightarrow A$.*

LEMMA 2.3 *A morphism $f : A \longrightarrow B$ in $\mathsf{Sp}(\mathsf{C})$ is a map if and only if $\lambda_f$ is an isomorphism in $\mathsf{C}$.*

The lemma holds in $\mathsf{Rel}(\mathsf{C})$ and $\mathsf{Part}(\mathsf{C})$ as well. Restricting to maps yields a sub-bicategory of $\mathsf{Sp}(\mathsf{C})$ in which the only 2-cells are isomorphisms. Quotienting by these 2-cells yields a category isomorphic to $\mathsf{C}$. Clearly each equivalence class has a canonical representative, namely the span whose backward leg is an identity. For further results concerning maps in bicategories of spans and relations see Carboni *et al* [CKS84] and Freyd and Scedrov [FS90].

In the same way that bicategories relax the associativity and unit axioms, one can relax the axioms for functors between bicategories and natural transformations. Below we define oplax functors and oplax natural transformations in the particular case where the source bicategory is trivial (the only 2-cells are identities) and the target is $\mathsf{Sp}(\mathsf{C})$.

DEFINITION 2.4 *Let $\mathsf{J}$ be a trivial bicategory. An* oplax functor $F : \mathsf{J} \mathrel{-\!\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ *consists of the following data:*
*(i) For every object $a \in \mathsf{J}$, an object $Fa \in \mathsf{Sp}(\mathsf{C})$.*
*(ii) for every pair of objects $a, b \in \mathsf{J}$, a functor*

$$F_{ab} : \mathsf{J}(a,b) \longrightarrow \mathsf{Sp}(\mathsf{C})(Fa, Fb)$$

*(iii) for every triple, $a, b, c \in \mathsf{J}$, a natural transformation $\gamma_{abc}$:*

$$\begin{tikzcd}
\mathsf{J}(a,b) \times \mathsf{J}(b,c) \arrow[r, "c_{abc}"] \arrow[d, "F_{ab} \times F_{bc}"'] & \mathsf{J}(a,c) \arrow[d, "F_{ac}"] \arrow[dl, Rightarrow, shorten=12mm, "\gamma_{abc}"'] \\
\mathsf{Sp}(\mathsf{C})(Fa,Fb) \times \mathsf{Sp}(\mathsf{C})(Fb,Fc) \arrow[r, "c_{Fa,Fb,Fc}"'] & \mathsf{Sp}(\mathsf{C})(Fa,Fc)
\end{tikzcd}$$

*(iv) for every object $a \in \mathsf{J}$, a natural transformation $\delta_a$:*

$$\begin{tikzcd}
1 \arrow[r, "u_a"] \arrow[d, equal] & \mathsf{J}(a,a) \arrow[d, "F_{aa}"] \arrow[dl, phantom, ""{name=U}] \\
1 \arrow[r, "u_{Fa}"'] & \mathsf{Sp}(\mathsf{C})(Fa,Fa)
\arrow[from=1-2, to=2-1, Rightarrow, shorten=12pt, "\delta_a"']
\end{tikzcd}$$

*The natural transformations must satisfy coherence conditions which we omit.*

The $f, g$ component of $\gamma_{abc}$ is shown in diagram 2.1. If the target is $\mathsf{Rel}(\mathsf{C})$ or $\mathsf{Part}(\mathsf{C})$ then $\gamma_{f,g}$ is monic giving the inequality $F(gf) \subseteq Fg \circ Ff$. Note that the naturality of $\gamma_{f,g}$ and $\delta_a$ is vacuous as there are only identity 2-cells in $\mathsf{J}$.

$$\begin{tikzcd}
 & & P_{F(gf)} \arrow[d, "\gamma_{f,g}"] \arrow[ddll, bend right] \arrow[ddrr, bend left] & & \\
 & & P_{Fg} \times_{Fb} P_{Fg} \arrow[dl] \arrow[dr] \arrow[dd, phantom, very near start, "\vee"] & & \\
 & P_{Ff} \arrow[dl] \arrow[dr] & & P_{Fg} \arrow[dl] \arrow[dr] & \\
Fa & & Fb & & Fc
\end{tikzcd}$$

Diagram 2.1.

Definition 2.5 *An oplax functor* $\mathsf{J} \mathrel{-\!\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ *is* normalized *if for each object* $a \in \mathsf{J}$, $\delta_a$ *is the identity natural transformation.*

When $F$ is normalized then it follows that for all $f : a \longrightarrow b$ in $\mathsf{J}$, $\gamma_{id_a,f} = \mathsf{unitl}_{Ff}$ and $\gamma_{f,id_b} = \mathsf{unitr}_{Ff}$.

Definition 2.6 *Let* $F, G : \mathsf{J} \mathrel{-\!\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ *be oplax functors. A* oplax natural transformation $\alpha : F \mathrel{=\!\!\circ\!\!\Rightarrow} G$ *consists of the following data:*
*(i) for every object* $a$ *in* $\mathsf{J}$ *a morphism* $\alpha_a : Fa \longrightarrow Ga$ *in* $\mathsf{Sp}(\mathsf{C})$.
*(ii) for each pair of objects* $a, b \in \mathsf{J}$ *a natural transformation* $\tau_{ab}$*:*

$$\begin{tikzcd}
\mathsf{J}(a,b) \arrow[r, "F_{ab}"] \arrow[d, "G_{ab}"'] & \mathsf{Sp}(\mathsf{C})(Fa,Fb) \arrow[d, "c_{Fa,Fb,Gb}(-{,}\alpha_b)"] \\
\mathsf{Sp}(\mathsf{C})(Ga,Gb) \arrow[r, "c_{Fa,Ga,Gb}(\alpha_a{,}-)"'] & \mathsf{Sp}(\mathsf{C})(Fa,Gb)
\arrow[from=1-2, to=2-1, Rightarrow, shorten=25pt, "\tau_{ab}"']
\end{tikzcd}$$

*where $c_{Fa,Ga,Gb}(\alpha_a, -)$ and $c_{Fa,Fb,Gb}(-, \alpha_b)$ are the functors obtained by fixing $\alpha_a$ or $\alpha_a$ in the bifunctors of composition. Once again we omit the coherence conditions.*

Diagram 2.2 shows oplax naturality for $f : a \longrightarrow b$ in $\mathsf{J}$ where $\tau_f$ is the $f$ component of the natural transformation $\tau_{ab}$. The definition translates to the requirement that the pentagons $\theta$ and $\phi$ must commute. The difference between this and a naturality diagram is simply that in the latter $\tau_f$ is the identity. For functors in $\mathsf{Rel}(\mathsf{C})$, $\tau_f$ is monic yielding the inequality: $\alpha_b \circ Ff \subseteq Gf \circ \alpha_a$.

$$\begin{tikzcd}
Fa & & \overset{\alpha_a}{\bullet} \arrow[ll] \arrow[rr] & & Ga \\
& \theta & & \bullet \arrow[ul] \arrow[dr] & \\
Ff \bullet \arrow[uu] \arrow[dd] & & & & \bullet Gf \arrow[uu] \arrow[dd] \\
& \bullet \arrow[ul] \arrow[dr] \arrow[uurr, dotted, "\tau_f"] & & \phi & \\
Fb & & \underset{\alpha_b}{\bullet} \arrow[ll] \arrow[rr] & & Gb
\end{tikzcd}$$

Diagram 2.2.

We write $\mathsf{L}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ for the bicategory with objects the functors $\mathsf{J} \longrightarrow \mathsf{Sp}(\mathsf{C})$ and morphisms the oplax natural transformations. Similarly, write $\mathsf{LL}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ for the bicategory which is the same as above except that objects are normalized oplax functors $\mathsf{J} \multimap \mathsf{Sp}(\mathsf{C})$ (rather than functors).

Lemma 2.7 $\mathsf{L}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ *and* $\mathsf{LL}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ *are bicategories.*

Lemma 2.8 *A morphism $\alpha$ is a map in* $\mathsf{L}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ *and* $\mathsf{LL}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ *if and only if the* 1*-cells components of $\alpha$ are maps in* $\mathsf{Sp}(\mathsf{C})$.

Now write $\mathsf{LM}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ and $\mathsf{LLM}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ for the sub-bicategories of $\mathsf{L}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ and $\mathsf{LL}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ respectively in which morphisms are restricted to be maps. As with $\mathsf{Sp}(\mathsf{C})$, the map restriction implies that the only 2-cells in $\mathsf{LM}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ and $\mathsf{LLM}(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ are isomorphisms. Moreover, each equivalence class in the corresponding classifying categories, $\mathsf{LM}^\flat(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ and $\mathsf{LLM}^\flat(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ has a canonical representative, namely the oplax natural transformation $\alpha$ such that for each 1-cell component $\alpha_a$, $\lambda_{\alpha_a} = id_{Fa}$. In effect, morphisms in these two categories are families of arrows of $\mathsf{C}$ indexed by the *morphisms* of $\mathsf{J}$. This will become clear shortly.

# 3 Twisted arrow categories

An oplax functor $F : \mathsf{J} \mathrel{-\!\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ selects not only a collection of objects and arrows in $\mathsf{Sp}(\mathsf{C})$ but indirectly a collection of objects and arrows in $\mathsf{C}$. This suggests that rather than oplax functors into $\mathsf{Sp}(\mathsf{C})$, we incorporate the spans into the domain and work with *functors* into $\mathsf{C}$. This requires a shape $\widetilde{\mathsf{J}}$ derived from $\mathsf{J}$ and a functor $F^* : \widetilde{\mathsf{J}} \longrightarrow \mathsf{C}$ which selects the same or an equivalent collection of objects and morphisms as $F$. The shape must be such that for every morphism in $\mathsf{J}$ there is a "span" in $\widetilde{\mathsf{J}}$. These is obtained by taking $\widetilde{\mathsf{J}}$ to be the *twisted arrow category* of $\mathsf{J}$ (see chapter IX, §6, exercise 3 of Mac Lane [Mac71]):

Definition 3.1 *An object $\widetilde{f}$ of a twisted arrow category $\widetilde{\mathsf{J}}$ is an arrow $f$ of $\mathsf{J}$. A morphism $\widetilde{f} \longrightarrow \widetilde{g}$ is a pair of morphisms $(l, m)$ in $\mathsf{J}$ such that $f = mgl$.*[1]

$$\begin{tikzcd}
A \arrow[r, "l"] \arrow[d, "f"'] & C \arrow[d, "g"] \\
B & D \arrow[l, "m"]
\end{tikzcd}$$

*The composition of $(l, m) : \widetilde{f} \longrightarrow \widetilde{g}$ and $(p, q) : \widetilde{g} \longrightarrow \widetilde{h}$ is $(pl, mq) : \widetilde{f} \longrightarrow \widetilde{h}$.*

An example of a category and its "twist" is shown in diagram 3.1 where, in the twisted category, we have written $a$ in place of $id_a$. The collection of objects $\widetilde{a}$ of $\widetilde{\mathsf{J}}$ associated with identity morphisms of $\mathsf{J}$ are referred to as the $\mathsf{J}$ *objects in* $\widetilde{\mathsf{J}}$.

The twisted arrow category $\widetilde{\mathsf{J}}$ displays the factorization structure of $\mathsf{J}$. There is a morphism $\widetilde{f} \longrightarrow \widetilde{g}$ for each factorization of $f$ involving $g$. The relationship between spans and twists rests on the property that each morphism $f : a \longrightarrow b$ in $\mathsf{J}$ determines a span $\widetilde{a} \xleftarrow{(a,f)} \widetilde{f} \xrightarrow{(f,b)} \widetilde{b}$ in $\widetilde{\mathsf{J}}$.

The following are easily verified.

Lemma 3.2 $(\widetilde{\mathsf{J}})^{op} = \int(\mathsf{J}^{op} \times \mathsf{J} \xrightarrow{hom} \mathsf{Set})$

This implies that $\widetilde{\mathsf{J}}$ is fibered over $\mathsf{J}^{op} \times \mathsf{J}$ for which there are projections: $\overrightarrow{\pi}, : \widetilde{\mathsf{J}} \longrightarrow \mathsf{J}$ and $\overleftarrow{\pi} : \widetilde{\mathsf{J}} \longrightarrow \mathsf{J}^{op}$:

$$\begin{array}{rclcrcl}
\overrightarrow{\pi}(\widetilde{f} : a \longrightarrow b) & = & a & \qquad & \overleftarrow{\pi}(\widetilde{f} : a \longrightarrow b) & = & b \\
\overrightarrow{\pi}(l, m) & = & l & & \overleftarrow{\pi}(l, m) & = & m
\end{array}$$

[1] This is the opposite of the category appearing in [Mac71].

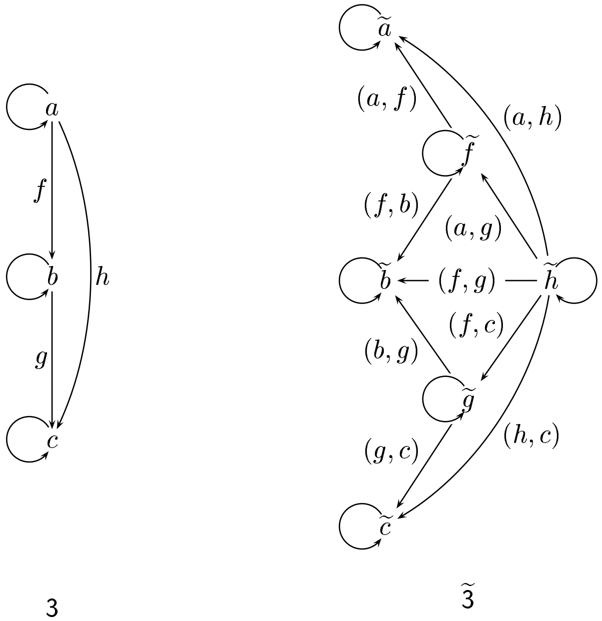


Diagram 3.1.

Lemma 3.3 $(\widetilde{-}) : \mathsf{Cat} \longrightarrow \mathsf{Cat}$ *is a functor.*

Proof Given $F : \mathsf{J} \longrightarrow \mathsf{K}$, define $\widetilde{F} : \widetilde{\mathsf{J}} \longrightarrow \widetilde{\mathsf{K}}$ by:

$$\widetilde{F}\widetilde{f} = \widetilde{Ff}$$
$$\widetilde{F}(l,m) = (Fl, Fm)$$

which is clearly functorial. □

Lemma 3.4 *If* $\mathsf{J}$ *is a discrete category then* $\mathsf{J} \cong \widetilde{\mathsf{J}}$.

Lemma 3.5 $(\widetilde{-}) : \mathsf{Cat} \longrightarrow \mathsf{Cat}$ *preserves all limits.*

Proof We show that the twist of a limiting cone in $\mathsf{Cat}$ is isomorphic to the limit of the twisted diagram. As is standard, it suffices to consider the terminal object (the empty diagram), products and equalizers.

By lemma 3.4 the terminal object is preserved. The situation for products is shown below. On the left is a product diagram in $\mathsf{Cat}$ and on the right is the twist of the same diagram together and a second product.

$$\begin{tikzcd}
\mathsf{J} & \mathsf{J}\times\mathsf{K} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \mathsf{K}
\end{tikzcd}
\qquad
\begin{tikzcd}
\widetilde{\mathsf{J}} & \widetilde{\mathsf{J}\times\mathsf{K}} \arrow[l, "\widetilde{\pi}_1"'] \arrow[r, "\widetilde{\pi}_2"] \arrow[d, dotted, bend left, "F"] & \widetilde{\mathsf{K}} \\
 & \widetilde{\mathsf{J}}\times\widetilde{\mathsf{K}} \arrow[ul, "\rho_1"] \arrow[ur, "\rho_2"'] \arrow[u, bend left, "G"] &
\end{tikzcd}$$

Here $F$ universal and $G$ is defined such that given morphisms $(l,m):\widetilde{f}\longrightarrow\widetilde{g}\in\widetilde{\mathsf{J}}$ and $(p,q):\widetilde{h}\longrightarrow\widetilde{k}\in\widetilde{\mathsf{K}}$ then:

$$G\big((\widetilde{f},\widetilde{h})\xrightarrow{((l,m),(p,q))}(\widetilde{g},\widetilde{k})\big)=\big(\widetilde{(f,h)}\xrightarrow{((l,p),(m,q))}\widetilde{(g,k)}\big)$$

This extends to a functor and clearly an isomorphism (the inverse of $F$).

With respect to equalizers, consider the diagram below showing an equalizer diagram and its twist.

$$\begin{tikzcd}
\mathsf{L} \arrow[r, tail, "E"] & \mathsf{J} \arrow[r, shift left, "F"] \arrow[r, shift right, "G"'] & \mathsf{K}
\end{tikzcd}
\qquad
\begin{tikzcd}
\widetilde{\mathsf{L}} \arrow[r, tail, "\widetilde{E}"] \arrow[d, dotted, bend left] & \widetilde{\mathsf{J}} \arrow[r, shift left, "\widetilde{F}"] \arrow[r, shift right, "\widetilde{G}"'] & \widetilde{\mathsf{K}} \\
\mathsf{M} \arrow[ur, tail, "H"'] \arrow[u, bend left] & &
\end{tikzcd}$$

Let $\mathsf{M}\overset{H}{\rightarrowtail}\widetilde{\mathsf{J}}$ be the equalizer following the twist. The categories $\mathsf{M}$ and $\widetilde{\mathsf{L}}$ are isomorphic by the following:

$$\begin{aligned}
&(l,m):\widetilde{f}\longrightarrow\widetilde{g}\in\mathsf{M}\\
&\qquad\iff \widetilde{F}(l,m)=\widetilde{G}(l,m)\wedge\widetilde{F}\widetilde{f}=\widetilde{G}\widetilde{f}\wedge\widetilde{F}\widetilde{g}=\widetilde{G}\widetilde{g}\wedge f=mgl\\
&\qquad\iff (Fl,Fm)=(Gl,Gm)\wedge Ff=Gf\wedge Fg=Gg\wedge f=mgl\\
&\qquad\iff Fl=Gl\wedge Fm=Gm\wedge Ff=Gf\wedge Fg=Gg\wedge f=mgl\\
&\qquad\iff f=mgl\in\mathsf{L}\\
&\qquad\iff (l,m):\widetilde{f}\longrightarrow\widetilde{g}\in\widetilde{\mathsf{L}}
\end{aligned}$$

□

The category $\widetilde{\mathsf{J}}$ has some important limits.

Lemma 3.6 (Lamarche) *If $a\xrightarrow{f}b\xrightarrow{g}c$ is a composable pair in $\mathsf{J}$ then the following diagram is a pullback in $\widetilde{\mathsf{J}}$:*

$$\begin{tikzcd}
& \widetilde{gf} \arrow[dl, "{(a,g)}"'] \arrow[dr, "{(f,c)}"] \arrow[dd, "{(f,g)}" description] & \\
\widetilde{f} \arrow[dr, "{(f,b)}"'] & & \widetilde{g} \arrow[dl, "{(b,g)}"] \\
& \widetilde{b} &
\end{tikzcd}$$

Proof Let $k = mbfl = qgbp$ (diagram 3.2). The morphisms $a$, $b$ and $c$ are all identities which makes $(l, q)$ universal. □

$$\begin{tikzcd}
& \widetilde{k} \arrow[d, dotted, "{(l,q)}" description] \arrow[ddl, bend right, "{(l,m)}"'] \arrow[ddr, bend left, "{(p,q)}"] & \\
& \widetilde{gf} \arrow[dl, "{(a,g)}"'] \arrow[dr, "{(f,c)}"] \arrow[dd, "{(f,g)}" description] & \\
\widetilde{f} \arrow[dr, "{(f,b)}"'] & & \widetilde{g} \arrow[dl, "{(b,g)}"] \\
& \widetilde{b} &
\end{tikzcd}$$

Diagram 3.2.

Pullbacks in $\widetilde{\mathsf{J}}$ which arise in this way are called $\circ$-*pullbacks*. More generally, every finite string of composable arrows in $\mathsf{J}$ gives rise to a limiting cone in $\widetilde{\mathsf{J}}$. We call these $\circ$-*limits*. In this sense composition in $\mathsf{J}$ corresponds to limits in $\widetilde{\mathsf{J}}$.

The purpose of introducing twisted arrow categories was to enable an *oplax functor* $\mathsf{J} \mathrel{-\!\circ\!\!\rightarrow} \mathsf{Sp}(\mathsf{C})$ to be replaced by a *functor* $\widetilde{\mathsf{J}} \longrightarrow \mathsf{C}$. The precise relationship between spans and twists is expressed in the following theorem.

Theorem 3.7 *The quotient functor category* $\mathsf{LLM}^\flat(\mathsf{J}, \mathsf{Sp}(\mathsf{C}))$ *is isomorphic to* $\mathsf{C}^{\widetilde{\mathsf{J}}}$*.*

So whereas the we began with the prospect of working with bicategories, oplax functors and oplax natural transformations, theorem 3.7 reduces this to a well understood universe. In particular, when the target category is $\mathsf{Set}$, then $\mathsf{LLM}^\flat(\mathsf{J}, \mathsf{Sp}(\mathsf{Set}))$ is isomorphic to the category of presheaves $\mathsf{Set}^{\widetilde{\mathsf{J}}}$ and hence a topos.

Corollary 3.8 *The functor category* $\mathsf{LM}^\flat(\mathsf{J},\mathsf{Sp}(\mathsf{C}))$ *is isomorphic to the subcategory of* $\mathsf{C}^{\widetilde{\mathsf{J}}}$ *consisting of those functors which preserve* $\circ$*-limits.*

Corollary 3.9 *The functor category* $\mathsf{LLM}^\flat(\mathsf{J},\mathsf{Part}(\mathsf{C}))$ *is isomorphic to the subcategory of* $\mathsf{C}^{\widetilde{\mathsf{J}}}$ *consisting of those functors with the property that for all morphisms* $f : a \longrightarrow b$ *in* $\mathsf{J}$*, the image of* $(a,f) : \widetilde{f} \longrightarrow \widetilde{a}$ *is monic.*

It is worth mentioning that the analogous correspondence for relations does not hold. That is, it is not the case that the functor category $\mathsf{LLM}^\flat(\mathsf{J},\mathsf{Rel}(\mathsf{C}))$ is isomorphic to the subcategory of $\mathsf{C}^{\widetilde{\mathsf{J}}}$ of functors where for all $f : a \longrightarrow b$ in $\mathsf{J}$, the image of $(a,f) : \widetilde{f} \longrightarrow \widetilde{a}$ and $(f,b) : \widetilde{f} \longrightarrow \widetilde{b}$ is a monic pair. This is because composition is $\mathsf{Rel}(\mathsf{C})$ is such that, given $gf$ in $\mathsf{J}$ and an oplax functor $\mathsf{J} \multimap \mathsf{Rel}(\mathsf{C})$, the 2-cell $\gamma_{f,g}$ *does not* determine morphisms from the apex $P_{gf}$ to $P_f$ and $P_g$.

# 4 Algorithmic functors

In this section we restrict our attention to twisted functors which correspond to algorithms in a very general sense.

Definition 4.1 *A twisted functor* $\widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ *is* algorithmic *if it preserves* $\circ$*-limits in* $\widetilde{\mathsf{J}}$.

By Corollary 3.8, the requirement that an algorithmic functor $F : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ preserve $\circ$-limits means that it corresponds to a functor $\mathsf{J} \longrightarrow \mathsf{Sp}(\mathsf{C})$. In fact to preserve $\circ$-limits, it suffices that the functor preserve $\circ$-pullbacks. The reason for the name *algorithmic* stems from the fact that, given a computation path $f = a_0 \xrightarrow{f_1} \cdots \xrightarrow{f_n} a_n$ in $\mathsf{J}$, then the *effect* of following the path is given by the composite span $Fa \xleftarrow{(a_0,f)} F\widetilde{f} \xrightarrow{(f,a_n)} Fa_n$.

Write $\mathsf{Set}^{\widetilde{\mathsf{J}}}_{alg}$ for the subcategory of algorithmic functors. This category has useful structure. First, since limits commute with limits and since $\mathsf{Set}$ is distributive, then, like $\mathsf{Set}^{\widetilde{\mathsf{J}}}$, it is complete and cocomplete. Also, $\mathsf{Set}^{\widetilde{\mathsf{J}}}_{alg}$ is well-complete. Intersections are formed from pullbacks. Unions can be constructed from coproducts and image factorizations. The following two lemmas ensure that these constructions yield algorithmic functors.

Lemma 4.2 *If* $F, G : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ *are algorithmic then* $F + G$ *is algorithmic.*

Proof It is straightforward to show that the sum of a pair of pullbacks in $\mathsf{Set}$ is again a pullback. Alternatively the lemma is an instance of a more general result concerning the interchange of limits and colimits in categories like $\mathsf{Set}$. See Borceux [Bor94]. □

Lemma 4.3 *Let $F, G : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ be algorithmic. If $\eta = F \overset{\epsilon}{\twoheadrightarrow} Q \overset{\mu}{\rightarrowtail} G$ is an epi-mono factorization of $\eta$, then $Q$ is algorithmic.*

Proof We shall prove a more general case when the target is any regular category rather than $\mathsf{Set}$. Let $a \xrightarrow{f} b \xrightarrow{g} c$ be a pair of morphisms in $\mathsf{J}$. Consider the image factorization of the components of $\eta$ between the induced $\circ$-pullbacks as shown in diagram 4.1. Assume $Q\widetilde{f} \xleftarrow{p} X \xrightarrow{q} Q\widetilde{g}$ is the pullback as shown. There are universal maps $h : F\widetilde{gf} \longrightarrow X$, $k : X \longrightarrow F\widetilde{gf} \longrightarrow X$ and $l : Q\widetilde{gf} \longrightarrow X$ where the latter is necessarily monic.

$$\begin{tikzcd}
 & F\widetilde{gf} \arrow[rrr, two heads, "\epsilon_{gf}"] \arrow[ddl, "F(a,g)"'] \arrow[ddd] \arrow[d, phantom, "\lrcorner"] & & & Q\widetilde{gf} \arrow[rr, tail, "\mu_{gf}"] \arrow[ddll] \arrow[ddd] & & G\widetilde{gf} \arrow[ddl, "G(a,g)"'] \arrow[ddd, "G(f,c)"] \arrow[d, phantom, "\lrcorner"] \\
 & & & X \arrow[dl] \arrow[ddr] & & & \\
F\widetilde{f} \arrow[rr, two heads, "\epsilon_f"] \arrow[dd, "F(f,b)"'] & & Q\widetilde{f} \arrow[rrr, tail, "\mu_f"] \arrow[dd] & & & G\widetilde{f} \arrow[dd] & \\
 & F\widetilde{g} \arrow[rrr, two heads, "\epsilon_g"] \arrow[dl, "F(b,g)"] & & & Q\widetilde{g} \arrow[rr, tail, "\mu_g"] \arrow[dll] & & G\widetilde{g} \arrow[dl, "G(b,g)"] \\
Fb \arrow[rr, two heads, "\epsilon_b"'] & & Qb \arrow[rrr, tail, "\mu_b"'] & & & Gb &
\end{tikzcd}$$

Diagram 4.1.

First we show that $k$ is a monomorphism. Let $Y \underset{v}{\overset{u}{\rightrightarrows}} X$ be a parallel pair. Then:

$$\begin{aligned}
ku = kv &\implies G(a,g)ku = G(a,g)kv \wedge G(f,c)ku = G(f,c)kv \\
&\implies \mu_f pu = \mu_f pv \wedge \mu_g qu = \mu_g qv \\
&\implies pu = pv \wedge qu = qv \\
&\implies u = v
\end{aligned}$$

Let $Z \underset{z}{\overset{w}{\rightrightarrows}} F\widetilde{gf}$ be the parallel pair for which $\epsilon_{gf}$ is the coequalizer. Therefore $h = l\epsilon_{gf}$ also coequalizes $w$ and $z$. This gives a universal monomorphism $X \longrightarrow Q\widetilde{gf}$ and therefore an isomorphism from which we conclude that $Q$ is algorithmic. □

Recall that for any functor $F : \mathsf{J} \longrightarrow \mathsf{Set}$, there is a standard discrete fibration over $\mathsf{J}$, $\pi : (\int F) \longrightarrow \mathsf{J}$. Objects in the category of elements, $\int F$, are pairs $(a, x)$ where $a$ is an object in $\mathsf{J}$ and $x \in Fa$. A morphism $(a, x) \longrightarrow (b, y)$ is a morphism $f : a \longrightarrow b$ such that $(Ff)(x) = y$. In the case of algorithmic functors, there is an alternative fibration over $\mathsf{J}$ rather than $\widetilde{\mathsf{J}}$.

Definition 4.4 (Lamarche) *Let $F : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ be an algorithmic functor. Define the category of* twisted elements, *$\oint_{\mathsf{J}} F$, by:*

*(i) objects are pairs $(a, x)$ where $a$ is an object of $\mathsf{J}$ and $x \in F\widetilde{a}$.*

*(ii) a morphism $(a, x) \longrightarrow (b, y)$ is a pair $(f, z)$ where $f : a \longrightarrow b$ is a morphism of $\mathsf{J}$ and $z$ is an element of $F\widetilde{f}$ such that $F(a, f)(z) = x$ and $F(f, b)(z) = y$.*[2]

*(iii) Given $(a, u) \xrightarrow{(f,x)} (b, v) \xrightarrow{(g,y)} (c, w)$, then $x \in F\widetilde{f}, y \in F\widetilde{g}, v \in F\widetilde{b}, F(f, b)(x) = v$, and $F(b, g)(y) = v$. $F$ preserves $\circ$-pullbacks, hence there exists a unique $z \in F\widetilde{gf}$ such that $F(a, g)(z) = x$ and $F(f, c)(z) = y$ and $(f, z) : (a, u) \longrightarrow (c, w)$ is the composite.*

Given $\oint_{\mathsf{J}} F$ and $\oint_{\mathsf{J}} G$, a morphism $\oint_{\mathsf{J}} F \longrightarrow \oint_{\mathsf{J}} G$ is a functor $\theta$ making the following commute:

$$\begin{tikzcd}
\oint_{\mathsf{J}} F \arrow[rr, "\theta"] \arrow[dr, "\pi_F"'] & & \oint_{\mathsf{J}} G \arrow[dl, "\pi_G"] \\
& \mathsf{J} &
\end{tikzcd}$$

The twisted elements construction can be made functorial such that, given $\eta : F \Longrightarrow G$, $\oint_{\mathsf{J}} \eta$ is the functor:

$$(\oint_{\mathsf{J}} \eta)(a, x) = \eta_a x$$
$$(\oint_{\mathsf{J}} \eta)(f, z) = \eta_f z$$

Next, let $\pi : (\oint F) \longrightarrow \mathsf{J}$ be the evident projection functor. This functor has the following property.

Lemma 4.5 (Lamarche) *If $r$ is a morphism in $\oint F$ such that $h = \pi r$, then each factorization of $h = gf$ in $\mathsf{J}$, uniquely determines a factorization of $r = ts$ in $\oint F$ where $g = \pi t$ and $f = \pi s$.*

Proof Given $(h, z) : (a, u) \longrightarrow (c, w)$ in $\oint F$ where $h = gf$ then $(f, F(a, g)(z)) : (a, u) \longrightarrow (b, F(f, g)(z))$ and $(g, F(f, c)(z)) : (b, F(f, g)(z)) \longrightarrow (c, w)$ are morphisms in $\oint F$ which compose to $(h, z)$. Since $F$ associates a $\circ$-pullback with $h = gf$, the factorization is unique. □

This makes $\pi : (\oint F) \longrightarrow \mathsf{J}$ a fibration in the sense of Conduché. See Johnstone [Joh77].

Finally, the categories $\int F$ and $\oint F$ are related as follows.

[2] A similar construction was introduced by Abramsky and Pavlović [AP97] for functors $\mathsf{J} \longrightarrow \mathsf{Sp(C)}$.

Lemma 4.6 *Given an algorithmic functor* $F : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$, *then*

$$\int F \cong \widetilde{(\oint F)}$$

Proof For objects, note that if $f : a \longrightarrow b$ is a morphism in $\mathsf{J}$, then:

$$\begin{aligned} (\widetilde{f}, u) \in \int F &\iff u \in F\widetilde{f} \wedge F(a,f)(u) \in Fa \wedge F(f,b)(u) \in Fb \\ &\iff (f,u) : (a, F(a,f)(u)) \longrightarrow (b, F(f,b)(u)) \in \oint F \\ &\iff \widetilde{(f,u)} \in \widetilde{(\oint F)} \end{aligned}$$

In the case of morphisms, if $(l,m) : (\widetilde{f}, x) \longrightarrow (\widetilde{g}, y)$ is a morphism in $\int F$, then $f = m \circ g \circ l$ and $F(l,m)(x) = y$. By lemma 4.5 there exists unique $u$, $v$ and $w$ such that $(l,u), (g,v)$ and $(m,w)$ are morphisms in $\oint F$, $(f,x) = (m,w) \circ (g,v) \circ (l,u)$ and $y = v$. This implies $\big((m,w),(l,u)\big) : \widetilde{(f,x)} \longrightarrow \widetilde{(g,y)}$ is a morphism in $\widetilde{(\oint F)}$. The uniqueness of $w$ and $u$ ensures the argument reverses. □

# 5 Twisted Systems

A transition system labelled in an alphabet $L$ is a diagram in $\mathsf{Grph}$ of the form:

$$1 \xrightarrow{\nabla} J \xrightarrow{f} G_L$$

The graph $J$ is the *shape* of the transition system. The vertices of $J$ are states and the edges are transitions. The graph $G_L$ has a single vertex and an edge for each label in $L$. The morphism $\nabla$ selects the start state.

*Categorical transition systems* (or simply *systems*) generalize this in two ways. First we abandon the requirement that shapes be graphs. A universe of shapes is a category $\mathsf{Shp}$ such that each shape, $J$, determines a category $\kappa(J)$, via a functor: $\kappa : \mathsf{Shp} \longrightarrow \mathsf{Cat}$. Second, a shape is labelled in a category rather than an alphabet or term algebra. Thus, a categorical transition system, $\mathsf{S}$, is a pair consisting of a shape and a functor:

$$(J, \kappa(J) \xrightarrow{S} \mathsf{C})$$

Abandoning graphs offers the opportunity of choosing shapes more suitable for modelling concurrent and asynchronous computation. For example, higher-dimensional automata [Gla91, Pra91, GJ92, Gou95] are a reasonable choice. Goubault's definition of hda is similar to that for simplicial sets. But, whereas simplicial sets deal with collections of unoriented $n$-triangles, an hda is a collection of oriented hypercubes. One way to obtain a category from an hda is by a construction similar to the fundamental groupoid for a topological space.

Labelling a shape in a category means that transitions can have more structure. Actions can be functions, machine instructions or even processes. See [Err] for further motivation and details concerning categorical transition systems and concurrency.

*Twisted systems* are a particular class of systems in which the domain of the functor is twisted.

$$(J, \widetilde{\kappa(J)} \xrightarrow{S} \mathsf{C})$$

They form a category $\widetilde{\mathsf{CTS}}(\mathsf{Shp}, \mathsf{C})$ where a morphism $\mathbf{f}$ is a pair

$$(f, \eta) : (J, \widetilde{\kappa(J)} \xrightarrow{S} \mathsf{C}) \longrightarrow (K, \widetilde{\kappa(K)} \xrightarrow{T} \mathsf{C})$$

such that $f : K \longrightarrow J$ is a morphism in $\mathsf{Shp}$ and $\eta : S \circ \widetilde{\kappa(f)} \Longrightarrow T$ is a natural transformation.

$$\begin{tikzcd}
J & \widetilde{\kappa(J)} \arrow[rrd, "S"] & & \\
 & & \Downarrow\eta & \mathsf{C} \\
K \arrow[uu, "f"] & \widetilde{\kappa(K)} \arrow[uu, "\widetilde{\kappa(f)}"] \arrow[rru, "T"'] & &
\end{tikzcd}$$

This category arises as an instance of the Grothendieck construction:

$$(\widetilde{\mathsf{CTS}}(\mathsf{Shp}, \mathsf{C}))^{op} = \int (\mathsf{Shp}^{op} \xrightarrow{\kappa^{op}} \mathsf{Cat}^{op} \xrightarrow{\widetilde{(-)}^{op}} \mathsf{Cat}^{op} \xrightarrow{(\mathsf{C}^{(-)})^{op}} \mathsf{CAT})$$

A consequence of this is that results due to Gray [Gra66] and Tarlecki *et al* [TBG91] concerning the existence of limits and colimits apply.

Note that the functor and natural transformation in a morphism are opposed. The "semi-dual" category in which the functor and natural transformation are in the same direction is easily defined. The latter is used for interpreting processes in [Err96, Err].

# 6 A bicategory of algorithms

The next objective is to construct systems like that for the factorial program in a compositional manner. Note that a system with a start and end state is an algorithm typed in $\mathsf{C}$. If $\nabla_S$ and $\triangle_S$ are the start and end states in $\mathbf{S} = (J, \widetilde{\kappa(J)} \xrightarrow{S} \mathsf{C})$, then $\mathbf{S}$ accepts input of type $S\nabla_S$ and produces output of type $S\triangle_S$. This suggests constructing a simple category with the same objects as $\mathsf{C}$ and where morphisms are algorithms. Composing algorithms sequentially means gluing one system on to the end of the other. This can be done using the general theory advocated by Goguen [Gog91]. We will use similar construction which is marginally simpler in this instance.

For the remainder of this paper we fix the target category to be $\mathsf{Set}$. With respect to shapes, let $\mathsf{FGrCat}$ be the Kleisli category for the monad $FU : \mathsf{Cat} \longrightarrow \mathsf{Cat}$ where $U : \mathsf{Cat} \longrightarrow \mathsf{Grph}$ and $F : \mathsf{Grph} \longrightarrow \mathsf{Cat}$ is the familiar adjunction. Objects are freely generated from graphs and morphisms are arbitrary functors. The category of shapes are bipointed free categories: $\mathsf{FGrCat}_{**} = 1 + 1 \downarrow \mathsf{FGrCat}$. A system $\mathbf{S}$ is a pair of the form:

$$(1 + 1 \xrightarrow{[\nabla_{\mathbf{S}}, \triangle_{\mathbf{S}}]} \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set})$$

Often we will write simply $(\mathsf{J}, S : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set})$ and only refer to the start and end states when necessary.

Now define the bicategory of algorithms, $\mathsf{Alg}$, as follows. Objects are sets. A morphism $A \longrightarrow B$ is a system $\mathbf{S}$ such that $S\nabla_{\mathbf{S}} = A$ and $S\triangle_{\mathbf{S}} = B$. Given a second system $\mathbf{T} = (1 + 1 \xrightarrow{[\nabla_T, \triangle_T]} \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{T} \mathsf{Set})$ such that $S\triangle_S = T\nabla_T$. Then $\mathbf{T} \circ \mathbf{S}$ is the system

$$(1 + 1 \longrightarrow \mathsf{J} +_1 \mathsf{K}, \widetilde{\mathsf{J} +_1 \mathsf{K}} \xrightarrow{Q} \mathsf{Set})$$

The shape of $\mathbf{T} \circ \mathbf{S}$ is the pushout on the left in diagram 6.1.

$$\begin{tikzcd}
1 \arrow[r, "\triangle_S"] \arrow[d, "\nabla_T"'] & \mathsf{J} \arrow[d, "q"] \\
\mathsf{K} \arrow[r, "p"] & \mathsf{J} +_1 \mathsf{K} \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

$$\begin{tikzcd}
1 \cong \widetilde{1} \arrow[rr, "\widetilde{\triangle}_S"] \arrow[dd, "\widetilde{\nabla}_T"'] & & \widetilde{\mathsf{J}} \arrow[dl, "\widetilde{p}"] \arrow[dd, "r"] \\
& \widetilde{\mathsf{J} +_1 \mathsf{K}} & \\
\widetilde{\mathsf{K}} \arrow[ur, "\widetilde{q}"] \arrow[rr, "s"'] & & \widetilde{\mathsf{J}} +_1 \widetilde{\mathsf{K}} \arrow[ul, dotted] \arrow[ulul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

Diagram 6.1.

To define the functor $Q$, first twist the pushout on the left and form a second pushout as shown on the right. To illustrate the difference between $\widetilde{\mathsf{J} +_1 \mathsf{K}}$ and $\widetilde{\mathsf{J}} +_1 \widetilde{\mathsf{K}}$, consider the example in diagram 6.2 where $\mathsf{J} = \mathsf{K} = 2$. The left side shows the effect of taking the pushout followed by twisting. On the right, the shapes are twisted first followed by the pushout. Constructing the colimit first may introduce new composable pairs of morphisms which must be added. This is the case on the left with the new morphism is labelled $h$. When the twist comes first, there are no morphisms to be added by colimits identifying $\mathsf{J}$ objects. Let $E : \widetilde{\mathsf{J}} +_1 \widetilde{\mathsf{K}} \rightarrowtail \widetilde{\mathsf{J} +_1 \mathsf{K}}$ be the universal arrow which is necessarily an injection.

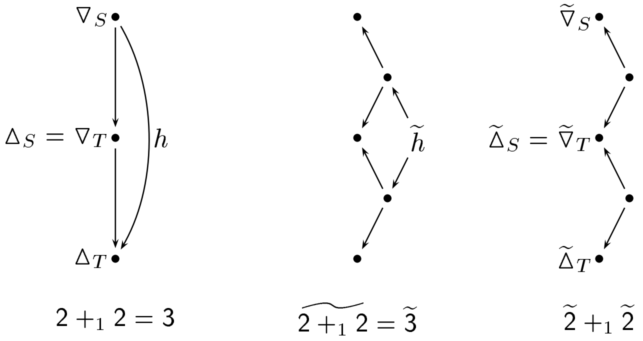


Diagram 6.2.

Since $S\Delta_S = T\nabla_T$, there is a universal morphism $[S,T]_1 : \widetilde{\mathsf{J}} +_1 \widetilde{\mathsf{K}} \longrightarrow \mathsf{Set}$. To define $Q$ requires extending $[S,T]_1$ to the objects and morphisms in $\widetilde{\mathsf{J} +_1 \mathsf{K}}$ which do no appear in either $\widetilde{\mathsf{J}}$ or $\widetilde{\mathsf{K}}$. We take $Q = \mathrm{Ran}_E[S,T]_1$ (see diagram 6.3). In the example, $Q\widetilde{h}$ is the apex of the zig-zag diagram in $\mathsf{Set}$ determined by $f$ and $g$. This implies that if $\mathbf{S}$ and $\mathbf{T}$ are algorithmic systems, then $\mathbf{T} \circ \mathbf{S}$ is algorithmic.

$$\begin{tikzcd}
\widetilde{\mathsf{J} +_1 \mathsf{K}} \arrow[rd, "Q"] & \\
\widetilde{\mathsf{J}} +_1 \widetilde{\mathsf{K}} \arrow[u, tail, "E"] \arrow[r, "{[S,T]_1}"'] & \mathsf{Set}
\end{tikzcd}$$

Diagram 6.3.

# 7 An imperative language

This section introduces a version of Dijkstra's guarded command language [Dij75]. The next section gives an interpretation in $\mathsf{Alg}$.

The basic types and expressions in the language are provided by an algebraic theory $\mathsf{Th}(\Sigma)$. This will be left unspecified but it is assumed to contain types and standard constants for the natural numbers and booleans. The collection of well-formed terms in the algebra are generated in the usual way by rules containing judgments or *terms in context* of the form $t{:}X\ [\Gamma]$. Here $t$ is a term, $X$ a sort and $\Gamma = [x_1{:}X_1, \ldots, x_n{:}X_n]$ is a *context* containing a list of typed variables including the free variables in $t$. As is standard, the algebra comes with an equational theory and determines a syntactic category (also called $\mathsf{Th}(\Sigma)$). It is assumed there is semantic functor $[\![-]\!] : \mathsf{Th}(\Sigma) \longrightarrow \mathsf{Set}$. For example,

$$[\![t{:}X\ [\Gamma]]\!] = [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![X]\!]$$

where $[\![\Gamma]\!] = [\![X_1]\!] \times \cdots \times [\![X_n]\!]$.

The syntax of the programming language is defined on top of the algebra. The grammar is as follows:

$$\begin{aligned} gc \quad &::= \quad \mathsf{nil} \mid x := t \mid \mathsf{var}\ x{:}X\ \mathsf{in}\ gc \\ &\quad\;\;\;\; \mid gc\,;gc \mid \mathsf{if}\ alt \mid \mathsf{do}\ alt \\ alt \quad &::= \quad b \rightarrow gc \,\Box \ldots \Box\, b \rightarrow gc \end{aligned}$$

Here, $b$ is a predicate on the state, $x$ is a typed variable and $t$ is a term all of which come from the underlying algebra. Often we will write $\Box_{i \le n}(b_i \rightarrow gc_i)$ in place of $b_1 \rightarrow gc_1 \,\Box \cdots \Box\, b_n \rightarrow gc_n$.

The rules for well-formed programs are given in diagram 7.1. A judgment $gc\ [\Gamma]$ means that the program $gc$ is a well-formed program in the context, $\Gamma$. A context in a judgment $gc\ [\Gamma]$ serves two purposes. Like in the algebra, it assigns types to the variables free in $gc$. Contexts also keep track of the variables currently in scope. Note that the variables in scope before and after a command are the same and a judgment $gc\ [\Gamma]$ implies a type assignment $gc : \Gamma \longrightarrow \Gamma$.

The informal meaning of conditionals and repetitions are as follows. Given a variable assignment $s$, the $\mathsf{if}$ is evaluated by choosing non-deterministically one of the guarded commands from those whose guard is satisfied by $s$. If none of the guards are satisfied the program deadlocks. In the case of $\mathsf{do}$, the selection of a guarded command is repeated until no guard is satisfied whereupon the loop terminates successfully.

$$\frac{}{\mathsf{nil}\ [\Gamma]} \qquad \frac{t\!:\!X\ [\Gamma, x\!:\!X]}{x := t\ [\Gamma, x\!:\!X]} \qquad \frac{gc_1\ [\Gamma] \qquad gc_2\ [\Gamma]}{gc_1\ ;\ gc_2\ [\Gamma]}$$

$$\frac{\forall i \leq n \qquad b_i\!:\!\mathsf{bool}\ [\Gamma] \qquad gc_i\ [\Gamma]}{\mathsf{if}\ \square_{i \leq n}(b_i \to gc_i)\ [\Gamma]} \qquad \frac{\forall i \leq n \qquad b_i\!:\!\mathsf{bool}\ [\Gamma] \qquad gc_i\ [\Gamma]}{\mathsf{do}\ \square_{i \leq n}(b_i \to gc_i)\ [\Gamma]}$$

$$\frac{gc\ [\Gamma, x\!:\!X]}{\mathsf{var}\ x\!:\!X\ \mathsf{in}\ gc\ [\Gamma]}\ x \notin \Gamma \qquad \frac{gc\ [\Gamma]}{gc\ [\Gamma, x\!:\!X]}\ x \notin \Gamma$$

Diagram 7.1.

The structural operational semantics for the language is given in diagram 7.2. Note that the evaluation of guards is done by appealing to the algebraic theory rather than assuming an interpretation. Also note that the syntax does not require new variables to be initialized. Instead, the semantics branches for every possible assignment of a new variable.

# 8 Categorical operational semantics

A categorical semantics of the language is defined inductively as follows.

**Empty program** The system for the program nil $[\Gamma]$ is:

$$(1 + 1 \longrightarrow 1, 1 \xrightarrow{[\![\Gamma]\!]} \mathsf{Set})$$

**Sequential composition** This is categorical composition in Alg.

**Assignment** Assume $\Gamma = [\Delta, x\!:\!X, \Phi]$ and let 2 be the graph $0 \longrightarrow 1$. The meaning of $x := t\ [\Gamma]$ is the system $\mathbf{S} = (1 + 1 \xrightarrow{[0,1]} 2, \widetilde{2} \xrightarrow{S} \mathsf{Set})$ where $S$ labels the transition with the span $[\![\Gamma]\!] \xleftarrow{id} [\![\Gamma]\!] \xrightarrow{\langle \pi_\Delta, [\![t]\!], \pi_\Phi \rangle} [\![\Gamma]\!]$.

**Local variables** Let $\mathbf{S}$ be the interpretation of $gc$ in var $x : X$ in $gc\ [\Gamma]$. To introduce a new variable, $\mathbf{S}$ is prefixed by a simple program $\mathbf{A}$ in which the variable is allocated and suffixed by the reciprocal program, $\mathbf{A}^\circ$, where the variable is discarded. The program which allocates the variable is $\mathbf{A} =$

$$\frac{\langle gc, s\rangle \longrightarrow \langle gc', s'\rangle}{\langle \mathsf{nil}\,;\, gc, s\rangle \longrightarrow \langle gc', s'\rangle} \qquad \frac{}{\langle x := t, s\rangle \longrightarrow \langle \mathsf{nil}, s[x \mapsto (s\ t)]\rangle}$$

$$\frac{\langle gc_1, s\rangle \longrightarrow \langle gc_1', s'\rangle}{\langle gc_1\,;\, gc_2, s\rangle \longrightarrow \langle gc_1'\,;\, gc_2, s'\rangle}$$

$$\frac{\vdash_A (s\ b_i) = \mathsf{true}}{\langle \mathsf{if}\ \Box_{i\leq n}(b_i \to gc_i), s\rangle \longrightarrow \langle gc_i, s\rangle}$$

$$\frac{\vdash_A (s\ b_i) = \mathsf{true}}{\langle \mathsf{do}\ \Box_{i\leq n}(b_i \to gc_i), s\rangle \longrightarrow \langle gc_i\,;\, \mathsf{do}\ \Box_{i\leq n}(b_i \to gc_i), s\rangle}$$

$$\frac{\vdash_A \bigwedge_{i\leq n}(s\ b_i) = \mathsf{false}}{\langle \mathsf{do}\ \Box_{i\leq n}(b_i \to gc_i), s\rangle \longrightarrow \langle \mathsf{nil}, s\rangle}$$

$$\frac{\vdash_A t : X}{\langle \mathsf{var}\ x : X\ \mathsf{in}\ gc, s\rangle \longrightarrow \langle gc, s[x \mapsto t]\rangle}$$

Diagram 7.2.

$(1+1 \xrightarrow{[0,1]} 2, \widetilde{2} \xrightarrow{Q} \mathsf{Set})$ with a single transition labelled by $Q$ with the span $[\![\Gamma]\!] \xleftarrow{\pi} [\![\Gamma, x : X]\!] \xrightarrow{id} [\![\Gamma, x : X]\!]$. The meaning of the program in context is $\mathbf{A}^\circ \circ \mathbf{S} \circ \mathbf{A}$. Note that variables are universally quantified (uninitialized) when introduced.

**Alternation** Before looking at conditionals and loops we consider alternation. Let $\mathbf{S}$ be the interpretation of $gc$ in the single guarded command $b \to gc\ [\Gamma]$. A predicate $b : \mathsf{bool}\ [\Gamma]$ determines a subobject of $[\![\Gamma]\!]$ by pulling back against the classifying arrow in $\mathsf{Set}$:

$$\begin{tikzcd}
X \arrow[r, tail, "s"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket \Gamma \rrbracket \arrow[d, "\llbracket b \rrbracket"] \\
1 \arrow[r, tail, "\mathsf{true}"'] & 2 = \llbracket \mathsf{bool} \rrbracket
\end{tikzcd}$$

Construct the system $\mathbf{B}$ with a single transition labelled by the span

$$\llbracket \Gamma \rrbracket \xleftarrow{s} X \xrightarrow{s} \llbracket \Gamma \rrbracket$$

This is a transition which can be taken only for tuples satisfying the predicate $\llbracket b \rrbracket$. The meaning of the single guarded command is $\mathbf{S} \circ \mathbf{B}$.

Now consider the family of guarded commands $(b_i \to gc_i)_{i \leq n}$ $[\Gamma]$ where the meaning of each member is given by $\mathbf{S}_i = (\mathsf{1}+\mathsf{1} \xrightarrow{[\nabla_i, \Delta_i]} \mathsf{J}_i, \widetilde{\mathsf{J}}_i \xrightarrow{S_i} \mathsf{Set})$. The meaning of the family is obtained by identifying respectively the elements of the sets $(\nabla_i)_{i \leq n}$ and $(\Delta_i)_{i \leq n}$. This yields a system $(\mathsf{1}+\mathsf{1} \xrightarrow{[\nabla, \Delta]} \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set})$ where the shape $\mathsf{J}$ is the colimit on the left of diagram 8.1. The functor $S : \widetilde{\mathsf{J}} \longrightarrow \mathsf{C}$ is obtained by exchanging the order of the colimit and the twist as in sequential composition. In the diagram on the right, $\mathsf{K}$ is the colimit following the twist and $D$ is the induced universal arrow. Define $S = \mathrm{Ran}_D[S_1, \ldots, S_n]$.

$$\begin{tikzcd}
 & \mathsf{J}_1 \arrow[dd, dotted, no head] \arrow[dr] & \\
\mathsf{1}+\mathsf{1} \arrow[ur, "{[\nabla_1, \Delta_1]}"] \arrow[dr, "{[\nabla_n, \Delta_n]}"'] & & \mathsf{J} \\
 & \mathsf{J}_n \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \widetilde{\mathsf{J}}_1 \arrow[dd, dotted, no head] \arrow[dr] \arrow[drr] & & \\
\mathsf{1}+\mathsf{1} \arrow[ur, "\widetilde{[\nabla_1, \Delta_1]}"] \arrow[dr, "\widetilde{[\nabla_n, \Delta_n]}"'] & & \widetilde{\mathsf{J}} \arrow[ddr, "{\mathrm{Ran}_D[S_1, \ldots, S_n]}"'] & \mathsf{K} \arrow[l, dotted, "D"'] \arrow[dd, "{[S_1, \ldots, S_n]}"] \\
 & \widetilde{\mathsf{J}}_n \arrow[ur] \arrow[urr] & & \\
 & & & \mathsf{Set}
\end{tikzcd}$$

Diagram 8.1.

**Conditional** The meaning of a conditional $\mathsf{if}$ $\square_{i \leq n}(b_i \to gc_i)$ $[\Gamma]$ is the process for the alternation as defined above.

**Loops** Let $\mathbf{A}$ be the interpretation of the alternation in $\mathsf{do}\ \square_{i\leq n}(b_i \to gc_i)\ [\Gamma]$. Recall that the intended semantics of $\mathsf{do}$ is that the loop exits when none of the guards are satisfiable. Define *exit* to be the program which exits when all the other guards fail:

$$exit = \neg(\bigwedge_{i\leq n} b_i) \to \mathsf{nil}\ [\Gamma]$$

Using the scheme outlined above, this is interpreted by the system $\mathbf{E}$ having a single transition which can be taken precisely when the other guards fail.

Take $\mathbf{A}$ and $\mathbf{E}$ to be the systems:

$$\mathbf{A} = (\mathsf{1}+\mathsf{1} \xrightarrow{[\nabla_a,\Delta_a]} \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{A} \mathsf{Set}) \qquad\qquad \mathbf{E} = (\mathsf{1}+\mathsf{1} \xrightarrow{[\nabla_e,\Delta_e]} \mathsf{K}, \widetilde{\mathsf{K}} \xrightarrow{E} \mathsf{Set})$$

To interpret the $\mathsf{do}$ loop requires combining $\mathbf{A}$ and $\mathbf{E}$ in such a way that the loop is closed but can exit when the alternation fails. This is achieved by identifying three points: the start and end points of the alternation (closing the loop) and the start point of $\mathbf{E}$. This yields a system $(\mathsf{1}+\mathsf{1} \xrightarrow{[\nabla,\Delta]} \mathsf{L}, \widetilde{\mathsf{L}} \xrightarrow{S} \mathsf{Set})$. The functor $S$ is computed by the now familiar exchange of colimits and twists. The shape $\mathsf{L}$ is the colimit on the left in diagram 8.2, $\mathsf{M}$ is the colimit following the twist, and $D$ is universal. Define $S = \mathrm{Ran}_D[A,E]$.

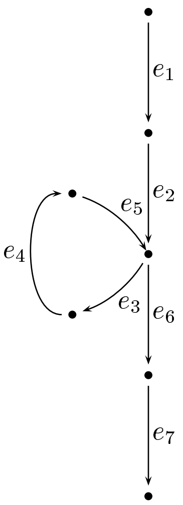
$$\begin{tikzcd}
\mathsf{1} \arrow[r, shift left, "\nabla_a"] \arrow[r, shift right, "\Delta_a"'] \arrow[dr, "\nabla_e"'] & \mathsf{J} \arrow[r, "p"] & \mathsf{L} \\
& \mathsf{K} \arrow[ur, "q"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\widetilde{\mathsf{1}} \arrow[r, shift left, "\widetilde{\nabla}_a"] \arrow[r, shift right, "\widetilde{\Delta}_a"'] \arrow[dr, "\widetilde{\nabla}_e"'] & \widetilde{\mathsf{J}} \arrow[r, "\widetilde{p}"] \arrow[dr] & \widetilde{\mathsf{L}} \arrow[dr, "S = \mathrm{Ran}_D[A,E]"] & \\
& \widetilde{\mathsf{K}} \arrow[ur, "\widetilde{q}"'] \arrow[r] & \mathsf{M} \arrow[u, dotted, "D"] \arrow[r, "{[A,E]}"'] & \mathsf{Set}
\end{tikzcd}$$

Diagram 8.2.

Figure 8.3 shows the factorial program, the graph shape generated by the scheme outlined above and the assignment of types and spans to the vertices and edges of the graph.

Lemma 8.1 *The semantics of a program by the scheme outlined above is an algorithmic system.*

Proof This is a straightforward structural induction. The shapes generated by the semantics are graphs. Since arrows freely added when joining shapes with colimits are labelled by means of a right Kan extension, then $\circ$-limits in a twisted shape are always mapped to limits in $\mathsf{Set}$. □



```
$x := 1$                 $f_1$
$y := 1$                 $f_2$
do
    $x < n \rightarrow$            $f_3$
        $x := x + 1$     $f_4$
        $y := x \times y$     $f_5$
$n := x$                 $f_7$  $[n : \mathbb{N}, x : \mathbb{N}, y : \mathbb{N}]$
```

| | |
|---|---|
| $e_1$ | $N^3 \xleftarrow{id} N^3 \xrightarrow{\lambda(n,x,y).(n,1,y)} N^3$ |
| $e_2$ | $N^3 \xleftarrow{id} N^3 \xrightarrow{\lambda(x,x,y).(n,x,1)} N^3$ |
| $e_3$ | $N^3 \xleftarrow{\lambda(n,x,y).(n,x,y)} X \xrightarrow{\lambda(n,x,y).(n,x,y)} N^3$ |
| $e_4$ | $N^3 \xleftarrow{id} N^3 \xrightarrow{\lambda(n,x,y).(n,x+1,y)} N^3$ |
| $e_5$ | $N^3 \xleftarrow{id} N^3 \xrightarrow{\lambda(n,x,y).(n,x,x \times y)} N^3$ |
| $e_6$ | $N^3 \xleftarrow{\lambda(n,x,y).(n,x,y)} Y \xrightarrow{\lambda(n,x,y).(n,x,y)} N^3$ |
| $e_7$ | $N^3 \xleftarrow{id} N^3 \xrightarrow{\lambda(n,x,y).(x,x,y)} N^3$ |
| $X$ | $\{(n, x, y) \in N^3 \mid x < n\}$ |
| $Y$ | $\{(n, x, y) \in N^3 \mid x \geq n\}$ |

Diagram 8.3.

# 9 Procedures

A notable omission from the language are user defined procedures and functions. We have relied solely on the underlying algebraic theory for operations and expressions. Procedures can be accommodated in the framework, however, it is difficult to give a compositional semantics. Below we sketch a scheme for handling procedures.

First the syntax of the language is extended such that now, in addition to local variable declarations, there are Pascal-like procedure/function declarations:

```
proc $p$ $(x_1{:}X_1, \ldots, x_n{:}X_n){:}Y$ where
    $gc_1$
    return $t$
in
    gc$_2$
```

This creates a new scope and declares a procedure $p$ of type $X_1 \times \cdots \times X_n \longrightarrow Y$. The body of the procedure is $gc_1$ and the scope of the procedure is $gc_2$. The identifiers $x_1, \ldots, x_n$ are the formal parameters. Any variable in scope when the procedure is declared is in scope within the procedure. Procedures may have side-effects and parameters are passed by value. Define the context $\Phi = [\Gamma, x_1{:}X_1, \ldots, x_n{:}X_n]$. Then a well-formed declaration must satisfy the following rule:

$$\frac{gc_1\ [\Phi] \qquad t{:}Y\ [\Phi] \qquad gc_2\ [\Gamma, p{:}X_1 \times \cdots \times X_n \longrightarrow Y]}{\mathsf{proc}\ p\ (x_1{:}X_1, \ldots, x_n{:}X_n){:}Y\ \mathsf{where}\ gc_1\ \mathsf{return}\ t\ \mathsf{in}\ \mathsf{gc}_2\ [\Gamma]}$$

We further extend the syntax with procedure calls in assignments:

$$x := p(t_1, \ldots, t_n)$$

It follows that procedure calls may not be nested within terms of the underlying algebra. Calls must satisfy the following rule:

$$\frac{t_i{:}X_i\ [\Gamma, x{:}Y, p{:}X_1 \times \cdots \times X_n \longrightarrow Y]}{x := p(t_1, \ldots, t_n)\ [\Gamma, x{:}Y, p{:}X_1 \times \cdots \times X_n \longrightarrow Y]}$$

One way to define the semantics of procedures is to construct a system for the procedure and then substitute a copy of that system at every call. This, however, makes recursion difficult. An alternative is to handle procedures as they are handled by a compiler. This has the advantage that only one copy of the system is needed. We first construct the system for the body of the procedure. Intuitively the imperative state of this system is then augmented with a "variable" $r : R$. The set $R$ has an element for each reference to the procedure in the program. It represents the set of *return addresses*. A call to the procedure is a transition to the procedure which sets the return address according to the location where the call is made. A call from the $i$'th reference sets the variable $r$ to $i$. The return to the $i$'th reference is a transition back to the program which is taken only when $r = i$.

The two methods for defining the semantics of procedures are related. Assume again that there are $R$ references to the procedure. Then following the first scheme, there are $R$ copies of the procedure. If we form the coproduct of the $R$ copies, we obtain a system isomorphic to the system described in the second scheme in which the imperative state is augmented by $R$. This follows since $\coprod_R X \cong R \times X$ in Set . Setting the return address to $i$ corresponds to the $i$'th injection into the coproduct.

We illustrate the construction with an example. Diagram 9.1 shows the system for a simple program defined in a context $\Gamma = [x : \mathbb{N}, y : \mathbb{N}]$. The set $2 = \{0, 1\}$.

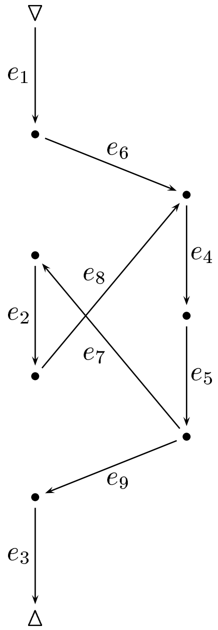
To simplify the example we have assumed that no variables are introduced between the declaration of $p$ and the references to $p$. This can happen as follows:

```
proc $p$ $(x_1 : X_1, \ldots , x_n : X_n) : Y$ where
    $gc_1$;
    return $t$
in
    var $x : Y$ in
        $\ldots$
        $x := p(t_1, \ldots , t_n)$
```

In such situations it is necessary to *weaken* the context in which $p$ is declared to incorporate variables introduced between the declaration and the call. If $p$ is referenced from different contexts then it must be weakened to the most general context.

The same construction applies to recursive procedures. If we consider the unfolding of a recursive procedure, then with each unfolding new variables corresponding to the formal parameters are introduced. Following the argument above, the context of the procedure must be weakened to include the variables for each unfolding. This naturally introduces a stack discipline.

Diagram 9.2 shows a system for a recursive factorial program. We write $\vec{x}$ to denote an infinite vector of variables $(x_0, x_1, \ldots , x_i, \ldots)$ and $\vec{x}\,|_i\, t$ for the vector $(x_0, \ldots , x_{i-1}, t, x_{i+1}, \ldots)$ which is the same as $\vec{x}$ except with the term



```
proc $p$ $(u{:}\mathbb{N}, v{:}\mathbb{N}){:}\mathbb{N}$ where
    $u := u + v$            $e_4$
    return $2u$             $e_5$
in
    $x := 1$                $e_1$
    $x := p(x, 3)$          $e_6, e_8$
    $y := x$                $e_2$
    $y := p(x, y)$          $e_7, e_9$
    $y := y^2$              $e_3$
```

$$
\begin{array}{l|l}
e_1 & N^2 \xleftarrow{id} N^2 \xrightarrow{\lambda(x,y).(1,y)} N^2 \\
e_2 & N^2 \xleftarrow{id} N^2 \xrightarrow{\lambda(x,y).(x,x)} N^2 \\
e_3 & N^2 \xleftarrow{id} N^2 \xrightarrow{\lambda(x,y).(x,y^2)} N^2 \\
e_4 & 2 \times N^4 \xleftarrow{id} 2 \times N^4 \xrightarrow{\lambda(r,x,y,u,v).(r,x,y,u+v,v)} 2 \times N^4 \\
e_5 & 2 \times N^4 \xleftarrow{id} 2 \times N^4 \xrightarrow{\lambda(r,x,y,u,v).(r,x,y,2u)} 2 \times N^3 \\
e_6 & N^2 \xleftarrow{id} N^2 \xrightarrow{\lambda(x,y).(0,x,y,x,3)} 2 \times N^4 \\
e_7 & 2 \times N^3 \xleftarrow{\lambda(r,x,y,u).(0,x,y,u)} N^3 \xrightarrow{\lambda(x,y,u).(u,y)} N^2 \\
e_8 & N^2 \xleftarrow{id} N^2 \xrightarrow{\lambda(x,y).(1,x,y,x,y)} 2 \times N^4 \\
e_9 & 2 \times N^3 \xleftarrow{\lambda(x,y,u).(1,x,y,u)} N^3 \xrightarrow{\lambda(x,y,u).(x,u)} N^2
\end{array}
$$

Diagram 9.1.

$t$ at position $i$. The stack has been simplified with respect to the scheme discussed above. In a pair $(i, \vec{x})$, $\vec{x}$ is the stack and $i$ is the stack pointer.

# 10 The relationship with conventional semantics

For Burstall, the denotation of a program is the input/output relation it computes. For a program $(G, F(G) \xrightarrow{S} \mathsf{Rel})$, this is the union of the relations $\{Sf \mid f \in (F(G))(\nabla, \vartriangle)\}$. A similar construction can be made here. Let $\mathsf{S} = (1+1 \xrightarrow{[\nabla,\vartriangle]} \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set})$ be the meaning of a program obtained from the scheme outline earlier. Then the "denotation" of the program is the sum of the spans:

$$\{S\nabla \xleftarrow{S(\nabla,f)} S\widetilde{f} \xrightarrow{S(f,\vartriangle)} S\vartriangle \mid f \in \mathsf{J}(\nabla, \vartriangle)\}$$

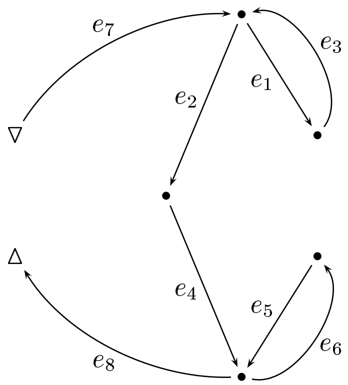
The legs of the sum span are universal. See diagram 10.1.

Somewhat more interesting is that the categorical semantics for the guarded command language is closely related to a conventional structural operational semantics. Consider the category of twisted elements $\oint S$. An object is a pair consisting a program point and a variable assignment at that point. Morphisms are transitions. It is easy to show that for every transition in the operational semantics there is a corresponding transition in $\oint S$. The converse does not hold. In part this is because there are identity and composite transitions in $\oint S$. This suggests comparing $\oint S$ with the reflexive transitive closure of the transition relation. Alternatively, note that since $\mathsf{J}$ is a free category, then by lemma 4.5, $\oint S$ is also a free category. Therefore we can compare the transition relation with the graph generating $\oint S$. However, there are still more states and transitions in the categorical semantics than in the transition relation. This arises because, for example, in the categorical semantics evaluating a guard involves a transition which is absent from the transition relation.

# 11 Systems as predicates

The systems used to give meaning to programs are algorithmic. Also the span associated with each transition is a partial map; for each $f : a \longrightarrow b$ in $\mathsf{J}$, the morphism $S(a, f)$ is monic. For the remainder of the paper we investigate systems as specifications of program properties. This makes use of the generality of spans (relative to partial maps) and also the underlying

```
proc $fact$ $(x : \mathbb{N}) : \mathbb{N}$ where
    if
        $x > 0 \rightarrow$                          $e_1$
            var $y : \mathbb{N}$ in
                $y := fact(x-1)$                 $e_3, e_6$
                $x := x \times y$                $e_5$
        $x = 0 \rightarrow$                          $e_2$
            $x := 1$                             $e_4$
    return $x$
in
    z $:= fact(3)$                               $e_7, e_8$
```



$$\begin{array}{r|l}
e_1 & N \times N^\omega \xleftarrow{\lambda(i,\vec{x}).(i,\vec{x})} W \xrightarrow{\lambda(i,\vec{x}).(i,\vec{x})} N \times N^\omega \\
e_2 & N \times N^\omega \xleftarrow{\lambda(i,\vec{x}).(i,\vec{x})} X \xrightarrow{\lambda(i,\vec{x}).(i,\vec{x})} N \times N^\omega \\
e_3 & N \times N^\omega \xleftarrow{id} N \times N^\omega \xrightarrow{\lambda(i,\vec{x}).(i+1,\vec{x}\,|_{i+1}\,(x_i-1))} N \times N^\omega \\
e_4 & N \times N^\omega \xleftarrow{\lambda(i,\vec{x}).(i,\vec{x})} Y \xrightarrow{\lambda(i,\vec{x}).(i,\vec{x}\,|_i\,1)} N \times N^\omega \\
e_5 & N \times N \times N^\omega \xleftarrow{id} N \times N \times N^\omega \xrightarrow{\lambda(i,y,\vec{x}).(i,\vec{x}\,|_i\,x_i \times y)} N \times N^\omega \\
e_6 & N \times N^\omega \xleftarrow{id} N \times N^\omega \xrightarrow{\lambda(i,\vec{x}).(i-1,x_i,\vec{x})} N \times N \times N^\omega \\
e_7 & 1 \xleftarrow{!} N \times N^\omega \xrightarrow{\lambda(i,\vec{x}),(0,\vec{x}\,|_0\,3)} N \times N^\omega \\
e_8 & N \times N^\omega \xleftarrow{\lambda(i,\vec{x}).(i,\vec{x})} Z \xrightarrow{\lambda(i,\vec{x}).x_0} N \\
W & \{(i,\vec{x}) \in N \times N^\omega \mid x_i > 0\} \\
X & \{(i,\vec{x}) \in N \times N^\omega \mid x_i = 0\} \\
Y & \{(i,\vec{x}) \in N \times N^\omega \mid i > 0\} \\
Z & \{(i,\vec{x}) \in N \times N^\omega \mid i = 0\}
\end{array}$$

Diagram 9.2.

$$\begin{tikzcd}
& S\widetilde{f}_1 \arrow[dl, "S(\nabla, f_1)"'] \arrow[r] \arrow[dd, dotted, no head] \arrow[drr] & \coprod S\widetilde{f}_i \arrow[dll] \arrow[dr] & \\
S\nabla & & & S\Delta \\
& S\widetilde{f}_n \arrow[ul, "S(\nabla, f_n)"] \arrow[uur] \arrow[urr, "S(f_n, \Delta)"'] & &
\end{tikzcd}$$

Diagram 10.1.

oplaxness arising from twisting shapes. To motivate the generality we begin by considering spans in isolation.

First, spans provide a simple and symmetric way of dealing with predicate transformers. When a program fragment is represented by a span, then the precondition for the fragment is the image factorization of the backward leg.

$$\begin{tikzcd}
& & X \arrow[dl, two heads] \arrow[ddrr] & & \\
& A' \arrow[dl, tail] & & & \\
A & & & & B
\end{tikzcd}$$

Similarly the post condition is the image factorization of the forward leg. Given a predicate $\theta \rightarrowtail B$ on the codomain, the *weakest precondition* is the precondition for the span $A \longleftarrow X' \longrightarrow B$ formed by the pullback:

$$\begin{tikzcd}
& X \arrow[dl] \arrow[dr] & \\
A & X' \arrow[u, tail] \arrow[dr] \arrow[dr, phantom, "\lrcorner", very near start] & B \\
& & \theta \arrow[u, tail]
\end{tikzcd}$$

The dual construction gives the *strongest postcondition.* See also [Ple96, Vic95].

Spans are generalized relations and, as with relations, 2-cells provide a way to relate programs and properties. Consider the program

$$x := x \times 2; y := 4 \ [x : \mathbb{N}, y : \mathbb{N}]$$

and the postcondition "$x$ is even" where no constraints are placed on $y$. In the diagram below the program is represented by the upper span and the property by the lower span where $f = \lambda(x, y).(2x, 4)$ and $g = \lambda((x, y), (x', y')).(2x', y')$.

$$\begin{tikzcd}
& N \times N \arrow[dl, "id"'] \arrow[dr, "f"] \arrow[dd, tail] & \\
N \times N & & N \times N \\
& (N \times N)^2 \arrow[ul, "\pi_1"] \arrow[ur, "g"'] &
\end{tikzcd}$$

The program has the property when there is a 2-cell between the spans which is an inclusion. When it exists, then standard results for image factorizations ensure that there is also a monic from the image (postcondition) of $f$ to the image of $g$. The same method can express when a program satisfies an input/output relation. An advantage of spans over relations is that properties can be expressed in terms of variables which are not in scope at the beginning and end of the block.

These principles extend to oplax functors $\mathsf{J} \multimap \mathsf{Sp}(\mathsf{Set})$ related by oplax map natural transformations. The 2-cells in an oplax map natural transformation serve the same purpose as the 2-cells above. An oplax functor associates a span with each transition. Each span asserts pre- or postconditions of a transition and relates the values of variables between program points. The oplaxness means it is possible to associate a stronger property with the transition $gf$ than is associated with $f$ and $g$ alone. More generally, it is possible to express that a computation path must compute some input/output relation without imposing constraints on *how* that relation is realized.

In practice we work with twisted systems rather than oplax functor categories. To relate programs and properties we shall follow the example of categorical models of first-order logic which we review below.[3]

Briefly, if $\Gamma = [x_1 : X_1, \ldots, x_n : X_n]$ is a context, then the meaning of a formula in context $\theta\ [\Gamma]$ in a category $\mathsf{C}$ is a subobject of the product $[\![\Gamma]\!] = [\![X_1]\!] \times \cdots \times [\![X_n]\!]$. Write $\mathit{Sub}(X)$ for the lattice of subobjects of $X$. It is a Heyting algebra where the ordering corresponds to satisfaction such that if $\theta \vdash \phi\ [\Gamma]$ then $[\![\theta\ [\Gamma]]\!]$ factors through $[\![\phi\ [\Gamma]]\!]$ in the fibre over $[\![\Gamma]\!]$. Conjunction is meet and disjunction is join in the fibre. To translate properties between fibres, let $f : X \longrightarrow Y$ be a morphism in $\mathsf{C}$ and write $f^* : \mathit{Sub}(Y) \longrightarrow \mathit{Sub}(X)$ for the functor which maps a subobject of $Y$ to its pullback against $f$.

Weakening of contexts is modelled by pulling back against projection maps:

$$[\![\theta\ [\Gamma, x{:}X]]\!] = \pi_1^*([\![\theta\ [\Gamma]]\!])$$

where $\pi_1 : [\![\Gamma, x : X]\!] \longrightarrow [\![\Gamma]\!]$. Given such a projection map, then universal quantification is the right adjoint to the pullback functor $\pi_1^*$. Existential quantification is the left adjoint. These functors satisfy the Beck-Chevally condition and Frobenius reciprocity.

[3] The handbook chapter by Pitts [Pit] is a good introduction to categorical models of predicate logic and a source for other references. Our notation has been adapted from Pitts.

The same constructions apply to programs and properties. We will not, however, interpret a logic. For the remainder of this section it is convenient to forget the start and end states. Given a system $\mathbf{B} = (\mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{B} \mathsf{Set})$, a *subsystem* has the same shape as $\mathbf{B}$ and a functor which is a subfunctor of $B$. As above, properties are objects in $\mathit{Sub}(\mathbf{B})$. A program $\mathbf{S}$ has a safety property $\mathbf{P}$ when it factors through $\mathbf{P}$ in the fibre over $\mathbf{B}$. As the shape is fixed, all this happens in the topos $\mathsf{Set}^{\widetilde{\mathsf{J}}}$.

This leaves the issue of choosing a base for the fibre in which to relate a program and the properties of interest. As a rule, a program will inhabit many fibres. However, each program has a *standard base* obtained by ignoring the actions of the algorithm and considering only the declaration and scoping of variables. The standard base specifies the shape, the type and scoping of variables and assigns a span to each transition which relates program variables in the most general way. For a program $\mathbf{S} = (\mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{C})$, the standard base is the system $\mathbf{B} = (\mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{B} \mathsf{C})$ where the functor $B$ is the right Kan extension shown in diagram 11.1.

$$\begin{tikzcd}
\widetilde{\mathsf{J}} \arrow[rrd, "B = \mathrm{Ran}_E(S \circ E)"] & & \\
\widetilde{|\mathsf{J}|} \arrow[u, tail, "E"] \arrow[r, tail, "E"'] & \widetilde{\mathsf{J}} \arrow[r, "S"'] & \mathsf{C}
\end{tikzcd}$$

Diagram 11.1.

Since $\mathsf{J}$ is a free category, the effect of the Kan extension is to map each $\widetilde{f}$ to a product. If $f = a_0 \xrightarrow{f_1} \cdots \xrightarrow{f_n} a_n$ is a path in $\mathsf{J}$ in which $S\widetilde{a_i} = X_i$, then $B\widetilde{f} = X_0 \times \cdots \times X_n$. When $n = 1$ and $f$ is an edge of the underlying graph, then $B\widetilde{f} = X_0 \times X_1$. Therefore, any $f$ transition in the fibre must be labelled with a subobject of $X_0 \times X_1$ (a relation). For $n > 1$, $\widetilde{f}$ is mapped to the apex of a span which represents an assertion expressed in terms of all the variables along the path.

Weakening between fibres of the same or different shape is defined as in first-order logic. Natural transformations whose components are projections can isolate any combination of program variables and particular occurrences of variables at arbitrary points in the program. All the propositional connectives are available as are the quantifiers.

Consider the following simple example.

$x := x + 2$
$x := 2 \times x \quad [x : \mathbb{N}]$

The property we wish to express is that $x$ is even at the end of the second transition. This requires a base object $\mathbf{B}$, and subobjects $\mathbf{S}$ and $\mathbf{P}$ for the program and property respectively. The situation is depicted in diagram 11.2. The solid arrows in the diagram depict three systems. All the unlabelled morphisms are projections. The base $\mathbf{B}$ is at the bottom. The program $\mathbf{S}$ (upper left) is constructed according to the scheme outlined in the previous section. The system $\mathbf{P}$ (upper right) is the largest subobject of $\mathbf{B}$ having the desired property. The dotted arrows are the components of the natural transformations relating the three functors. All the components are monomorphisms.

Often safety and liveness are duals and this is the case here. To express liveness the ordering in the fibre is reversed. If $\mathbf{S}$ is the system representing a program and $\mathbf{Q}$ is a liveness property and both are in the fibre of subobjects over $\mathbf{B}$, then $\mathbf{S}$ has property $\mathbf{Q}$ when $\mathbf{Q}$ factors through $\mathbf{S}$. The program $\mathbf{S}$ is then obliged to compute everything specified by $\mathbf{Q}$.

This is often stronger than necessary. Usually it suffices to express only that a particular state is not deadlocked, that a particular transition can fire or that the program can terminate. This can be done by moving to a different fibre. Assume $\mathbf{B} = (\mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{B} \mathsf{Set})$ is the base of the fibre. Define $\mathbf{L} = (\mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{L} \mathsf{Set})$ such that $L\widetilde{f} = 1$ for all $f \in \mathsf{J}$ and define the morphism $(Id, \eta)$ where $\eta_{\widetilde{f}} : B\widetilde{f} \longrightarrow 1$. Forward properties above $\mathbf{L}$ assert only that some point or points in $\mathsf{J}$ are reachable or that some transitions may fire. The functors $\forall, \exists : Sub(\mathbf{B}) \longrightarrow Sub(\mathbf{L})$ (the adjoints of $(Id, \eta)^*$) transfer properties into the fibre over $\mathbf{L}$.

## 12 Forward and Backward Systems

A system constructed according the scheme outline in section 8 may fail to exhibit properties which intuitively the program should have. As discussed earlier, tuples in the product $\widetilde{Sb}$ represent variable assignments at point $b \in \mathsf{J}$. The problem is that often there are tuples in $\widetilde{Sb}$ which do not correspond to a computation. That is, there is no initial variable assignment and computation path to $b$ which computes the tuple. Consider the fragment:

$x := 2 \times x;$
$x := x + 2 \quad [x : \mathbb{N}]$

$$\begin{tikzcd}
N \arrow[rrrrrrr, dotted] \arrow[ddddddrrr, dotted] & & & & & & & N \arrow[ddddddllll, dotted] & & & & & & \\
& & N \arrow[ull, "id"'] \arrow[rrrrrrr, dotted] \arrow[ddr, "\lambda x.x+1"'] \arrow[ddddddrrr, dotted] & & & & & & & N^2 \arrow[ull] \arrow[ddr] \arrow[ddddddllll, dotted] & & & & \\
& & & & N \arrow[ull, "id"'] \arrow[dl] \arrow[ddr, "\lambda x.x+1"] \arrow[rrrrrrr, dotted] \arrow[ddddddrrr, dotted] & & & & & & & N^3 \arrow[ull] \arrow[dl] \arrow[ddr] \arrow[ddddddllll, dotted] & & \\
& & & N \arrow[rrrrrrr, dotted] \arrow[ddddddrrr, dotted] & & & & & & & N \arrow[ddddddllll, dotted] & & & \\
& & & & & N \arrow[ull, "id"'] \arrow[dr, "\lambda x.2x"'] \arrow[rrrrrrr, dotted] \arrow[ddddddrrr, dotted] & & & & & & & N^2 \arrow[ull] \arrow[dr, "\lambda(x,y).2y"] \arrow[ddddddllll, dotted] & \\
& & & & & & N \arrow[rrrrrrr, dotted] \arrow[ddddddrrr, dotted] & & & & & & & N \arrow[ddddddllll, dotted] \\
& & & N & & & & & & & & & & \\
& & & & & N^2 \arrow[ull] \arrow[ddr] & & & & & & & & \\
& & & & & & & N^3 \arrow[ull] \arrow[dl] \arrow[ddr] & & & & & & \\
& & & & & & N & & & & & & & \\
& & & & & & & & N^2 \arrow[ull] \arrow[dr] & & & & & \\
& & & & & & & & & N & & & &
\end{tikzcd}$$

Diagram 11.2.

The value of $x$ is even at then end of the block but this is not reflected in the postcondition of the span interpreting the second command. This motivates the following definition:

Definition 12.1 *An algorithmic system* $\mathbf{S}$ *is* forward *if for all objects* $b \in \mathsf{J}$, *the collection of arrows*

$$\{S\widetilde{f} \xrightarrow{S(f,b)} S\widetilde{b} \mid f \in \mathsf{J}(\nabla, b)\}$$

*is an epimorphic family.*

The definition means that the forward legs of the spans relating the initial state to any other state are a covering for the variable assignments at that state. Alternatively, every variable assignment at every reachable state is justified by some initial variable assignment and a computation path to that state. We write $\mathsf{FS}$ for the full subcategory of systems which are forward.

For each algorithmic system $\mathbf{S}$, there is a forward system $\overrightarrow{\mathbf{S}}$ which is the most precise description of the behaviour of the program. If $\mathbf{S} = (\mathsf{1} + \mathsf{1} \longrightarrow \mathsf{J}, \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set})$, then $\overrightarrow{\mathbf{S}}$ has the same shape and a functor $\overrightarrow{S}$ which is the largest forward algorithmic subfunctor of $S$. It represents the strongest assertion which can be made about the program. The name "forward" comes from "forward collecting semantics" in the Cousots' framework for abstract interpretation [CC77]. A forward collecting semantics associates with each program point, the set of all possible variable assignments at that point. This corresponds to the object part of a forward system. If $S : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ is the functor of a forward system then the composite functor $|\mathsf{J}| \rightarrowtail \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set}$ is the forward collecting semantics in the Cousot sense. It follows that a forward system carries considerably more information. It records how each variable assignment at each program point is obtained, what transitions can fire from each state, and the relationship (if any) between variable assignments between arbitrary program points.

Below we provide two ways of constructing forward systems. In the first $\overrightarrow{S}$ is the union of forward subobjects. This requires showing that $\mathsf{FS}$ is well-complete.

Lemma 12.2 *The coproduct of forward systems is forward.*

Proof Consider the sum $S + T$ with injections $\mathsf{inl} : S \longrightarrow S + T$ and $\mathsf{inr} : T \longrightarrow S + T$ and the family of arrows:

$$\{S\widetilde{f} + T\widetilde{f} \xrightarrow{S(f,b)+T(f,b)} S\widetilde{b} + T\widetilde{b} \mid f \in \mathsf{J}(\nabla, b)\}$$

Given the pair $S\widetilde{b} + T\widetilde{b} \underset{k}{\overset{h}{\rightrightarrows}} C$ and assuming the sets $\{S(f,b) \mid f \in \mathsf{J}(\nabla, b)\}$ and $\{T(f,b) \mid f \in \mathsf{J}(\nabla, b)\}$ are epimorphic families, then

$$\begin{aligned}
&\forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ (S(f,b) + T(f,b)) = k \circ (S(f,b) + T(f,b)) \\
&\implies \forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ (S(f,b) + T(f,b)) \circ \mathsf{inl}_{\widetilde{f}} = k \circ (S(f,b) + T(f,b)) \circ \mathsf{inl}_{\widetilde{f}} \\
&\qquad \wedge\ h \circ (S(f,b) + T(f,b)) \circ \mathsf{inr}_{\widetilde{f}} = k \circ (S(f,b) + T(f,b)) \circ \mathsf{inr}_{\widetilde{f}} \\
&\implies \forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ \mathsf{inl}_b \circ S(f,b) = k \circ \mathsf{inl}_b \circ S(f,b) \\
&\qquad \wedge\ h \circ \mathsf{inr}_b \circ T(f,b) = k \circ \mathsf{inr}_b \circ T(f,b) \\
&\implies h \circ \mathsf{inl}_b = k \circ \mathsf{inl}_b \ \wedge\ h \circ \mathsf{inr}_b = k \circ \mathsf{inr}_b \\
&\implies h = k
\end{aligned}$$

□

Lemma 12.3 *Forwardness is stable under image factorization. Let $\eta : S \Longrightarrow T$ be a natural transformation where $S$ is the functor of a forward system. If $\eta$ has an image factorization $\eta = S \overset{\epsilon}{\Longrightarrow} M \overset{\mu}{\Longrightarrow} T$, then $M$ is forward.*

Proof The diagram below shows the factorization of a naturality diagram for a morphism $(l,m) : \widetilde{f} \longrightarrow \widetilde{g} \in \widetilde{\mathsf{J}}$. The dotted arrow is uniquely determined by the factorizations of the components $\eta_{\widetilde{f}}$ and $\eta_{\widetilde{g}}$.

$$\begin{tikzcd}
S\widetilde{f} \arrow[r, two heads, "\epsilon_{\widetilde{f}}"] \arrow[d, "S(l,m)"'] & M\widetilde{f} \arrow[r, tail, "\mu_{\widetilde{f}}"] \arrow[d, dotted, "M(l,m)"'] & T\widetilde{f} \arrow[d, "T(l,m)"] \\
S\widetilde{g} \arrow[r, two heads, "\epsilon_{\widetilde{g}}"'] & M\widetilde{g} \arrow[r, tail, "\mu_{\widetilde{g}}"'] & T\widetilde{g}
\end{tikzcd}$$

Now consider the naturality diagrams for the morphisms $\{(f,b) \mid f \in \mathsf{J}(\nabla, b)\}$ for some $b \in \mathsf{J}$. Let $M\widetilde{b} \underset{k}{\overset{h}{\rightrightarrows}} C$ be a parallel pair in $\mathsf{Set}$.

$$\begin{aligned}
&\forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ M(f,b) = k \circ M(f,b) \\
&\qquad \implies \forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ M(f,b) \circ \epsilon_{\widetilde{f}} = k \circ M(f,b) \circ \epsilon_{\widetilde{f}} \\
&\qquad \implies \forall f \in \mathsf{J}(\nabla, b) \,.\, h \circ \epsilon_{\widetilde{b}} \circ S(f,b) = k \circ \epsilon_{\widetilde{b}} \circ S(f,b) \\
&\qquad \implies h \circ \epsilon_{\widetilde{b}} = k \circ \epsilon_{\widetilde{b}} \qquad \{S(f,b) \mid f \in \mathsf{J}(\nabla, b)\} \text{ is an epimorphic family} \\
&\qquad \implies h = k \qquad \epsilon_{\widetilde{b}} \text{ is epimorphic}
\end{aligned}$$

□

The combination of lemmas 12.2 and 12.3 ensures that $S \cup T$ is forward when $S$ and $T$ are forward.

The construction of forward systems extends to a functor $\overrightarrow{(-)} : \widetilde{\mathsf{CTS}} \longrightarrow \mathsf{FS}$ as follows. Let $S : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ and $T : \widetilde{\mathsf{K}} \longrightarrow \mathsf{Set}$ be systems and $(F, \eta) : \mathbf{S} \longrightarrow \mathbf{T}$ be a morphism. Let $\nu : \overrightarrow{S} \rightarrowtail S$ be the forward subfunctor of $S$. This determines a subfunctor $\mu : R \rightarrowtail T$ of $T$ by the image factorization $\mu \circ \epsilon = \eta \circ \nu\widetilde{F} : \overrightarrow{S}\widetilde{F} \Longrightarrow T$. By lemma 12.3, $R$ is itself forward and hence $\mu$ factors uniquely through the forward subfunctor for $T$ by a natural transformation $\delta : R \rightarrowtail \overrightarrow{T}$. This yields a morphism $(F, \delta \circ \epsilon) : \overrightarrow{\mathbf{S}} \longrightarrow \overrightarrow{\mathbf{T}}$.

Next we sketch an alternative definition of the functor $\overrightarrow{(-)} : \widetilde{\mathsf{CTS}} \longrightarrow \mathsf{FS}$ which generalizes predicate transformers defined earlier for individual spans.

Given an algorithmic functor $S : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$, then for each object $b \in \mathsf{J}$ consider the image factorization of the family of $\{S(f, b) \mid f \in \mathsf{J}(\nabla, b)\}$ as shown in diagram 12.1.

$$\begin{tikzcd}
S\widetilde{f}_1 \arrow[dr, "S(f_1, b)"] & \\
\vdots & S\widetilde{b} \\
S\widetilde{f}_n \arrow[ur, "S(f_n, b)"'] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
S\widetilde{f}_1 \arrow[dr] & & \\
\vdots & Qb \arrow[r, tail, "\phi_b"] & S\widetilde{b} \\
S\widetilde{f}_n \arrow[ur] & &
\end{tikzcd}$$

Diagram 12.1.

Let $J : |\mathsf{J}| \longrightarrow \widetilde{\mathsf{J}}$ be the obvious embedding and define the functor $|S| : |\mathsf{J}| \xrightarrow{J} \widetilde{\mathsf{J}} \xrightarrow{S} \mathsf{Set}$. Define the system $\mathbf{Q} : |\mathsf{J}| \longrightarrow \mathsf{Set}$ such that $Qb$ is the image as shown in diagram 12.1. This gives two morphisms of systems:

$$(Id, \phi) : \mathbf{Q} \longrightarrow |\mathbf{S}| \qquad (J, id) : \mathbf{S} \longrightarrow |\mathbf{S}|$$

The forward system $\overrightarrow{\mathbf{S}}$ is the pullback:

$$\begin{tikzcd}
\overrightarrow{\mathbf{S}} \arrow[r, "{(J, id)}"] \arrow[d, "{(Id, \varepsilon)}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{Q} \arrow[d, "{(Id, \phi)}"] \\
\mathbf{S} \arrow[r, "{(J, id)}"'] & |\mathbf{S}|
\end{tikzcd}$$

Given systems, $\mathbf{S} : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ and $\mathbf{T} : \widetilde{\mathsf{K}} \longrightarrow \mathsf{Set}$ and a morphism $(F, \eta) : \mathbf{S} \longrightarrow \mathbf{T}$, then $\overrightarrow{(F, \eta)}$ is the universal arrow as shown in diagram 12.2. The natural transformation $|\eta|$ consists of the components of $\eta$ indexed by the objects of $|\mathsf{J}|$ (rather than $|\widetilde{\mathsf{J}}|$). For the natural transformation $\rho$, consider

$$\begin{tikzcd}
\overrightarrow{\mathbf{S}} \arrow[rr] \arrow[dr, dotted, "\overrightarrow{(F,\eta)}"] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[dd, tail] & & \mathbf{Q} \arrow[dr, "(|F|,\rho)"] \arrow[dd, tail, "(Id,\theta)"] & \\
& \overrightarrow{\mathbf{T}} \arrow[rr] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[dd, tail] & & \mathbf{R} \arrow[dd, tail, "(Id,\phi)"] \\
\mathbf{S} \arrow[rr, "(J,id)"] \arrow[dr, "(F,\eta)"'] & & |\mathbf{S}| \arrow[dr, "(|F|,|\eta|)"'] & \\
& \mathbf{T} \arrow[rr, "(K,id)"'] & & |\mathbf{T}|
\end{tikzcd}$$

Diagram 12.2.

$$\begin{tikzcd}
& S\widetilde{g}_m \arrow[dr, "S(g_m,Fb)"] & & \\
& & \overrightarrow{S}\widetilde{F}b \arrow[r, tail, "\theta_{\widetilde{F}b}"] \arrow[ddd, tail, "h"] & S\widetilde{\widetilde{F}b} \arrow[ddd, equal] \\
S\widetilde{g}_1 \arrow[uur, dotted, no head] \arrow[urr, "S(g_1,Fb)"'] & & & \\
& S\widetilde{F}\widetilde{f}_n \arrow[dr, "S(Ff_n,Fb)"] \arrow[ddd, "\eta_{\widetilde{f}_n}"'] & & \\
& & X \arrow[r, tail, "k"] \arrow[ddd, dotted, "l"] & S\widetilde{\widetilde{F}b} \arrow[ddd, "|\eta|_b"] \\
S\widetilde{F}\widetilde{f}_1 \arrow[uur, dotted, no head] \arrow[urr, "S(Ff_1,Fb)"'] \arrow[ddd, "\eta_{\widetilde{f}_1}"'] & & & \\
& T\widetilde{f}_n \arrow[dr, "T(f_n,b)"] & & \\
& & \overrightarrow{T}b \arrow[r, tail, "\phi_b"'] & T\widetilde{b} \\
T\widetilde{f}_1 \arrow[uur, dotted, no head] \arrow[urr, "T(f_1,b)"'] & & &
\end{tikzcd}$$

Diagram 12.3.

the image factorization of the three families of arrows as shown in diagram 12.3.

The lower plane shows the factorization for $\overrightarrow{T}b$. In the middle is the factorization of the family:

$$\{S\widetilde{F}\widetilde{f_i} \xrightarrow{S(Ff_i,Fb)} S\widetilde{F}b \mid f_i \in \mathsf{K}(\nabla, b)\}$$

This is a subset of the family of arrows:

$$\{S\widetilde{g_i} \xrightarrow{S(g_i,Fb)} S\widetilde{F}b \mid g_i \in \mathsf{J}(\nabla, Fb)\}$$

which appear at the top level and whose image factorization is $\overrightarrow{S}(Fb)$. Hence there is a unique $h$ such that $\theta_{\widetilde{F}b}$ factors via $h$ through $k$. The morphism $l$ is the unique arrow determined between the two image factorizations. Define $\rho_b = l \circ h$.

Lemma 12.4 $\mathsf{FS}$ *is a reflective subcategory of* $\widetilde{\mathsf{CTS}}$.

$$\begin{tikzcd}
\widetilde{\mathsf{CTS}} \arrow[r, shift left, "\overrightarrow{(-)}"] \arrow[r, phantom, "\top"] & \mathsf{FS} \arrow[l, shift left, hook']
\end{tikzcd}$$

All the definitions and lemmas for forward systems have duals.

Definition 12.5 *An algorithmic system* $\mathbf{S}$ *is* backward *if for all objects* $b \in \mathsf{J}$*, the collection of arrows*

$$\{S\widetilde{b} \xrightarrow{S(b,f)} S\widetilde{f} \mid f \in \mathsf{J}(b, \triangle)\}$$

*is an epimorphic family.*

Intuitively, if $\mathbf{S}$ is backward, then for each object $b$ in the shape $\mathsf{J}$, a tuple in $Sb$ represents a variable assignment for which there exists a terminating computation. Naturally there is a category of backward systems $\mathsf{BS}$ and a functor $\overleftarrow{(-)} : \widetilde{\mathsf{CTS}} \longrightarrow \mathsf{BS}$. Backward system correspond to backward collecting semantics in abstract interpretation.

A system can be both forward and backward. Let $\mathsf{FBS}$ be the category of algorithmic systems which are both forward and backward. If $\mathbf{S}$ is forward, then $\overleftarrow{\mathbf{S}}$ remains forward. The easily seen by considering the second construction of forward systems. Pullbacks are computed pointwise and $\mathsf{Set}$ is a regular, epimorphisms (and epimorphic families) are stable under pullback. The converse is also true, if $\mathbf{S}$ is backward, then $\overrightarrow{\mathbf{S}}$ remains backward. The following lemma follows directly.

Lemma 12.6 $\overrightarrow{(-)} \circ \overleftarrow{(-)} \cong \overleftarrow{(-)} \circ \overrightarrow{(-)} : \widetilde{\mathsf{CTS}} \longrightarrow \mathsf{FBS}$

Finally there are constructions analogous to forward and backward systems for categories of twisted elements. Given a system $\mathbf{S} : \widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ then $\oint S$ is forward when all the objects in $\oint S$ are reachable from an object in the fibre above $\nabla_{\mathsf{J}}$. The category $\overrightarrow{\oint S}$ is obtained by discarding unreachable objects. The dual holds for $\overleftarrow{\oint S}$.

# 13 Conclusions

We have show that the operational semantics of a program can be expressed using twisted systems, and moreover, that the system representing the program can be constructed compositionally. The functor obtained in this way is closely related to the transition relation of a operational semantics via the category of twisted elements. Procedures are accommodated though it should be said that the semantics of procedures is less precise than it should be.

We have also shown that both programs and properties of programs can be structured in a manner analogous to categorical models of predicate logic. We hope to find program logics or perhaps first-order modal or temporal logics which can be interpreted in this framework. Related to this is possibility of presenting Cousots theory of abstract in categorical terms.

There is considerable structure available which has yet to be exploited. We have not made use of the fibrational structure of the category of twisted systems, and only limited use of the "bifibrations" produced by the category of twisted elements. Nor have we made use of the fact that category of programs and properties of a particular shape form a presheaf topos.

Finally, twisted arrow categories have featured prominently in this paper. To the best of our knowledge the relationship between twists and spans summarized here is novel. The connection is also exploited in [Err96, Err] in the context of communicating processes. We expect further applications of twisted arrow categories in computing will follow.

# Acknowledgements

I wish to thank three successive office mates: Francois Lamarche, Dusko Pavlović and Till Plewe. What category theory I know, I learned from them or with their help. Francois is directly responsible for some of the material which appears in this paper and I have attached his name to those constructions and results for which he is wholly responsible.

I also wish to thank David Clark and my supervisor Prof. Chris Hankin for their support and advice, both technical and otherwise.

# References

[AP97] Samson Abramsky and Dusko Pavlović. Specifying interaction categories. In E. Moggi et al., editor, *Category Theory and Computer Science '97*, Lecture Notes in Computer Science, page 14. Springer Verlag, 1997. to appear.

[Bén67] Jean Bénabou. Introduction to bicategories. *Lecture Notes in Mathematics*, 47, 1967.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra 1, Basic Category Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994.

[Bur72] Rod Burstall. An algebraic description of programs with assertions, verification and simulation. In J. Mack Adams, John Johnston, and Richard Stark, editors, *Conference on Proving Assertions about Programs*, pages 7–14. ACM, 1972.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM POPL*, pages 238–252, 1977.

[CKS84] Aurelio Carboni, Stefano Kasangian, and Ross Street. Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, 33:259–267, 1984.

[Dij75] E.W. Dijkstra. Guarded commands. *Communications of the ACM*, 18(8):453–457, 1975.

[Err] Lindsay Errington. *Categorical Transition Systems.* PhD thesis, Imperial College, In preparation.

[Err96] Lindsay Errington. Categories of processes with state. In *Third Theory and Formal Methods Workshop.* IC Press, 1996.

[FS90] Peter J. Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library.* North-Holland, Amsterdam, 1990.

[GJ92] Eric Goubault and Thomas Jensen. Homology of higher-dimensional automata. In *CONCUR '92*, Lecture Notes in Computer Science. Springer Verlag, 1992.

[Gla91] R.J. van Glabbeek. Bisimulations for higher dimensional automata. Email message, July 7, 1991, 1991. Available at `http://theory.stanford.edu/~rvg/hda`.

[GM83] Joseph Goguen and José Meseguer. Correctness of recursive parallel non-deterministic flow programs. *Journal of Computer and System Sciences*, 27(2):268–290, 1983.

[Gog74] Joseph Goguen. On homomorphisms, correctness, termination, unfoldments and equivalence of flow diagram programs. *Journal of Computer and System Sciences*, 8:333–365, 1974.

[Gog91] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[Gou95] Eric Goubault. *Géométrie du Parallélisme*. PhD thesis, École Polytechnique, 1995.

[Gra66] J.W. Gray. Fibred and cofibred categories. In S. Eilenberg, D.K. Harrison, S. MacLane, and H. Röhrl, editors, *Conference on Categorical Algebra*, pages 21–83. Springer Verlag, 1966.

[Joh77] Peter T. Johnstone. *Topos Theory*. Academic Press, 1977.

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.

[Pit] Andrew M. Pitts. Categorical logic. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press. To appear.

[Ple96] Till Plewe. Specifications as spans of geometric morphisms. In *Third Theory and Formal Methods Workshop*. IC Press, 1996.

[Pra91] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.

[RR88] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.

[TBG91] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991.

[Vic95] S.J. Vickers. Geometric logic as a specification language. In *Proceedings of the 1994 Theory and Formal Methods Section Workshop*. Imperial College Press, 1995.