

Getting to Grips with Sweedler Runners for Systems

Appendix A: the IntRun construction

September 2026

A A first look at $\text{IntRun}(\mathcal{D})$

The construction $\text{Run}(\mathcal{D})$ has given us a useful way to speak about one theory running on another. There is a slightly richer construction, which we shall denote by

$$\text{IntRun}(\mathcal{D}),$$

whose objects are pairs of theories. In the examples below we continue to work in

$$\mathcal{D} = [\mathbf{Set}, \mathbf{Set}]_{\text{acc}},$$

so that our theories are accessible monads on $\mathbf{Set}$. The purpose of this appendix is not to develop the abstract theory of IntRun . Instead we shall try to get an operational feeling for what one of its loose arrows is saying. The resulting picture is rather close to a stateful prism.

A.1 The prism shape before theories

Before putting whole theories into the four slots, it is worth isolating the underlying shape of the construction in a much simpler setting. Think of an interface as having two faces,

$$(A^-, A^+),$$

and another as having two faces

$$(B^-, B^+).$$

Very loosely, A^+ and B^+ may be thought of as forward or continuation channels, while A^- and B^- are the corresponding return or redirected channels. A prism-shaped interaction between these interfaces has a crossed character. Something arriving on one side may either continue through or be redirected across to the other side. Schematically one encounters maps of the form

$$B^- + A^+ \longrightarrow B^+ + A^-,$$

or, in pointed settings where termination or failure is allowed,

$$B^- + A^+ \longrightarrow B^+ + A^- + 1.$$

The precise formula is less important here than the geometry it suggests:

one channel passes through, while another may be redirected across.

This is the basic prism picture. It already appears in familiar settings of partial maps, control flow and feedback. No algebraic theories are needed to see it. The construction considered in this

appendix has the same general shape, but the four faces are no longer mere sets or data types. They are whole theories. A loose arrow

$$(T, T') \rightsquigarrow (U, U')$$

contains a through leg

$$T' \rightsquigarrow U'$$

and a prism leg

$$U \rightsquigarrow T + U'.$$

Thus the simple crossed picture has been upgraded as follows:

simple prism picture	IntRun picture
data type or channel	monad/theory
map between channels	stateful runner
either side of a coproduct	computation in the combined theory $T + U'$
hidden control state	residual runner state S .

So the apparent complexity of

$$\text{IntRun}([\mathbf{Set}, \mathbf{Set}]_{\text{acc}})$$

comes largely from what is being placed on the four faces. The underlying geometry is comparatively simple: one family of behaviour passes through, while another is handled using both the continuation side and an additional fallback side. One may therefore think of the passage

$$\text{simple prism} \rightsquigarrow \text{IntRun}([\mathbf{Set}, \mathbf{Set}]_{\text{acc}})$$

as replacing ordinary channels by complete languages of operations. This point of view will be useful in the examples below. For instance, in

$$(W_C, W_A) \rightsquigarrow (W_B, W_D),$$

the four faces carry free unary theories rather than bare sets. The through leg sends A -commands into D -computations, while the prism leg sends B -commands into computations built from both C - and D -commands. The geometry is still the same crossed geometry; only the labels on the channels have become much richer.

Now consider a loose arrow

$$(T, T') \rightsquigarrow (U, U')$$

with residual state space S contains two pieces of running data, sharing the same state S :

$$T' \rightsquigarrow U', \quad U \rightsquigarrow T + U'.$$

Here $T + U'$ denotes the coproduct of the two theories. It is useful to give these two pieces names. We shall call

$$T' \rightsquigarrow U'$$

the *through leg*: operations from T' are implemented directly using the theory U' . The second piece

$$U \rightsquigarrow T + U'$$

is the *prism leg*: an operation from U may be implemented using the combined language generated by T and U' . Thus the four positions have different operational roles:

position	informal role
T'	operations which must pass through
U'	the theory into which they pass
U	operations which are inspected or handled
T	an additional or fallback theory available when handling them.

The two pieces are not independent: they use the same residual state. It is this shared state which makes the loose arrow into one stateful device rather than a pair of unrelated runners.

A.2 Free unary commands

Let

$$W_A := F(Ay),$$

the free monad generated by one unary operation for every $a \in A$. Explicitly,

$$W_A(X) \cong A^* \times X.$$

Thus W_A is simply the writer monad for the free monoid A^* . An ordinary stateful runner

$$W_A \rightsquigarrow W_D$$

is determined by its action on a single generator:

$$A \times S \longrightarrow D^* \times S.$$

If

$$(a, s) \longmapsto (d_1 \cdots d_n, s'),$$

we read this as follows: in state s , an incoming A -command is implemented by the finite sequence of D -commands

$$d_1, \dots, d_n,$$

and the residual state becomes s' . The action on a word

$$a_1 \cdots a_k$$

is then obtained by processing the letters successively, carrying the residual state along from one step to the next. Now consider an IntRun arrow

$$(W_C, W_A) \rightsquigarrow (W_B, W_D).$$

Its through leg is

$$W_A \rightsquigarrow W_D,$$

and is therefore described by

$$\alpha : A \times S \longrightarrow D^* \times S.$$

The prism leg is

$$W_B \rightsquigarrow W_C + W_D.$$

Since the coproduct of the two freely generated unary theories is again freely generated by the disjoint union of their generators,

$$W_C + W_D \cong W_{C+D},$$

this second runner is described by

$$\beta : B \times S \longrightarrow (C + D)^* \times S.$$

Thus the whole loose arrow may be pictured simply as

$$\boxed{\begin{array}{l} A \times S \longrightarrow D^* \times S, \\ B \times S \longrightarrow (C + D)^* \times S. \end{array}}$$

This is perhaps the most elementary example of the prism character of IntRun. An A -command has to pass into the D -channel. A B -command, by contrast, can produce a computation involving both C -commands and D -commands. For example, one might have

$$\beta(b, s) = (c_1 d_1 d_2 c_2 d_3, s').$$

The point is not that the command chooses once and for all between the C -side and the D -side. Rather, it may generate a whole mixed computation in the combined theory. In the stateless case $S = 1$ this reduces to the particularly simple pair

$$A \longrightarrow D^*, \quad B \longrightarrow (C + D)^*.$$

So even without hidden state, the prism leg already has more structure than a plain map of signatures.

A.3 Recovering an ordinary transition system

The previous example contains familiar machines as special cases. Consider

$$(\text{Id}, \text{Id}) \rightsquigarrow (W_A, \text{Id}).$$

The through leg is trivial, while the prism leg reduces to a runner

$$W_A \rightsquigarrow \text{Id}.$$

This is just an action of the free monoid A^* on a set S , equivalently a map

$$A \times S \longrightarrow S.$$

Thus

$$\boxed{(\text{Id}, \text{Id}) \rightsquigarrow (F(Ay), \text{Id})}$$

is nothing more exotic than an A -labelled deterministic transition system. This is a useful sanity check. The elementary machines already encountered in $\text{Run}(\mathcal{D})$ have not disappeared; they sit naturally on the boundary of the richer construction.

A.4 A free observation operation

Now take

$$F(y^O),$$

the free monad generated by a single O -ary operation. Before imposing any equations, a comodel of this theory consists of a map

$$S \longrightarrow O \times S.$$

Thus performing the observation at a state s can both return an element $o \in O$ and change the state:

$$s \longmapsto (o, s').$$

Accordingly,

$$(\text{Id}, \text{Id}) \rightsquigarrow (F(y^O), \text{Id})$$

describes a stateful observation device. This is worth comparing with the reader monad.

A.5 The reader monad

The actual reader monad is

$$y^O(X) = X^O.$$

It is obtained from the freely generated observation theory by imposing the reader equations. In particular, repeated observations must behave as repeated reads of the same environment. A comodel

$$y^O \rightsquigarrow \text{Id}$$

therefore reduces to a map

$$S \longrightarrow O.$$

The read operation observes the current O -value but does not itself update the state. So we have the useful contrast

$\begin{array}{ll} F(y^O) & : S \longrightarrow O \times S, \\ y^O & : S \longrightarrow O. \end{array}$
--

The distinction is small syntactically but important operationally. A free observation may disturb the machine. A lawful reader is genuinely read-only. In a more general IntRun arrow, one can place either theory in one of the active slots. The same distinction remains: the equations of the reader theory constrain what an implementation is allowed to do. It is therefore important that IntRun works with theories, rather than merely with collections of operation symbols.

A.6 Moore machines

Let

$$M_{A,B}$$

denote the theory used earlier for Moore machines with input alphabet A and output set B . A comodel of this theory is an ordinary Moore machine

$$(S, \delta, o)$$

with

$$\delta : A \times S \longrightarrow S, \quad o : S \longrightarrow B.$$

Consequently,

$$(\text{Id}, \text{Id}) \rightsquigarrow (M_{A,B}, \text{Id})$$

is precisely a Moore machine. Again, this places an old example from $\text{Run}(\mathcal{D})$ inside $\text{IntRun}(\mathcal{D})$ without any change of meaning. More interesting things happen when the Moore theory occupies one of the other positions.

A.7 A Moore theory as the fallback language

Consider

$$(M_{A,B}, \text{Id}) \rightsquigarrow (W_C, \text{Id}).$$

The prism leg is now

$$W_C \rightsquigarrow M_{A,B}.$$

Thus a C -command is implemented by a computation in the Moore theory. Informally,

$$c \in C \rightsquigarrow \text{a program which interacts with the Moore machine.}$$

Such a program may perform A -transitions, inspect the current B -output, and continue according to the terms permitted by the Moore theory. This gives a useful interpretation of the first component T in a pair (T, T') . In the prism leg

$$U \rightsquigarrow T + U',$$

the theory T is an additional implementation language available when handling U . If instead we consider

$$(M_{A,B}, \text{Id}) \rightsquigarrow (W_C, W_D),$$

the prism leg becomes

$$W_C \rightsquigarrow M_{A,B} + W_D.$$

A C -command may now be implemented using both interactions with a Moore machine and ordinary D -commands. This is already close to the usual programming intuition behind a prism: successful behaviour can continue along one channel, while the other channel provides an alternative vocabulary in which the computation can be handled.

A.8 A reader together with a Moore interface

As a slightly richer example consider

$$(W_C, y^O) \rightsquigarrow (M_{A,B}, W_D).$$

There are two pieces. The through leg is

$$y^O \rightsquigarrow W_D.$$

Thus an O -valued read is implemented using D -behaviour. Because the source is the reader monad rather than the free observation theory, its equations impose the expected consistency on repeated reads. The prism leg is

$$M_{A,B} \rightsquigarrow W_C + W_D \cong W_{C+D}.$$

Hence the transition and observation operations of the Moore theory are implemented using computations built from both C -commands and D -commands. At the informal level one may therefore think of the same residual state S as supporting two related activities:

- an O -observation which is implemented through the D -channel;
- a Moore interface whose operations may generate a mixture of C - and D -behaviour.

The D -side is privileged in the sense that the through leg must land there. The Moore operations, on the other hand, may make use of both sides.

A.9 Moore machine against Moore machine

Finally consider

$$(M_{A,B}, M_{C,D}) \rightsquigarrow (M_{E,F}, M_{G,H}).$$

The through leg is

$$M_{C,D} \rightsquigarrow M_{G,H}.$$

This is a stateful implementation of one Moore interface in another. A C -transition is realised using the (G, H) -machine, while the available H -observations are used to provide the required D -observations. So this leg can be thought of as an adapter

$$(C, D) \longrightarrow (G, H)$$

carrying its own residual state. The prism leg is

$$M_{E,F} \rightsquigarrow M_{A,B} + M_{G,H}.$$

An (E, F) -Moore interface is therefore implemented in the combined theory generated by two other Moore interfaces. The important point is that the coproduct is not merely a once-and-for-all choice between the two machines. A single implementation may involve a mixed computation:

$$A\text{-step}; \quad G\text{-step}; \quad \text{observe } H; \quad A\text{-step}; \quad \text{observe } B; \quad \dots$$

subject, of course, to the equations of the theories involved. Thus an arrow

$$(M_{A,B}, M_{C,D}) \rightsquigarrow (M_{E,F}, M_{G,H})$$

can be read informally as a stateful open-system adapter with two aspects:

- one interface is passed straight through and implemented in another;
- a second interface is allowed to interact with both the fallback side and the through side.

A.10 Ordinary runners as a boundary case

It is useful to see that the older construction is still present. Since the identity theory contributes no additional operations, a loose arrow of the form

$$(\text{Id}, \text{Id}) \rightsquigarrow (U, R)$$

essentially contains just an ordinary runner

$$U \rightsquigarrow R.$$

In particular,

$$(\text{Id}, \text{Id}) \rightsquigarrow (T, \text{Id})$$

is simply a T -comodel. So one may regard $\text{Run}(\mathcal{D})$ as occupying a boundary of $\text{IntRun}(\mathcal{D})$. The latter does not replace the runner picture, but enlarges it by allowing an operation to be handled in a combined theory.

A.11 A small dictionary

For the examples above, the following table is a useful guide:

theory or position	operational reading
$F(Ay)$	stateful commands labelled by A
$F(y^O)$	an observation which may change state
y^O	a lawful read-only observation
$M_{A,B}$	a Moore-machine interface
$T' \rightsquigarrow U'$	the through part
$U \rightsquigarrow T + U'$	the matching or prism part
T	fallback or auxiliary implementation language
U'	the continuation language.

The simplest slogan is therefore

a runner tells us how one theory runs on another,

while

an IntRun arrow lets one interface pass through while another is handled using both sides.

The shared residual state is what binds the two operations together. From the point of view of systems theory, this makes an IntRun arrow look less like an isolated translation between theories and more like an *open component equipped with an internal adapter state*. Even the elementary examples above suggest why this may be useful. Commands, observations and complete machine interfaces can all occupy the different slots, and the prism leg allows behaviour to be assembled from two theories at once. The construction therefore adds a modest but significant piece of structure to the runner picture: not merely “how does this run?”, but also “when this interface is encountered, which part continues straight through, and which part is handled by the surrounding system?”

A.12 Handlers and the module viewpoint

There is another useful way to read a loose IntRun arrow, namely through its action on handlers. Recall first the ordinary runner situation. If

$$P \rightsquigarrow Q$$

is a runner with residual state S , then handlers may be transported along it in the two familiar directions. Delegation carries a Q -handler on a carrier X to a P -handler on a carrier of the form

$$X^S,$$

while induction carries a P -handler to a Q -handler on a carrier of the form

$$X \times S.$$

For present purposes we shall concentrate on delegation, since it makes the prism shape especially transparent. Now take a loose IntRun arrow

$$(T, T') \rightsquigarrow (U, U')$$

with residual state S . Its two pieces are

$$T' \rightsquigarrow U', \quad U \rightsquigarrow T + U'.$$

Suppose we are given, on the same carrier X , a T -handler and a U' -handler:

$$TX \longrightarrow X, \quad U'X \longrightarrow X.$$

Together these amount to a handler for the coproduct theory

$$T + U'.$$

Indeed, to handle a computation in the combined theory one simply supplies a way to handle the operations from each of the two constituent theories. We may therefore delegate this combined handler along the prism leg

$$U \rightsquigarrow T + U'.$$

This produces a U -handler on the carrier

$$X^S.$$

At the same time, the given U' -handler may be delegated along the through leg

$$T' \rightsquigarrow U',$$

producing a T' -handler on the same carrier X^S . Thus a single IntRun component transforms a pair of handlers for the two “outside” theories into a pair of handlers for the two “inside” theories:

$T\text{-handler on } X$	$U'\text{-handler on } X$
$\Downarrow$	
$U\text{-handler on } X^S$	$T'\text{-handler on } X^S$

This crossed pattern is a useful way to remember what the prism is doing. The environment supplies handlers for the theories T and U' . The IntRun component then uses those handlers, together with its hidden state S , to manufacture handlers for U and T' . The asymmetry of the two legs is again visible. The T' -part passes directly through to the U' -handler. A U -operation, on the other hand, may be decomposed into a computation in the combined theory $T + U'$, and so may call upon both handlers supplied by the environment.

The unary-command example

For the elementary arrow

$$(W_C, W_A) \rightsquigarrow (W_B, W_D),$$

recall that its two pieces have the form

$$A \times S \longrightarrow D^* \times S,$$

and

$$B \times S \longrightarrow (C + D)^* \times S.$$

Now suppose that X carries a C -handler and a D -handler:

$$C \times X \longrightarrow X, \quad D \times X \longrightarrow X.$$

If a B -command is translated into a mixed word

$$c_1 d_1 d_2 c_2 d_3,$$

we handle it simply by applying the supplied C - and D -handlers in sequence. The hidden state records whatever extra information is required by the translation. In this way we obtain a B -handler on X^S . Similarly, an A -command is translated into a word containing only D -letters, so the supplied D -handler is enough to give an A -handler on the same carrier X^S . Thus in this example

C -handler	D -handler
$\Downarrow$	
B -handler	A -handler.

This is a particularly concrete instance of the general pattern.

The Moore-machine example

For an arrow

$$(M_{A,B}, M_{C,D}) \rightsquigarrow (M_{E,F}, M_{G,H}),$$

suppose the environment supplies an $M_{A,B}$ -handler and an $M_{G,H}$ -handler on X . Together these form a handler for

$$M_{A,B} + M_{G,H}.$$

Delegating this combined handler along

$$M_{E,F} \rightsquigarrow M_{A,B} + M_{G,H}$$

gives an $M_{E,F}$ -handler on X^S . Meanwhile the $M_{G,H}$ -handler alone may be delegated along

$$M_{C,D} \rightsquigarrow M_{G,H},$$

producing an $M_{C,D}$ -handler on the same carrier. Hence

$M_{A,B}$ -handler	$M_{G,H}$ -handler
$\Downarrow$	
$M_{E,F}$ -handler	$M_{C,D}$ -handler.

This makes the system-theoretic interpretation especially clear. One may think of the surrounding world as supplying ways to respond to the two exposed interfaces of a component. The component uses those responses, together with its own internal state, to provide the responses required at its other two interfaces.

A module-like picture

This suggests a useful informal module interpretation. An ordinary runner may be acted on by a handler for the theory on which it runs, thereby producing a handler for the theory being run. An IntRun component admits a two-sided version of the same idea: it is acted on by a pair of

handlers, one for T and one for U' , and produces a pair of handlers, one for U and one for T' . Schematically,

$$\boxed{((T, T') \rightsquigarrow (U, U')) + (T\text{-handler}, U'\text{-handler}) \longmapsto (U\text{-handler}, T'\text{-handler}).}$$

At this informal level it is therefore reasonable to think of an IntRun prism as a kind of two-sided module for handlers. We shall not try here to formalise that statement categorically. The important point is the operational one: the component can be plugged into an environment which knows how to handle its two exposed languages, and the result is a new pair of handlers on the other side. There is, of course, also the induction direction. Using the ordinary runner induction operation, handlers can be transported along the two legs with carriers of the form $X \times S$ rather than X^S . Thus the familiar induction–delegation story for Run is not replaced in IntRun; rather, it appears twice, arranged in the prism-shaped pattern determined by the two legs. This handler viewpoint also gives a useful intuition for composition. If two IntRun components are plugged together, handlers produced by one component can be fed into the exposed handler slots of the next. In that sense, composition of components looks operationally like composition of handler-transforming modules.

A.13 Wiring diagrams

Our IntRun construction has relied heavily on the interface geometry provided by the variety of optic designated a prism. An n -ary loose operation in $\text{IntRun}(\mathcal{D})$ should be regarded as a single stateful wiring box with n incoming paired interfaces and one outgoing paired interface. Its n through components and its one prism component are the directional faces of this box, all coupled through a common residual. In this sense it admits the same kind of string-diagrammatic reading familiar from lenses and prisms, but with whole theories decorating the wires.

There are many such kinds of optics, such as lenses and transversals, designed as bidirectional data accessors. Some of these may be worth exploring in the context of $\text{Run}(\mathcal{D})$.

A.14 Dependent wiring and parameterised theories

In Part III we already encountered theories which vary with an external parameter. The basic example was the Moore theory

$$(A, B) \longmapsto T_{\text{Moore}(A, B)},$$

where the available operations depend on the chosen input and output types. There we suggested that the natural setting is not one fixed category of theories, but rather a family of categories of theories varying over a base of typed interfaces. The same point becomes even more important for IntRun. So far, our examples have treated a loose operation

$$(T_1, T'_1) + \cdots + (T_n, T'_n) \rightsquigarrow (U, U')$$

as if all of the theories involved already lived in one common ambient category. The corresponding prism data can then be written schematically as

$$T'_i \rightsquigarrow U' \quad (i = 1, \dots, n),$$

together with

$$U \rightsquigarrow T_1 + \cdots + T_n + U'.$$

For genuinely typed systems this is often too rigid. The theories attached to different interfaces may live in different fibres and must first be transported along the wiring before they can interact. This suggests a fibred or dependent version of the construction. Suppose there is a category $\mathcal{I}$ of interfaces and a fibration

$$p : \mathcal{E} \longrightarrow \mathcal{I},$$

where the fibre $\mathcal{E}_I$ contains the theories appropriate to an interface I . Thus, for example,

$$T_{\text{Moore}(A,B)} \in \mathcal{E}_{(A,B)}.$$

A map of interfaces

$$f : I \longrightarrow J$$

should then induce a reindexing operation

$$f^* : \mathcal{E}_J \longrightarrow \mathcal{E}_I.$$

An n -ary wiring operation with input interfaces

$$I_1, \dots, I_n$$

and output interface J would therefore have two stages. First, the base wiring tells us how to transport the theories from the fibres over the I_i and J into a common context. Only after this reindexing do we form the runner or prism data inside a single fibre. Schematically, instead of writing immediately

$$U \rightsquigarrow T_1 + \dots + T_n + U',$$

one should expect something of the form

$$g^*U \rightsquigarrow f_1^*T_1 + \dots + f_n^*T_n + g'^*U',$$

where the maps f_i, g, g' are determined by the underlying typed wiring. Thus the dependent picture has three layers:

base wiring + reindexing of theories + runner semantics in a common fibre.
--

This is very close in spirit to the theory of fibred or dependent optics. There, the two sides of an optic may lie over different base objects, and the optic is assembled by reindexing them into a common fibre before applying the usual optic construction. The role of the base is precisely to record how the types or contexts vary. For our purposes, this suggests a dependent version of IntRun in which the theories on the wires are allowed to vary with the interfaces they inhabit.

Typed Moore machines

The simplest example is the parameterised Moore theory already discussed in Part III:

$$(A, B) \longmapsto T_{\text{Moore}(A,B)}.$$

Suppose we have several Moore components with interfaces

$$(A_1, B_1), \dots, (A_n, B_n),$$

and a wiring which assembles them into an interface

$$(A, B).$$

In an ordinary IntRun expression we might be tempted to place all of the corresponding theories directly into one formula. In the dependent picture, however, the wiring first transports each

$$T_{\text{Moore}(A_i, B_i)}$$

along the appropriate maps of input and output types. Only then do the reindexed theories enter the common prism or runner expression. This is a more faithful account of typed wiring: the change of interface is not external bookkeeping, but part of the operation itself.

State-dependent repertoires

There is a second reason to allow dependence which does not come merely from typed interfaces. The very theory of operations available to a system may depend on its state or context. Instead of one fixed theory T , imagine a family

$$s \longmapsto T_s.$$

A transition

$$s \longmapsto s'$$

may then move the system not only to a new state but also to a new repertoire of available operations:

$$T_s \longmapsto T_{s'}.$$

Ordinary $\text{Run}(\mathcal{D})$ keeps these two pieces separate: the theory is fixed, while the residual state varies. A dependent version would allow the theory itself to vary with the residual context. This can be useful whenever available behaviour changes with circumstance. For example, in a resource-sensitive system the admissible operations may depend on the current resource level; in a typed system they may depend on the current interface; in a control system they may depend on the current mode.

Resource-sensitive systems

The same idea connects naturally with the graded constructions considered in Part VI. If a cost-ordered monoid C indexes resource levels, one may imagine fibres

$$\mathcal{E}_c$$

of theories available at level $c \in C$. Running a computation changes the resource state

$$c \longrightarrow c',$$

and therefore also changes the fibre in which subsequent operations live. A dependent runner would then keep track simultaneously of

$$\text{state change,} \quad \text{resource change,} \quad \text{change of available theory.}$$

This is exactly the kind of situation in which a fibred formulation becomes more natural than forcing everything into one fixed ambient category.

Relation to DOTS

This dependent viewpoint also sharpens the relation with DOTS. The DOTS picture is naturally organised around typed interfaces and their wiring. From our point of view, those interfaces should form the base category $\mathcal{I}$. Over each interface sits a fibre of theories and systems. Thus one may picture the architecture as

$$\boxed{\text{DOTS-style wiring in the base} \quad \text{and} \quad \text{Run/IntRun semantics in the fibres.}}$$

The base says how the interfaces are connected and reindexed. The fibre says how the corresponding theories run on one another, how residual state is carried, and how handlers are transported. This suggests a progression

$$\text{Run}(\mathcal{D}) \longrightarrow \text{IntRun}(\mathcal{D}) \longrightarrow \text{FibIntRun}(\mathcal{E} \rightarrow \mathcal{I}),$$

where the last expression is only schematic notation for now. The first level describes running between fixed theories. The second adds multi-sided, prism-like wiring between pairs of theories. The third would allow the theories themselves to vary with typed interfaces, resources, states or other parameters. We shall not attempt a formal construction here. A genuine fibred theory would need suitable compatibility between reindexing and the monoidal or duoidal structure used to form runners and Sweedler homs. The important point for the present tutorial is conceptual: once theories are parameterised, the wiring of the parameters and the running of the theories should be treated together, rather than as two unrelated pieces of bookkeeping.