CHAPTER 2

PIEs – the simplest black-box model

2.1 Introduction

We are interested in formalisation of various properties of interactive systems. We want to state these properties over abstract models of interactive systems, in order to make them generic and to make the correspondence between the informal and formal statements as easy as possible.

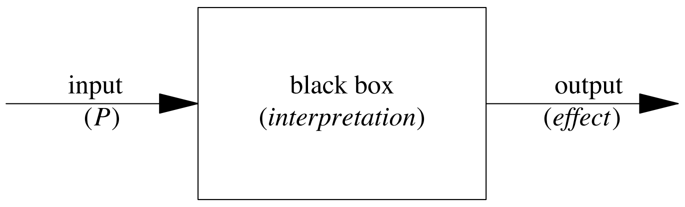
This chapter presents a very simple model called the PIE. This is probably one of the simplest models that can be defined, and yet many useful properties can be given some expression in terms of the model. Further, the more specific models given in the succeeding chapters will have very much the same flavour as PIEs and studying them will make these later models easier to understand.

2.2 Informal ideas behind PIEs

Although the PIE model is the simplest in terms of the number of elements in it, the treatment is probably the most complicated in this book. Many things are dealt with in detail here which have parallels in the more specific models developed later. When these issues arise later they can be skimmed over because of the experience gained in the PIE model. This first section gives an overview of the rest of the chapter so that the reader can pick up a good enough idea of the contents to read further chapters. The remaining, more formal, sections can thus be skipped on the first reading and returned to later.

2.2.1 PIEs

As we are interested in the human interface to computer systems, we want our models to refer only to what is visible (or audible, etc.) to the user. That is, we want a *black-box* model. The PIE model describes a system in terms of the possible inputs and the effect these have. Typically the input will consist of a sequence of commands (we will call the set of such allowable commands C, and the set of all such sequences the programs P). The relation between these inputs and the output is described using an *interpretation* function, what in control theory would be called a transfer function. The output we will call the *effect*:



The three together – inputs (P), interpretation (I) and effect (E) – give the model its acronym.

The effect is the most complex part of the model. We may wish to focus on the screen display, or some part of it. Perhaps the effect is some external activity, or it may be interpreted internally, perhaps the database or spreadsheet being manipulated. We deliberately leave this vague so that we may apply our analysis retrospectively to many different situations. However, when we define properties we will usually have some idea of what we mean by the effect in that circumstance, and we would not expect the same principles of usability to apply to all levels of analysis.

The input may similarly be applied at various levels. For instance, at one level we may wish to regard it as the physical keystrokes of the user, but at another we may want to think about the abstract application-level commands.

2.2.2 Properties of interest

Even this sparse model will be sufficient to give formal expressions of certain classes of principles. These same principles will arise in different forms throughout succeeding chapters:

- *Predictability* – Can we work out from the current effect what the effect of future commands will be? This can be thought of as the "gone for a cup of tea" problem. When you return and have forgotten exactly what you typed to get where you are, can you work out what to do next?

- *Observability* – Although the system is seen as a black box, the user can infer certain attributes of its internal state by external observation. How much can the user infer? How should the user go about examining the system? We will define the idea of a *strategy*: a plan that the user follows in order to uncover information about the system. Predictability can be viewed as a special case of observability when we are interested in observing the entire state as it affects subsequent interaction.
- *Reachability* – This is concerned with the basic functionality of the system. Is it possible to get to all configurations? Are there any blind alleys, where once you've entered some state there are other states permanently inaccessible? Related to the general notion of reachability, will be the specific case of *undo* and the problems associated with this.

2.2.3 Internal state and external behaviour

Although the primary definition of PIEs will make no reference to internal state, we will find it useful to distinguish a (not necessarily real) minimal state that can be inferred from external observation. It will be minimal in that it distinguishes fewer states than any other state representation that describes the same functionality. It is a very important concept, as it is often easier and clearer to give certain properties using the idea of state, and yet this would otherwise risk producing a definition that "knows too much" about the inside of the black box. We can always assume that such properties refer to this minimal state. Similarly, in later chapters when models are defined using state, these will be assumed to be minimal in the same way.

This operation of obtaining just enough information on the state we will call *monotone closure*, and given any view of a system V we will refer to the monotone closure of this as $V^\dagger$. This can be thought of as the part of the state which is just sufficient for predicting the future behaviour of V. The user cannot necessarily observe $V^\dagger$ directly, but if two systems differ in this respect there is some command sequence that the user can type which would expose the difference. Thus the user needs to know $V^\dagger$ in order to predict the behaviour of the system (and hence control it). Unfortunately, in general, the user would have to be able to try, and undo all possible command sequences to know for certain the state of $V^\dagger$. An example of this would be a pocket calculator. After having entered "3+4" the display would read "4". On its own the display is insufficient to predict what the effect of pressing the "=" key would be. The monotone closure of this view would include the pending operations and data.

The user can and does find out about the internal state of the system by experimenting with it. The simplest example of this is scrolling up and down in a word processor to see all of the document. By following such a strategy the user obtains a partial idea of the internal state. We call this the *observable effect*.

Obviously, different strategies are useful in different circumstances and yield different observable effects. The observability principles of the next chapter are framed largely in terms of the way the observable effect derived from the display tells us about the monotone closure of the result.

2.2.4 Exceptions and language

We will also consider some issues arising when the system as implemented differs from some "ideal" system, and what principles apply at these boundaries (the exceptions). We will consider two system models, one of the "ideal" system and one of the system as it actually is. The places where mismatch occurs are the exceptions. We will look at principles which demand that the user is made aware of exceptions (e.g. by a bell or message) and that the user is able to predict when exceptions will occur. Depending on the type of exception and the recovery principle used (see below), the designer may choose to apply the former or the latter (stronger) principle. These observability and predictability principles are supplemented by a discussion of recovery principles determining what sort of response the system should make to exceptions. The strongest is "no guesses", which asserts that when an exception occurs the system state is left unchanged. A weaker variant is "nothing silly please", which asserts that the system behaves in a way that the user *could* have achieved by non-exceptional commands.

This analysis of exceptions will lead naturally on to considering systems where the input language is for some reason restricted. This is not the normal "grammar" of the dialogue, which describes what the system *expects* of the user, but expresses some form of *physical* limitation of the user's input. The obvious example is a bank teller machine which covers the keypad with a perspex screen until the customer inserts a cash-card. The user is not free to perform all possible action sequences but, on the other hand, it may not be wise to assume that "illegal" sequences will not occur. The various principles need slight reformulations in the light of these properties.

2.2.5 Relationships between PIEs

We will briefly examine the ways in which we can relate PIE models to one another. This forms a basis for understanding the way that we can have several models of the same system, perhaps with different scopes or at different levels of abstraction. This will be taken up again in Chapter 7.

2.2.6 Health warning

The rest of this chapter will be quite "heavy", so as a reminder to the reluctant formalist, there are easier waters in subsequent chapters.

2.3 PIEs – formal definition

There are several ways we could use to define PIEs. The one we shall regard as basic, and probably the simplest, is as a triple $< P, I, E >$, as follows:

P – The set of sequences of commands from C (P stands for programs). More generally P can be a semi-group, which has little relevance for the user but can be useful occasionally in constructions (see §2.8).

E – The effects space, the set of all possible effects the system can have on the user. This may be thought of in different ways, and at different levels. For example, it may be regarded as the actual display seen by the user, or as the entire text of a document being edited, perhaps even the entire store of information available to the user.

I – The interpretation function, $P \rightarrow E$, representing all the computation done by the system.

We can represent a PIE as a diagram:

$$\begin{tikzcd} P \arrow[r, "I"] & E \end{tikzcd}$$

Later we will use the following class of functions. For any $p \in P$, we define I_p by:

$$I_p(q) \quad = \quad I(pq)$$

That is, I_p gives the functionality of the system as if when you get to it the command sequence p has already been entered. Obviously this will have relevance to the "gone away for a cup of tea" problem.

Alternative, equivalent formulations of the PIE model will be useful in different circumstances.

By state description and doit functions

Sometimes it is easier to describe a system as a state with transitions on each command. We will call the transition function *doit* and the set of states S:

$$doit : S \times C \rightarrow S$$

There will be, of course, some initial state $s_0 \in S$.

In general, the state will contain more information than we want to consider as the effect. In these case we will require an abstraction function *proj* yielding the effect:

$$proj : S \rightarrow E$$

That is, we need a sextuple $< C, S, E, s_0, doit, proj >$ in place of the triple!

We can then define a corresponding PIE with interpretation function I_{doit} as follows:

$$I_{doit}(p) = proj(doit^*(s_0, p))$$

where $doit^*$ is the iterate of *doit*:

$$\begin{array}{lcl} doit^*(s, null) & = & s \\ doit^*(s, p::c) & = & doit(doit^*(s, p), c) \end{array}$$

and "::" is the concatenation operator.

It is possible to reverse this and obtain a state representation for any PIE by simply taking $S = P$, and defining s_0, *doit* and *proj* thus:

$$\begin{array}{lcl} s_0 & = & null \\ doit(s, c) & = & s::c \\ proj & = & I \end{array}$$

However, taking the complete command history as state is a little excessive. In fact this is a maximal state representation: no other state representation can distinguish more reachable states. We will later, as promised, define a minimal (and much more useful) state representation, the *monotone closure*.

The fact that systems defined by *doit* functions can easily be related to PIEs, and hence have the various principles we will define over PIEs applied to them, is very important. The actual systems that I have specified have used *doit* functions as these are often the simplest way to describe interaction. In particular, the specification described in Chapter 11 uses just this mechanism. Also, in later chapters, we will use whichever representation is most convenient for the particular circumstances, and may even switch fluidly between the two representations.

By equivalence relations on *P*

A mathematically very simple (although not necessarily very meaningful to the user) way of defining an interactive system is by equivalence relations on P. Essentially, two command histories are equivalent if they have the same effect. That is, for any PIE $< P, I, E >$ we can define a relation $\equiv_I$:

$$p \equiv_I q \quad \hat{=} \quad I(p) = I(q)$$

Similarly, given such an equivalence, it defines (in some way) an interactive system and we can obtain a PIE $<P, I_\equiv, E_\equiv >$ from it:

$$\begin{array}{lcl} E_\equiv & \hat{=} & P / \equiv \\ I_\equiv(p) & \hat{=} & [\, p\,]_\equiv \end{array}$$

where $[\, p\,]_\equiv$ is the equivalence class of p in $P / \equiv$. These are inverses of one another, in that $\equiv_{I_\equiv}$ is the same as $\equiv$ and $I_{\equiv_I}$ is the same as I up to a one-to-one map on E. To prove the former:

$$\begin{array}{lcl} p \equiv_{I_\equiv} q & \hat{=} & I_\equiv(p) = I_\equiv(q) \\ & = & [\, p\,]_\equiv = [\, q\,]_\equiv \\ & = & p \equiv q \end{array}$$

The other way round is similar, except it is relative to the one-to-one map between $P / \equiv_I$ and E given by $[\, p\,]_{\equiv_I} \rightarrow I(p)$.

We will define other relations on P later which will be useful for defining observability properties.

By complete effect histories

We have given the basic definition of a PIE using an interpretation function relating complete command histories to the single effect they yield. On grounds of symmetry (and in analogy with transfer functions), we could have chosen to relate complete histories of commands to complete histories of effects. For example, for any interpretation I we can define a new interpretation I^* yielding the complete history of effects given by I:

$$I^* : P \rightarrow E^+$$

where E^+ is the set of non-empty sequences of effects from E, such that:

$$\begin{array}{l} I^*(c_0 c_1 c_2 \cdots c_n) \;\; = \\ \qquad (\, I(\mathit{null}), I(c_0), I(c_0 c_1), I(c_0 c_1 c_2), \ldots, I(c_0 c_1 c_2 \cdots c_n)\,) \end{array}$$

This form of definition is very similar to the use of streams to define interactive systems in lazy functional programming.

Any function generated thus satisfies some simple properties:

$\forall p, q \in P$

(i) $p \leq q \;\Rightarrow\; I^*(p) \leq I^*(q)$
(ii) $length(I(p)) = length(p) + 1$

Contrariwise, any such stream function can be turned into a PIE by taking the last element. Examining these conditions leads one to imagine generalisations of the PIE model based on such stream functions, but with a relaxation of one or other of these conditions:

(i) This is a necessary condition asserting that the function is temporally well behaved. If we drop this function we have systems that change their mind about effects already produced!

(ii) This is not so obviously necessary and it could fail to hold for one of two reasons (or both):

(ii_a) $\exists\, p, c$ **st** $I^*(pc) = I^*(p)$
(ii_b) $\exists\, p, c$ **st** $length(I^*(pc)) > length(I^*(p)) + 1$

(ii_a) This says that the new command has no additional effect. Note, this is not the same as saying the current effect is the same as the last one, which would be $I^*(pc) = I^*(p) :: last(I^*(p))$. How would you distinguish these at the interface? This would only seem useful as an abstraction from the true interface.

(ii_b) This has a more reasonable interpretation, namely that the system has some sort of dynamic behaviour after the command c. This (and ii_a) could be captured by using a more expressive effect space which represents within it the elements of dynamism: however, in doing this one might loose naturalness of expression.

Elements of this generalisation can be found in several places in this book. It could be seen as a half-way house between the PIEs and the fully temporal model considered in Chapter 5. Although we do not use the generalised form of stream function, we use the stream representation of PIEs when considering non-determinism in Chapter 6. Finally, there is some similarity to the model developed to describe status input in Chapter 10.

2.4 Some examples of PIEs

Although almost all interactive systems can be cast into the PIE framework, choosing appropriate examples is a little difficult. "Toy" examples are useful for getting to grips with nitty-gritty properties, but are of course unrealistic. (In fact, the toy examples may be parts or abstractions of more substantial systems.) If we look at realistic systems, we cannot expect to express the interpretation function very concisely or fully (if we did we would have a complete specification of the entire system). Bearing this tension in mind, we will look at a few examples.

2.4.1 Simple calculator

The first example we shall look at is a calculator that adds up single digits:

$$\begin{aligned} C &= \{\ 0, \ldots, 9\ \} \\ E &= \mathbb{N} \quad - \ \text{the natural numbers } \{\ 0, 1, \cdots, 57, 58, \cdots\ \} \end{aligned}$$

$$\begin{aligned} I(\ null\ ) &= 0 \\ I(\ pc\ ) &= I(\ p\ )\ +\ c \end{aligned}$$

The interpretation basically says:

- We start off with a running sum of zero.
- If at any stage we enter a new number (c) it gets added to the current running sum ($I(\ p\ )$).

To make the example a little more interesting we could add a clear key #. We augment C appropriately:

$$C_{\#} = \{\ 0, \ldots, 9, \#\ \}$$

The effect is the running total as before. The interpretation is built up recursively in a similar manner to I above:

$$\begin{aligned} I_{\#}(\ null\ ) &= 0 \\ I_{\#}(\ pc\ ) &= I(\ p\ )\ +\ c \qquad\qquad c \neq \# \\ I_{\#}(\ p\#\ ) &= 0 \end{aligned}$$

The only difference is the obvious one, that if a clear (#) is entered the running total is zeroed.

2.4.2 Calculator with memory

Now we will add a memory to the calculator. We will call the new commands *MS* and *MR* (memory store and memory recall). The command set is obvious:

$$C_m = \{\ 0, \ldots, 9, \#, MS, MR\ \}$$

Again the effect space is simply the natural numbers. The interpretation function is as before, but must "scan back" into the command history to find the last memory store, every time a memory recall is used:

$$\begin{array}{lcll}
I_m(\ null\ ) & = & 0 & \\
I_m(\ pc\ ) & = & I(\ p\ )\ +\ c & c\ \notin\ \{\ del,\ MS,\ MR\ \} \\
I_m(\ p\#\ ) & = & 0 & \\
I_m(\ p\ MS\ ) & = & I(\ p\ ) & \\
I_m(\ p\ MR\ ) & = & 0 & MS\ \notin\ p \\
I_m(\ p\ MS\ q\ MR\ ) & = & I(\ p\ ) & MS\ \notin\ q
\end{array}$$

Notice that the scanning back process makes the interpretation relatively complex. In fact, if readers are not familiar with such tricks they may well want to check it a few times to see that it really does what they would expect. The reason for the complexity is that the effect does not hold all the state information necessary for interpreting the next command. The appropriate information has to be gleaned from the command history.

We can produce an identical system using a state representation that explicitly includes the calculator's memory. We model the state as a pair of numbers, the first being the running total and the second the memory contents. The initial state is with both zero:

$$\begin{array}{lcl}
S & = & \mathbb{N} \times \mathbb{N} \\
s_0 & = & (\ 0, 0\ )
\end{array}$$

The projection function *proj* that extracts the effect is simply the first component:

$$proj(\ (\ e, m\ )\ ) \quad = \quad e$$

The *doit* function just does the obvious adding and swapping around between memory and running total:

$$\begin{array}{lcll}
doit(\ (\ e, m\ )\ , c\ ) & = & (\ e + c, m\ ) & c\ \notin\ \{\ \#, MS, MR\ \} \\
doit(\ (\ e, m\ )\ , \#\ ) & = & (\ 0, m\ ) & \\
doit(\ (\ e, m\ )\ , MS\ ) & = & (\ e, e\ ) & \\
doit(\ (\ e, m\ )\ , MR\ ) & = & (\ m, m\ ) &
\end{array}$$

2.4.3 Typewriter

From numbers to text. To simplify descriptions we will just consider simple typewriters and editors with no line-oriented commands. The whole text will be on a single line. Adding multiple lines just makes the descriptions rather longer. First the simple typewriter:

$$\begin{array}{l}
C\ =\ Chars\ =\ \{\text{a,b,c,d,e, ...,z,A,B, ..., etc. }\} \\
E\ =\ P\ =\ Chars^*
\end{array}$$

That is, the commands are the printable keys on the keyboard and the effect is a single sequence of these characters.

The effect of any key sequence is simply the sequence of keys hit:

$$I(\ p\ ) \quad = \quad p$$

or equivalently:

$$\begin{aligned} I(\ null\ ) \quad &= \quad null \\ I(\ pc\ ) \quad &= \quad I(\ p\ ) :: c \end{aligned}$$

The second recursive description can be used to extend the system, first to include a delete key. We will use ∇ for this:

$$C_\nabla \quad = \quad Chars \ + \ \nabla$$

The effect is as before:

$$E_\nabla \quad = \quad Chars^*$$

The effect is built up in a similar manner to the simple typewriter except that the delete key removes the last character:

$$\begin{array}{lll} I_\nabla(\ null\ ) & = \quad null & \\ I_\nabla(\ pc\ ) & = \quad I_\nabla(\ p\ ) :: c & c \ \in \ Chars \\ I_\nabla(\ p\nabla\ ) & = \quad null & \text{if } I_\nabla(\ p\ ) \text{ is empty} \\ I_\nabla(\ p\nabla\ ) & = \quad q & \text{if } I_\nabla(\ p\ ) \ = \ qc \end{array}$$

2.4.4 Editor with cursor movement

We can now add a cursor position with left and right movement. The command set has these additional two commands:

$$C_{cursor} \quad = \quad Chars \ + \ \nabla \ + \ LEFT \ + \ RIGHT$$

The effect we can regard as two sequences, those *before* and those *after* the cursor:

$$E \quad = \quad Chars^* \ \times \ Chars^*$$

This method of describing a cursor position is due to Sufrin (1982). Typing a character *C* from *Chars* simply adds text to the *before* half:

$$\begin{aligned} I_{cursor}(\ null\ ) \quad &= \quad \{\ null,\ null\ \} \\ I_{cursor}(\ pc\ ) \quad &= \quad \{\ before :: c,\ after\ \} \end{aligned}$$

where

$$\{\ before,\ after\ \} \quad = \quad I_{cursor}(\ p\ )$$

The delete ∇ operates on the *before* half in a similar manner to the typewriter above, so we will skip it and move on to the cursor commands:

$$I_{cursor}(\ p\ \mathit{LEFT}\ ) \quad = \quad \{\ before,\ c\!:\!after\ \}$$

where

$$\{\ before\!::\!c,\ \ after\ \} \quad = \quad I_{cursor}(\ p\ )$$

and

$$I_{cursor}(\ p\ \mathit{RIGHT}\ ) \quad = \quad \{\ before\!::\!c,\ after\ \}$$

where

$$\{\ before,\ \ c\!:\!after\ \} \quad = \quad I_{cursor}(\ p\ )$$

That is, the *LEFT* key moves characters from *before* to *after*, and *RIGHT vice versa*.

2.4.5 Editor with MARK

Finally (well almost finally), we have an editor with a marked position to which we can jump:

$$C_{mk} \quad = \quad Chars\ +\ \nabla\ +\ \mathit{LEFT}\ +\ \mathit{RIGHT}\ +\ \mathit{MARK}\ +\ \mathit{JUMP}$$

The effect space must be extended so that we can see where the mark is. We do this by just having the mark as a special character in the effect too:

$$E_{mk} \quad = \quad (\ Chars + \mathit{MARK}\ )^{*}\ \times\ (\ Chars + \mathit{MARK}\ )^{*}$$

This allows any number of marks, but we will arrange it so as there is only ever one. The interpretation of all the existing commands is just as before, and *MARK* is treated almost like any other character. We let delete remove a mark if it is the last "character" before it, and cursor movement go back and forth over it. However, to ensure that there is only one mark the adding of it must be slightly different:

$$I_{mk}(\ p\ \mathit{MARK}\ ) \quad = \quad \{\ strip(\ before\ )\!::\!\mathit{MARK},\ \ strip(\ after\ )\ \}$$

where $\{\ before,\ \ after\ \} \quad = \quad I_{mk}(\ p\ )$. We assume *strip* is a function that gets rid of any marks.

That only leaves us with the *JUMP* command:

$$\textbf{let} \quad \{\ \mathit{before},\ \mathit{after}\ \} \quad = \quad I_{mk}(\ p\ )$$

$$\begin{array}{lcl}
I_{mk}(\ p\ \mathit{JUMP}\ ) & = & \{\ b_1\ \mathit{MARK}\ ,\ b_2\ \mathit{after}\ \} \\
 & & \quad \textbf{if} \quad \mathit{before} \ = \ b_1\ \mathit{MARK}\ b_2 \\
I_{mk}(\ p\ \mathit{JUMP}\ ) & = & \{\ \mathit{before}\ a_1\ \mathit{MARK}\ ,\ a_2\ \} \\
 & & \quad \textbf{if} \quad \mathit{after} \ = \ a_1\ \mathit{MARK}\ a_2 \\
I_{mk}(\ p\ \mathit{JUMP}\ ) & = & \{\ \mathit{before},\ \mathit{after}\ \} \\
 & & \quad \textbf{otherwise}
\end{array}$$

These definitions simply move the cursor to just after the mark if there is one, or leave it where it is if there is none.

In Chapter 8 we will look more closely at positional information such as cursors and markers, and give a cleaner and more uniform treatment. However, these more detailed descriptions will still be able to be rendered as PIEs like the above.

2.4.6 A full wordprocessor

The last example will be the word processor I am using to write this. Unfortunately, if described in full it would take up the rest of the chapter, so we shall elide a few of the details.

First, let's say what are the commands and effect space:

$$\begin{array}{lcl}
C & = & \mathit{Keys} \quad – \ \text{the keys on my PC keyboard} \\
E & = & \{1 \cdots 25\} \times \{1 \cdots 80\} \ \rightarrow \ \mathit{Glyph}
\end{array}$$

Here *Glyph* is the set of characters, spaces and symbols that can appear on my screen. Basically, *E* is the set of all possible screen displays.

The interpretation function is quite simple:

$$I(\ p\ ) \ = \ \text{what I see when I have typed the keystrokes } p$$

2.5 Predictability and monotone closure

Now we come to the "gone away for a cup of tea" problem. Can we predict the future effect of commands from the current effect? It is likely that two different sequences of commands may yield the same effect, but if more commands are entered some difference comes to light. We then say that the original effect is *ambiguous,* as more than one internal state is possible:

$$ambiguous(\, e \,) \quad \hat{=} \quad \exists \; p, q, r \in P$$
$$\textbf{st} \quad I(p) = e = I(\, q \,) \;\; \textbf{and} \;\; I(\, pr \,) \neq I(\, qr \,)$$

If we look at the examples in the previous section, the calculator with memory is always ambiguous, since from the effect, we only can tell the running total. Thus, if we have a current effect of "57" and then use the memory recall command, we may get any effect depending on the precise history that led to the running total of "57". Similarly, the word processor I am using is ambiguous (when the effect is regarded as the screen image), as a scroll up key will reveal text that cannot be inferred from the current screen.

The other examples, such as the simple calculator and all the simple editors, have no ambiguous effects at all. If no effect is ambiguous then things that look the same are the same. We call a PIE that has this property *monotone:*

monotone:
$$\forall p, q, r \in P \quad I(\, p \,) \; = \; I(\, q \,) \;\; \Rightarrow \;\; I(\, pr \,) = I(\, qr \,)$$

Or alternatively, in terms of I_p:

$$\forall p, q \in P \quad I_p(\, null \,) \; = \; I_q(\, null \,) \;\; \Rightarrow \;\; I_p \; = \; I_q$$

Is this a useful property, and does it say what we wanted it to say? What it says is that the entire future effect can be predicted from the present one. In other words, the current effect could be regarded as a state, and in fact, any monotone PIE can be represented with a *doit* without a corresponding *proj* function.

If we think about an actual computer system (like my word processor) and take the effect to be the actual screen display, this monotone property is far too strong: we know that the screen could not possibly hold sufficient information for that. We don't even want it to. We want to view only relevant parts of what we are manipulating. Clearly, if this property is to be useful it should be applied to an effect which is not the underlying machine state (when it becomes tautologous) and not the display itself, but something "just underneath" the display, a widening of the display view. In the next section we will look at just such a widening.

We can give a very simple statement of monotonicity using the monotone equivalence $\equiv^{\dagger}$ defined by:

$$p \;\equiv^{\dagger}\; q \quad \hat{=} \quad \forall r \in P \quad I(\, pr \,) \; = \; I(\, qr \,)$$

That is, p and q are monotone equivalent if they have the same interpretation and whatever commands are entered afterwards they both yield the same effects. Monotonicity then becomes:

monotone:
$\forall p, q \in P \quad p \equiv q \;\Rightarrow\; p \equiv^{\dagger} q$

We can use $\equiv^{\dagger}$ to define a PIE using the construction defined in §2.3. This PIE is clearly monotone and can therefore be described completely using state transitions. Further, we can define a projection function from $E_{\equiv^{\dagger}}$ to E by $[\, p \,]_{\equiv^{\dagger}} \rightarrow I(\, p \,)$. This is a function, as clearly $p \equiv^{\dagger} q \;\Rightarrow\; I(\, p \,) = I(\, q \,)$.

This interpretation and projection together give us the *monotone closure,* the minimal state representation of the PIE. We will call the resulting interpretation $I^{\dagger}$, the effect space $E^{\dagger}$ and the projection $proj^{\dagger}$.

If it is preferred, we can give an (up to isomorphism) equivalent definition without the use of relations:

$$E^{\dagger} \quad \subset \quad P \rightarrow E$$

$$I^{\dagger}(\, p \,) \quad = \quad I_p$$

The monotone closure is minimal in that given any monotone (state-based) PIE $< P, I', E' >$ and projection $proj'$ such that $I \;=\; proj^{\dagger} \circ I^{\dagger}$, we can define a projection $proj''$ from E' to $E^{\dagger}$ so that the following diagram commutes:

$$\begin{tikzcd}
 & E' \arrow[dr, "proj'"] \arrow[dd, "proj''"] & \\
P \arrow[ur, "I'"] \arrow[dr, "I^{\dagger}"'] & & E \\
 & E^{\dagger} \arrow[ur, "proj^{\dagger}"'] &
\end{tikzcd}$$

Another way we can view this minimality is that any distinctions that are made between states in $E^{\dagger}$ are implied directly by the possible observations. In that sense it has no implementation bias. (Jones 1980) This is important because it means that we can freely assume the existence of a state-transition implementation of a PIE without sacrificing our surface philosophy, as the states within PIE† are directly observable from the interface.

It should be noted that this is intended as a theoretical construction, useful for the statement of principles and for defining terms, but not for the implementation of systems. The minimal state representation need not be realisable in practice, as even when I is computable and we can thus for any p represent the function $I^{\dagger}(\, p \,)$, we cannot in general decide whether two such functions are identical.

Having said that, however, the construction can be useful for implementation.

If the PIE represents a finite automaton then $I^\dagger$ is computable and this gives us a construction for the finite state automaton, with minimal number of states satisfying the behavioural specification given by the PIE.

Even when the PIE is not a finite automaton the monotone equivalence is sometimes useful. In many cases it turns out to be computable and it is possible to label the equivalence classes giving a constructed minimal state representation. Alternatively, the attempt to compute the monotone equivalence may suggest a non-minimal, but still good, state representation.

As an example, if we look at the calculator with memory, the monotone closure of the interpretation function yields precisely the state representation that we gave. The monotone closure of my wordprocessor is much more complex, including at least all the text off-display, an invisible marked location, find/replace buffers, a copy of the last deleted item, and a ruler with tab positions.

2.6 Observability and user strategy

In the last section, we said that we will often want the effect itself to be something that is not predictable, yet there is another wider effect just beneath the surface of what is seen directly and which can be inferred from the immediate defect. For example, when using a display editor, there is (say) a 25-line window which is visible on the screen, yet one conceptually "sees" an arbitrarily large text just beneath this. Because the process of discovering this underlying text from the display is relatively easy, it will for most purposes be safe to ask for the PIE defining the underlying text to be predictable.

How do we define this wider view? One answer has already been suggested in the introductory chapter, namely as the information contained in all possible subviews. However, we also saw there that such a definition is probably too strong: aliasing prevents it from working. A different way to approach the problem is to examine what a typical user might do if he wanted to know the entire text of a document. The scenario might be a bit like this:

1. Note where I am.
2. Use the TOP button to get to the top of the document.
3. Use the SCROLL DOWN button to move down page by page.
4. When I get to the end, use the SCROLL UP button to get back to where I started.

Depending on the functionality of the editor there may be more complex "short cuts" to speed up this process, but the essential feature is that the user has a sort of program or algorithm which he follows which "covers" the entire text. Further, when bringing these snippets of information together, the relative positions are known because of the strategy followed. So in the example in the Introduction, one could tell "«1011»" from "«1101»" because in the former the "11" window would come after the "10" window and in the latter it would precede it. We will use this idea to extend the concept of observability. We will call such a user algorithm a *strategy,* and the resulting wider view the *observable effect*.

One possible formulation of the requirement for strategies is as follows. If E^+ is the space of non-empty sequences of elements of E, we define a strategy as a function $s : E^+ \rightarrow P$. To use this function starting at a given initial effect e after a sequence of commands p, we define $q_1 \cdots$, $p_0 \cdots$ and $e_0 \cdots$ as follows:

$$\begin{array}{lcl} p_0 & = & p \\ p_{n+1} & = & p_n \; q_{n+1} \\ e_n & = & I(\; p_n \;) \\ q_{n+1} & = & s \; (\; e_0, e_1, e_{2,} \cdots, e_n \;) \end{array}$$

That is, from all the effects so far in our use of the strategy we work out a new command sequence q_n using s which is then issued resulting in a new set of commands to update p_n and a new effect e_n. We would demand that the strategy s actually comes to an end sometime and this is represented by $q_n = null$, the empty sequence.

We can distinguish between two states of the system generated by p and p' if for some i, $e_i \neq e'_i$ and the observable effect – "what you see" in the wide sense – is precisely the equivalence classes generated by this. That is, we can define an equivalence relation using the strategy $\equiv_s$ by:

$$p \equiv_s p' \quad \hat{=} \quad \forall i \quad e_i = e'_i$$

The observable effect O_s is then precisely $P/\equiv_s$ and we have the corresponding interpretation function I_s, derived in the standard manner. Note also that $e_0 = I(\; p_0 \;) = I(\; p \;)$: thus $p \equiv_s q$ implies $p \equiv q$, and so $\equiv_s$ is stronger than $\equiv$. This means that there is a natural projection $proj_s$ from O_s to E factoring I, $(I = proj_s \circ I_s)$:

$$\begin{tikzcd} P \arrow[r, "I_s"] & O_s \arrow[r, "proj_s"] & E \end{tikzcd}$$

On the other hand:

$$\begin{aligned} p \equiv^{\dagger} p' \quad &\Rightarrow \quad \forall q \quad p \leq q \;\Rightarrow\; I(pq) = I(p'q) \\ &\Rightarrow \quad \forall i \quad I(p_i) = I(p'_i) \\ &\Rightarrow \quad \forall i \quad e_i = e'_i \\ &\Rightarrow \quad p \equiv_s p' \end{aligned}$$

That is, $\equiv^{\dagger}$ is stronger than $\equiv_s$, and thus there is a projection $proj_O{}^{\dagger}$ from the monotone closure to O_s factoring I_s, ($I_s = proj_O{}^{\dagger} \circ I^{\dagger}$):

$$\begin{tikzcd} P \arrow[r, "I^{\dagger}"] & E^{\dagger} \arrow[r, "proj_O{}^{\dagger}"] & O_s \arrow[r, "proj_s"] & E \end{tikzcd}$$

It may be that there is a strategy such that the projection from the monotone closure is one-to-one, or in other words so that I_s is monotone. In this case the strategy has revealed sufficient information to predict all future effects. We will say that such a strategy *tames I* and if such a strategy exists, we will call the original PIE *tameable*.

Later we shall see a PIE which is not tameable: that is, how ever cleverly one devised a strategy, there will always be states that are indistinguishable using the strategy, but which eventually may differ given certain commands. We will prove this by using the following constructions and lemma.

For any p, p' and set $Q \subset P$ we say Q distinguishes p and p', if adjoining some sequence to p and p' yields a different effect:

$$Q \text{ distinguishes } p, p' \quad \hat{=} \quad \exists\, q \in Q \quad \textbf{st} \quad I(pq) \neq I(p'q)$$

We will say that a PIE is *grotty* if there is some p such that given any finite Q there is some p' which cannot be distinguished from p and yet is not equivalent to it using the monotone equivalence That is:

grotty:

$$\begin{aligned} &\exists\, p \in P \quad \textbf{st} \quad \forall\, Q \subset P \quad Q \textit{ finite} \;\Rightarrow \\ &\qquad \exists\, p' \quad \textbf{st} \quad \forall\, q \in Q \quad I(pq) = I(p'q) \;\; \textbf{and not} \;\; p \equiv^{\dagger} p' \end{aligned}$$

Note especially the order of the quantifiers: the indistinguishable element p' depends on the distinguishing set Q.

The lemma is that any grotty PIE cannot be tameable.

LEMMA: grotty $\Rightarrow$ **not** tameable.

We shall prove this by showing that any strategy we may choose does not tame the PIE.

PROOF:

Let p be the element which has the indistinguishability property, and assume the strategy is s. Define Q as follows, using the sequence q_i defined above.

$$Q \;\hat{=}\; \{\, null,\; q_1,\; q_1 q_2,\; q_1 q_2 q_3, \cdots \}$$

This set is finite because the strategy terminates, and thus the q_is are eventually null. Therefore, because the PIE is grotty, there must be some p' not monotone equivalent to p which cannot be distinguished using Q, that is:

$$\forall\; i \in \{0, ...\} \quad I(\, p\; q_1 q_2 \cdots q_i \,) \;=\; I(\, p'\; q_1 q_2 \cdots q_i \,)$$

But this means that $e'_0 \;=\; e_0$ and hence $q'_1 \;=\; q_1$, and then by induction $e'_n \;=\; e_n$ and $q'_n \;=\; q_n$ for all n. Thus:

$$\forall\; i \in \{0, ...\} \quad I(\, p\; q_1 q_2 \cdots q_i \,) \;=\; I(\, p'\; q'_1 q'_2 \cdots q'_i \,)$$

That is (by definition) $p \;\equiv_s\; p'$.

Hence we have a pair p and p' which are not monotone equivalent and yet $p \;\equiv_s\; p'$, so that the projection $proj_O{}^{\dagger}$ is not one-to-one. However, this argument was for an arbitrary strategy and hence there is no strategy that tames I and it is not tameable.

All the arguments and definitions in this section have been for an arbitrary strategy s, but often the strategy of interest is clear by context, and in this case we will drop the suffix s, for instance using O instead of O_s.

2.7 Reachability

Reachability properties are about what can be done with a system, and whether there are states one can get to from which other states are inaccessible. The simplest reachability condition is to demand that I is surjective. If the set E describes the intended set of effects then it will often be a basic requirement of the system that all such effects can be obtained by some sequence of commands. The surjectivity of I ensures this. We will call this condition *simple reachability*:

simple reachability:
I is surjective

Of course this is not enough: although we may originally be able to reach any given effect, we may well be able to get ourselves up a "blind alley", forever after being unable to obtain a desired effect. To make a stronger statement we must look at the situation when we have entered some sequence of commands p; I_p (defined in §2.3) must still be surjective. We will call this stronger condition *strong reachability*:

strong reachability:
$\forall\ p \quad I_p$ is surjective

That is, "you can get anywhere from anywhere".

There are equivalent definitions using I or $\equiv$ alone and assuming simple reachability:

$$\forall\ p, q \in P \quad \exists\ r \in P \quad \mathbf{st} \quad I(\ pr\ )\ =\ I(\ q\ )$$

$$\forall\ p, q \in P \quad \exists\ r \in P \quad \mathbf{st} \quad pr\ \equiv\ q$$

There is a third, yet stronger, reachability condition we may want to impose. Strong reachability says that any desired effect can be obtained after any initial command history: however, this ignores the "hidden" state that may manifest itself latter. We may want to say that we can get to anywhere in the stronger sense that we cannot distinguish the way we got there by later observation. This obviously refers to the monotone closure and we can express it as:

megareachability:
$\forall\ p \quad I^{\dagger}{}_p$ is surjective

Again there are equivalent definitions using $I^{\dagger}$ or $\equiv^{\dagger}$ alone, assuming simple reachability:

$$\forall\ p, q \in P \quad \exists\ r \in P \quad \mathbf{st} \quad I^{\dagger}(\ pr\ )\ =\ I^{\dagger}(\ q\ )$$

$$\forall\ p, q \in P \quad \exists\ r \in P \quad \mathbf{st} \quad pr\ \equiv^{\dagger}\ q$$

2.8 Undoing errors

2.8.1 Importance and problems of undo mechanisms

The ability to spot errors quickly is one of the advantages of highly interactive computing. But this ability is only appealing if the correction is just as easy! Many systems supply some form of undo facility, either by direct command or as a side effect of the form of the system. Shneiderman (1982) suggests that the ability to easily undo one's actions incrementally is one of the hallmarks of a direct manipulation system.

Undo mechanisms can get quite sophisticated, ranging from the simple restoration of delete buffers to the keeping of entire session history trees. (Vitter 1984) It is important to realise that there are fundamental incompatibilities between the various possible refinements. In practice, by ignoring these incompatibilities, designers may fall into several traps.

Any editor that retains deleted line buffers or complete copies of the last editor state is too inflexible to deal with the more general types of mistake, only being able to recover back one or at most a few steps. But any editor that retains a complete command and editor state history for the purposes of recovery has further problems:

- *Observability* – The behaviour of the editor is obviously determined by the state of the current history. So if we want to preserve monotonicity this history must be part of the observable effect: that is, there must be commands for perusing this structure (in fact, these commands are often included in the undo procedure itself).
- *Reachability* – Supposing the command and state history exists in some form and is observable, if the editor as a whole satisfies the reachability properties, there must be some way for the user to edit this structure since it is part of the editor's state. This is usually the point at which an undo system would call a halt, only asserting reachability for the unadorned system. On the other hand, if this editing is allowed...
- *Undoability* – Are the relevant commands themselves undoable? If so we start quickly to chase our tail!

2.8.2 Examples

To illustrate some of these problems, we consider two simple undo editors. Both are based on the simple typewriter but could easily have been defined generically over any PIE. They both seem quite reasonable at first glance, but they are lacking in either functionality or in observability.

First, a basic one-step undo:

$$\begin{aligned} C_1 &= \{ a, b, c, \cdots, z, \# \} \\ E_1 &= \{ a, b, c, \cdots, z \} \end{aligned}$$

$$\begin{array}{lcll} I_1(\ null\ ) & = & null & \\ I_1(\ \#\ ) & = & null & \\ I_1(\ p::c\ ) & = & I_1(\ p\ )::c & c \neq \# \\ I_1(\ p::c::\#\ ) & = & I_1(\ p\ ) & \forall\ c \end{array}$$

This is the sort of undo mechanism that is used in editors like vi: it essentially retains two "states" and flips between them. It is not strong reachable, and requires an additional delete mechanism to be so. Neither is it predictable, as it is not possible to tell from the current display what a future undo (#) will result in:

$$I_1(\ null\ ) \ = \ null \ = \ I_1(\ a\#\ )$$

but

$$I_1(\ \#\ ) \ = \ null \ \neq \ a \ = \ I_1(\ a\#\#\ )$$

It is, however, tameable, using the strategy "type '#' twice". We can prove this by defining a state transition function *doit* with associated projection *proj* equivalent to I_1 and then showing that the strategy can observe this state:

$$S \ = \ E_1 \times E_1$$

$$s_0 \ = \ (\ null,\ null\ )$$

$$\begin{array}{lcll} doit(\ c, (\ e_1, e_2\ )\ ) & = & (\ e_2, e_2{::}\,c\ ) & c \neq \# \\ doit(\ \#, (\ e_1, e_2\ )\ ) & = & (\ e_2, e_1\ ) & \end{array}$$

$$proj(\ (\ e_1, e_2\ )\ ) \ = \ e_2$$

One can quickly verify that $proj \circ doit(\ .\ , s_0\ )$ satisfies the conditions for I_1. Further, it is clear from it that the sequence "# then # again" followed no matter what the effect is, will give rise to the two effects e_1 and e_2.

One might be tempted to design a more elaborate and powerful undo system, where an indefinite number of commands could be undone:

$$\begin{array}{lcl} C_2 & = & \{\ a, b, c, \cdots, z, \#_1, \#_2, \cdots\ \} \\ E_2 & = & \{\ a, b, c, \cdots, z\ \} \end{array}$$

$$\begin{array}{lcll} I_2(\ null\ ) & = & null & \\ I_2(\ \#_n\ ) & = & null & \\ I_2(\ p{::}\,c\ ) & = & I_2(\ p\ ){::}\,c & c \neq \# \\ I_2(\ p{::}\,c{::}\,\#_n\ ) & = & I_2(\ p{::}\,\#_{n-1}\ ) & \forall\ c \end{array}$$

That is, $\#_n$ is the n-step undo which gets one back to the effect one had n commands ago; however, this is not exactly the same state as measured by the monotone closure. This new undo editor is in fact strong reachable, but not mega reachable. It is also not only, not predictable, but grotty!

The minimal state representation is in fact huge:

$$S \ \subset \ E_2^{\infty-}$$

where $E_2^{\infty-}$ is the set of sequences from E_2 semi-infinite to the left (in fact, the sequences generated will always have only a finite number of non-null entries) and:

$$s_0 \;=\; null^{\infty-} \;=\; (\,\cdots, null, null, null, null, null, null\,)$$

$$\begin{array}{lcll} doit(\,c, s{::}e\,) & = & s :: e :: (\,e{::}c\,) & \quad c \neq \# \\ doit(\,\#_n, s\,) & = & s :\; s[\,-n\,] & \end{array}$$

$$proj(\,s{::}e\,) \;=\; e$$

where $s[\,-n\,]$ represents the nth element of s from its end.

This state representation is clearly minimal, since any two states s_1 and s_2 can be distinguished by repeating the command "$\#_{len}$", *len* times, where *len* is the number of elements from the ends of the two sequences we need to go before all preceding elements are null. (This must be finite by simple examination of *doit*.)

This makes the non-megareachability obvious. If we entered a then b, we could never get from the latter state to the former, as the *doit* function only adds to the end of the state sequence and thus the state would always have ab as its first two non-null elements.

We can also now prove that I_2 is grotty. Take the command sequence a and any finite set of command sequences Q. Each command in Q may have some ordinary commands and some commands of the form $\#_n$. Let m be the maximum of all these ns over all the elements of Q. Finally, take p' to be the command history "$a\#_1\#_2\cdots\#_m a$" and p to be simply "a". p and p' will yield the states:

$$null^{\infty-}\; a \;=\; (\,..\, null, null, null, a\,)$$

and

$$null^{\infty-}\; a\; null^m\; a$$

The *doit* function always adds to the end of the state, and the commands within any element of Q can at most look m from the end of the state on which they act, and hence at most m from the original state: hence the two states generated by p and p' will yield exactly the same effects for each element of Q. Yet p and p' are not monotone equivalent by the minimality of S. Thus we have shown that for any finite Q we can construct a p' such that Q does not distinguish them, but which is not monotone equivalent to p ($\#_{m+2}$ distinguishes them). Hence I_2 is grotty.

2.8.3 A simple definition of undo

As quite reasonable and simple undo mechanisms seem to have quite complex semantic problems, can we progress further by framing some sort of formal definition of undo? A weak form of error correction has already been described in the reachability conditions. These tell us that we can get anywhere from anywhere, so in particular, if we do something wrong we can get back to the position before the mistake. However, the definitions of the reachability conditions involve the entire program. It is more attractive if error recovery is incremental, depending only on mistakenly entered commands, because typically these are most recent and are short. Imagine a user telephoning an advisor with the request "I've just done so-and-so and now I seem to be stuck; how can I get out of it?": a helpful reply should exist and not depend on any other information. Formally we could write this:

$$\begin{aligned} &\exists \;\; undo \in P \rightarrow P \\ &\qquad \textbf{st} \;\; \forall \;\; p, r, s \in P \quad I(\, r\; p\; undo(\, p\,)\, s) \;=\; I(\, r\; s\,) \end{aligned}$$

This appears to be not as strong as one might want: for instance, one might really want the undo to always be the same command. However, it turns out to be far more restrictive than is apparent. To see this we will define the *strong equivalence,* and restate the simple undo condition using it.

2.8.4 Strong equivalence

We have already used a fairly strong equivalence $\equiv^{\dagger}$. Regarding P as a semigroup, one would call such an equivalence a *right congruence*, as equivalence is preserved by adding commands to the right. This instantly suggests investigating the full congruence, where equivalence is preserved in all contexts. We will call this strong equivalence ($\sim$):

$$p \sim q \quad \hat{=} \quad \forall r, s \in P \quad rps \;\equiv\; rqs$$

That is, two command sequences are strong equivalent if they have exactly the same effect, no matter the context in which they are used (ignoring the temporary effects while they are being invoked). If the PIE represents a finite-state automaton, then the equivalences $\equiv^{\dagger}$ and $\sim$ are called Nerode and Myhill equivalence respectively. (Arbib 1969)

We can construct the semigroup $P^{\sim}$ as the quotient of P by the congruence $\sim$, $P^{\sim} = P/\sim$. We can then define the map $I^{\sim} : P^{\sim} \rightarrow E$ by $I^{\sim}([p]) = I(\,p\,)$. This is a well-defined function because $p \sim q \Rightarrow I(\,p\,) = I(\,q\,)$. We can then factor I using the canonical semigroup homomorphism $parse^{\sim} : P \rightarrow P/\sim$:

$$\begin{tikzcd}
P \arrow[rr, "I"] \arrow[dr, "parse^{\sim}"'] & & E \\
& P^{\sim} \arrow[ur, "I^{\sim}"'] &
\end{tikzcd}$$

This semigroup is in fact minimal for expressing $<P, I, E>$ in the sense that for any $<P', I', E'>$ and semigroup homomorphism $parse' : P \rightarrow P'$ such that $I = I' \circ parse'$, we can construct $parse'' : P' \rightarrow P^{\sim}$ so that the following diagram commutes:

$$\begin{tikzcd}
& P' \arrow[dd, "parse''"] \arrow[dr, "I'"] & \\
P \arrow[ur, "parse'"] \arrow[dr, "parse^{\sim}"'] & & E \\
& P^{\sim} \arrow[ur, "I^{\sim}"'] &
\end{tikzcd}$$

Combining this with the monotone closure, we have the following decomposition for any PIE:

$$\begin{tikzcd}
P \arrow[r, "parse^{\sim}"] & P^{\sim} \arrow[r, "I^{\sim\dagger}"] & E^{\dagger} \arrow[r, "proj^{\dagger}"] & E
\end{tikzcd}$$

where $P^{\sim}$ is "as far to the right" as possible for a semigroup, and $E^{\dagger}$ is "as far to the left" as possible for a state.

Again, we have no guarantee that $P^{\sim}$ is computable, as $\sim$ is in general not decidable.

2.8.5 Undo and group properties

Using this strong equivalence, we can restate the undo condition as:

simple undo condition:
$\exists \; undo \in P \rightarrow P$
st $\forall \; p \in P \quad p \; undo(\, p\,) \sim null$

This is in fact a way of saying that $P/\sim$ is a group. The resultant class of objects has been studied in the theory of formal languages. (Ansimov 1975) One consequence of this is that not only would there exist an undo for each command, but that they would all be different! Thus knowing only that there exists a way of undoing any action is of little comfort; we must also ask how easy it is to remember and to perform.

2.8.6 Undo for a stratified command set

A more realistic requirement is an editor with both ordinary commands, and a small distinguished set of undo commands. Ordinary commands obey much more stringent rules than undo commands, under the assumption that the latter are used more cogently. This distinction could be expanded to other classes of special commands.

Intuitively the class of ordinary commands includes some commands which have undos (e.g. letters being undone with delete), some commands that do behave like subgroups (e.g. cursor movement), and others like delete itself with no undo; the special undo commands would then be added. However, it is worth noting that in general even if some commands like cursor right/left have inverses relative to the ordinary commands, they do not do so once the new undo commands are added.

Formally, we have two alphabets C and U. We then have P, the sequences from C, and P_u, the set of sequences from $C + U$, and the following equivalent conditions:

$\exists \; undo \in P \rightarrow P_u$
st $\forall \; p, q, r \in P \quad I(\, p\, q \; undo(\, q\,)\, r) \; = \; I(\, p\, r\,)$

$\exists \; undo \in C \rightarrow C + U$
st $\forall \; p, r \in P, c \in C \quad I(\, p\, c \; undo(\, c\,)\, r) \; = \; I(\, p\, r\,)$

2.9 Exceptions and language

When designing a concrete system, we often begin by designing an idealised system, such as an editor with an unbounded text length or a graphics device with arbitrary resolution. However, when turning this ideal into a running system various boundary conditions arise. The user of a system may also have an idealised model of the system; for instance, "UP-LINE followed by DOWN-LINE gets me back where I started" (*cf.* undo above). This too may fail at boundaries. For a system designed with user engineering in the foreground the two ideal views should correspond. Given that some properties of the ideal system fail in the actual system (if nothing else, the commands that hit the boundary condition would behave inconsistently), we would expect that when an exception does arise there is some sort of consistent system action. That the system warn the user of such an exception by some means – bell, flashing light, etc. – goes without saying!

2.9.1 Modelling exceptions

Suppose then that we have two PIEs $< P, I, E >$, the ideal, and $< P, I_{ex}, E_{ex} >$, the exception PIE. They are related by a map *proj* from E_{ex} to E. There is also a boolean function *ex* on the elements of P, that satisfies $ex(\ p\ ) \ \Rightarrow\ ex(\ pq\ )$ for all q. This means exceptions don't go away – not as bad as it seems! The following diagram commutes on the set $Ok \ = \ \{ \ p \in P \ \mid \ \textbf{not}\ ex(\ p\ ) \}$:

$$\begin{tikzcd}
 & E \\
P \arrow[ur, "I"] \arrow[dr, "I_{ex}"'] & \\
 & E_{ex} \arrow[uu, "proj"']
\end{tikzcd}$$

That is, $I(p) \ = \ proj(\ I_{ex}(\ p\ ))$ unless $ex(\ p\ )$.

Note that in general the map *proj* need be neither surjective nor injective. The exception effects may have additions such as error status lines, which leads to *proj* taking several members of E_{ex} to one of E. Also there may be some effects in E that can never be reached by $proj \circ I_{ex}$ because exceptions block all paths.

For instance, if the exception represents a bounded version of a text editor, then elements of the effect corresponding to large texts may never occur.

2.9.2 Detecting exceptions

Clearly we should be able both to detect when an exception has occurred, and ideally be able to predict whether an exception will occur before submitting the command that raises the exception. We can state this formally by requiring two user decision functions, *is_ex*? and *will_be_ex*?. The former can tell from an effect whether an exception has occurred, and the latter can tell from the current effect whether a particular command will cause an exception:

$$is\_ex? : E \rightarrow Bool$$
$$will\_be\_ex? : C \times E \rightarrow Bool$$

$$is\_ex? \circ I_{ex} \quad = \quad ex$$
$$\forall p \in P, c \in C \qquad ex(\ pc\ ) = will\_be\_ex(\ c, I_{ex}(\ p\ )\ )$$

We could equally well have been more positive and defined the "ok" operators *is_ok*? = **not** *is_ex*? and *will_be_ok*? = **not** *will_be_ex*?: the two formulations would be logically (but perhaps not psychologically) identical. Also one should note that although these are expressed as user decision functions their importance lies also in the constraints they put on the system.

The latter, prediction, information may be hard to achieve, and in case of doubt one should of course aim for safety, warning the user against exceptional conditions if one is unsure, especially if the consequences are major. This corresponds to a weakening of the condition for this to an implication:

$$\forall p \in P, c \in C \qquad ex(\ pc\ ) \Rightarrow will\_be\_ex(\ c, I_{ex}(\ p\ )\ )$$

In Chapter 5, we use similar decision functions for detecting steady state and when commands will be ignored. These can be thought of as specific cases for particular exception conditions.

2.9.3 Exception recovery principles

Although we've said that the exception effect may, and should, contain error indicators, we will assume that we are dealing with an abstraction of the full effect which does not include this extra information for the statement of the following principles. These are concerned with the possible error recovery rules after an exception has occurred.

One likely design principle for exceptions is "no guesses please". That is, when a command causes an exception to be raised, subsequent commands behave as if the command had never been issued. Formally:

$$\forall \; p, q \in P, c \in C$$
$$\textbf{not } ex(\, p\,) \textbf{ and } ex(pc) \;\Rightarrow\; I_{ex}{}^{\dagger}(\, pc\,) \;=\; I_{ex}{}^{\dagger}(\, p\,)$$

Or there is a weaker condition, "nothing silly please":

$$\forall \; p, q \in P, c \in C$$
$$\textbf{not } ex(\, p\,) \textbf{ and } ex(pc) \;\Rightarrow$$
$$\exists \; p' \;\; \textbf{st} \;\; \textbf{not } ex(\, p'\,) \;\textbf{and}\; I_{ex}{}^{\dagger}(\, pc\,) \;=\; I_{ex}{}^{\dagger}(\, p'\,)$$

This just says that when an exception is raised, at least the position the user is left in is one that could have been reached by legitimate means.

The reason for not including the error signalling in these principles is that clearly the effect after an exception would be different, as the error message would be there. One can clearly extend the rules to cover the more complex case: for instance, one can ask that all but the immediate effect is identical when considering the additional information. This would correspond to asking that all error messages be cleared after the next (correct) response, rather than leaving them there until the next error. There are many permutations of this ilk, and the above simplifications give the flavour of possible exception conditions.

A tentative basis on which to choose between the alternative requirements is that "nothing silly" is appropriate where the exception is associated with a limit of the system, but "no guesses" should apply where the user has made an error. For example, line break algorithms for most editors follow "nothing silly", but cuts without previous marks follow "no guesses" – not deleting the entire document up to the cut point, say! We can also reformulate our request for undos, only asking for an undo when an exception has not occurred. (This is safe if the "no guesses" condition is used.) For instance, up-line/down-line undo one another *except* at the top/bottom of the document.

2.9.4 Languages

An exception occurs when a user does something that we wish he hadn't. There is the stronger restriction on input arising from what is *possible*. That is, at the level we are designing the PIE it may be impossible for certain command sequences to be generated. This may occur in two ways:

- There may be some *physical* constraint. For instance, a cash dispenser may have a screen across it until the user puts the card in. The screen closes down when cash is removed. Similarly, the "cash removed" event cannot occur until the cash is presented to the user.
- There may be some *logical* constraint imposed by surrounding software. For instance, we may be considering an abstraction of the real system which is enclosed by a layer performing some syntactic checking.

In either case we may represent the set of possible actions by a subset L of P. However, this subset must satisfy a simple temporal condition. If some command sequence is possible, then all initial subsequences must also be possible, else it wouldn't have been possible to produce the longer sequence. That is:

$$\forall q \in P,\ p \in L \qquad q \leq p \quad \Rightarrow \quad q \in L$$

If we have an exception PIE then the persistence condition on *ex* is exactly equivalent to this condition on the exception-free language:

$$L_{ex} \;=\; Ok \;=\; \{\ p \mid \textbf{not}\ ex(\ p\ )\ \}$$

Note that this condition is as weak as possible, as the temporal condition is a minimal one to make the language sensible. Often there will be some more complex condition. For instance, the possible command histories may be initial subsequences of a context-free grammar, as is the case in Anderson's work. (Anderson 1985)

We may require that it is possible to predict whether a command will be part of the language from the effect, as we have for exceptions, but usually the constraints will be evident if they are physical, or given by the enveloping system if they are logical. Thus in general such a requirement will not be necessary; if it is, then the exception conditions give a reasonable template.

Of greater importance is the way this affects the functionality of the system. For instance, if we design cash dispenser software that asks for the customer's PIN number, then reads in the cash-card, then gives the money, it will appear to have sufficient functionality when considered without the language constraints. However, when we take into account the language, we see that the customer cannot type in the PIN number until the card has been read and the cover raised. The machine turns out to be rather tight-fisted. On the other hand, it is rarely safe to assume when one is designing a system that the input will conform to what one expects to be inviolable constraints: the cash machine's window that covers the keyboard may be broken!

The latter problem means that one of the exception rules should be used for inputs that are not part of the language, and because this is a serious problem the "no guesses" rule would be appropriate. Thus we can assume that any command history a system has is effectively from the language, even if not actually so. The restriction in the input set must, however, be reflected in the formal statement of our principles. For instance, strong reachability for language PIEs becomes:

strong reachability with language:

$$\forall p, q \in L \qquad \exists\, r \in P \qquad \textbf{st} \qquad pr \in L \quad \textbf{and} \quad I(\ pr\ ) \;=\; I(\ q\ )$$

The monotone closure and hence megareachability must also be modified, as two states cannot be said to be equivalent based only on their effect: they must also have the same language extensions. Thus we redefine monotone equivalence for languages:

$$\begin{aligned} &\forall p, q \in L \quad p \equiv^{\dagger} q \quad \hat{=} \\ &\qquad \forall r \in P \quad pr \in L \Leftrightarrow qr \in L \quad \textbf{and} \quad pr \in L \Rightarrow I(pr) = I(qr) \end{aligned}$$

If the language does have warning information like *is_ex*? then this is no different from the standard definition, as the effects being the same ensures the same language extension.

2.10 Relations between PIEs

As a mathematician, one of the first things one does when faced with a new class of objects is to look at the relationships between those objects. This section will examine the relationships beween pairs and collections of PIEs. There are several ways one can relate PIEs at an abstract level, and it will be seen how these correspond to different ways of describing interactive systems. On the whole, the tenor of this section will be rather abstract, but it forms a sound basis for the more concrete discussion of layered systems in Chapter 7.

This examining of the relationship between things is the very stuff of mathematics, but is not just abstract speculation. It is fairly obvious that there is some insight to be gained by, for instance, looking at the way the cursor behaves in a word processor in isolation. For the purposes of thinking about the cursor, our keyboard could effectively have just a single "x" key for typing and the normal complement of cursor control keys. By thinking about the cursor on its own, we lose the distraction of other aspects of the system, and are able to better understand the specific problems related to cursor movement. We then happily relate this abstract view back to the full system, enabling us to extend our understanding of the part to the whole. This process of relating different models of a single system happens all the time in our informal analysis, often without our ever noticing. The different relationships in this chapter attempt to capture some of this informal reasoning in a formal framework.

2.10.1 Isomorphism

We saw in §2.3 that the $<P, I/\equiv_I, P/\equiv_I>$ was equivalent to $<P, I, E>$ in the sense that the following diagram commuted, where *proj* is a one-to-one function:

$$\begin{tikzcd}
 & E \\
P \arrow[ur, "I"] \arrow[dr, "I/\equiv_I"'] & \\
 & P/\equiv_I \arrow[uu, "proj"']
\end{tikzcd}$$

This is, of course, a special case of isomorphism of algebras. The general form of an isomorphism between $<P, I, E>$ and $<P', I', E'>$ requires two one-to-one relations, *parse* between P and P' and *proj* between E and E', such that the following diagram commutes:

$$\begin{tikzcd}
P \arrow[r, "I"] \arrow[d, leftrightarrow, "parse"] & E \arrow[d, leftrightarrow, "proj"] \\
P' \arrow[r, "I'"] & E'
\end{tikzcd}$$

In fact, as we are interested in P as a semigroup, we also require that *parse* is a semigroup isomorphism. That is:

$$parse(\ p,\ p'\ ) \quad \textbf{and} \quad parse(\ q,\ q'\ ) \quad \Rightarrow \quad parse(\ pq,\ p'q'\ )$$

For the normal case, where P is freely generated over the command set C and likewise P' from C', this means that *parse* is generated from a one-to-one relation between C and C'.

Of course, although this seems like a sensible isomorphism when considering the PIEs as mathematical objects, it is not necessarily sensible for the user. For instance, a projection that involved reversing the letters in the alphabet ("abc" to "zyx", etc.) would be perfectly acceptable mathematically, and in regard to its information content, but no user would regard them as being the same. Similarly, when we say that E and $P/\equiv_I$ are equivalent up to isomorphism, that does not mean that a display editor and an equivalence class of command histories are equally useful for the user!

2.10.2 Other relations

Clearly, the most general relationship between PIEs would be obtained by letting *parse* and *proj* be general relations, perhaps with some restrictions on *parse*. However, we will only deal with a few special cases, in all of which we assume that the relations are in fact functions. There are clearly two major cases to consider:

- *1-morphisms* – both *parse* and *proj* go in the same direction. Without loss of generality, say *parse*: $P \rightarrow P'$ and *proj*: $E \rightarrow E'$.
- *2-morphisms* – *parse* and *proj* go in opposite directions. Say *parse*: $P \rightarrow P'$ and *proj*: $E' \rightarrow E$.

In addition, there is the case when one or other of *parse* or *proj* is one-to-one, and hence 1-morphisms and 2-morphisms coincide. We will call this special case:

- *0-morphism* – either *parse* or *proj* one-to-one.

Further, we will consider only two special cases of 1-morphisms: when both *parse* and *proj* are injective, which we shall call *extension,* and when they are both surjective, which we will call *abstraction.*

We will deal with each of these cases in the succeeding sections in the order 0-morphism, extension, abstraction and finally 2-morphism. But first we will consider the restrictions we may want to place on *parse*.

2.10.3 Restrictions on *parse*

If we are interested in P and P' as general subgroups then it is reasonable to consider the meaning of requiring *parse* to be a subgroup homomorphism. That is:

$$\forall p, q \in P \qquad parse(\ p ; q\ ) \ = \ parse(\ p\ )\ ;\ parse(\ q\ )$$

In the normal case, when P is generated from C, this corresponds to simple macro expansion of commands from C into sequences of commands from P'.

Often this will prove a too stringent a condition, and in these cases we will often find that the temporal well-ordering condition introduced in §2.3 will be useful:

$$\forall p, q \in P \qquad q \le p \;\Rightarrow\; parse(\, q \,) \le parse(\, p \,)$$

This, of course, only has meaning if the semigroup is in fact free (that is, *P* is the sequences from *C*): however, this is the most usual case, the only exception we have dealt with being the construction of *P*/~. Note also that it is strictly weaker than the semigroup homomorphism condition.

This condition does seem to be quite general; the only reason for breaking it is when we want to "backtrack" on some decision of *parse*. Not surprisingly, the relationships that do not obey this condition, are those generated by undo mechanisms. We will therefore assume that the temporal relation holds for all the morphisms we consider, but only ask for the semigroup condition when necessary.

Why didn't we ask for this rather than semigroup isomorphism when considering PIE isomorphism above? It would indeed have been possible, and indeed the sort of bisimulation condition needed between the handle space models of window managers would obey the weaker condition only. However, for simple PIEs, the informal idea of two systems being isomorphic must surely be that they differ in a one-to-one way between keypress and effects (perhaps also in any manifest effect structure). There is, of course, no right answer: one uses the definition that is most appropriate to the circumstances.

Before we move on, the temporal condition does suggest the possible generalisation of PIEs to where *P* is simply a well-ordered set. However, this has no obvious meaning at the user interface, and I have found no useful construction yielding such an object.

2.10.4 0-morphisms

We have already dealt with several 0-morphisms: in particular, the relation between monotone closure and its original PIE is a 0-morphism where the *P* spaces coincide:

$$\begin{tikzcd}
 & E \\
P \arrow[ur, "I"] \arrow[dr, "I^{\dagger}"'] & \\
 & E^{\dagger} \arrow[uu, "proj^{\dagger}"']
\end{tikzcd}$$

The relation between P and $P^{\sim}$ also forms a 0-morphism, except here it is the effect spaces which coincide:

$$\begin{tikzcd}
P \arrow[dr, "I"] \arrow[dd, "parse^{\sim}"'] & \\
 & E \\
P^{\sim} \arrow[ur, "I^{\sim}"'] &
\end{tikzcd}$$

In this case we have a general semigroup, and the semigroup homomorphism condition does in fact hold. The temporal condition is of course inapplicable to general semigroups.

The exception relation (§2.9) is a generalisation of the 0-morphism. Two further examples will come in the next chapter when we consider red-PIEs: the relations between I and I_r and between I and I_d will both be 0-morphisms.

2.10.5 Extension and restriction

This is the special case of a 1-morphism where both *parse* and *proj* are injective. A 1-morphism, we recall, is when *parse* and *proj* both go in the same direction:

$$\begin{tikzcd}
P \arrow[r, "I"] \arrow[d, "parse"] & E \arrow[d, "proj"] \\
P' \arrow[r, "I'"] & E'
\end{tikzcd}$$

This would arise if, having an existing system $<P, I, E>$, we added extra commands to it to increase its functionality. Alternatively, we might start with $<P', I', E'>$ and *restrict* it by cutting down the commands available (perhaps to meet size or performance constraints).

In §2.5 when we considered simple predictability, we suggested that it would be a suitable condition to apply to certain parts of the system. These parts could be specified formally as a restriction of the system. For instance, we may restrict the command set of an editor to cursor movement and direct typing, and apply the condition to the restricted PIE so obtained.

Another example would be the editor with simple cursor movement of §2.4. We extended it by adding commands to set and jump to a mark. In these examples the command and effects of the latter were simple extensions of the former. So long as we never used the mark, the two interpretation functions matched exactly.

In both these examples the *parse* function of the extension is a semigroup homomorphism, and I cannot think of any reason to have it otherwise in the case of extensions in general. If we do assume this, then it is fairly simple to prove that monotonicity of PIE$'$ implies monotonicity of PIE. So, if we have a predictable system, restricting it will not spoil it. On the other hand, if we have an unpredictable system we cannot make it predictable by extending it. This is a case of "adding commands to a bad system doesn't make a good system".

It must be emphasised, however, that this only applies to the definition of predictability given by monotonicity. If we look at strategies and observable effects, and define predictability in terms of these, we see that restricting the command set may remove commands necessary for the strategy and hence destroy predictability; contrariwise, adding commands may make an effective strategy possible.

The notion of extension is not really that useful, as although the extended system behaves exactly like the original when *no* other commands are entered, it tells us nothing about whether there is any sort of behaviour consistent with the original when other commands have been used. We would normally want to say something a bit stronger. For instance, in an editor with cut/paste, we want to say that the effect of using just the cursor keys and normal typing is "the same" even when some other commands have been typed. The start state will of course be different. This can be partially captured by a "bundle" of extensions, one for each member of P'. Explicitly, for each p' from P' there exists p and $proj_{p'}$ as follows:

$$p \in P$$
$$proj_{p'} : E \to E'$$
$$proj_{p'} \text{ is injective}$$
$$I'_{p'} \circ parse = proj_{p'} \circ I_p$$

That is, the following is an extension:

$$\begin{tikzcd}
P \arrow[r, "I_p"] \arrow[d, "parse"] & E \arrow[d, "proj_{p'}"] \\
P' \arrow[r, "I'_{p'}"] & E'
\end{tikzcd}$$

The *parse* functions are all the same, as we would expect the connection between commands to be the same. The projections need to be different, however, as the results of the "extra" commands in p' may change the connection between the effects.

As an example of such a bundle, let us look at the extension of the simple typewriter to the editor with cursor movement. The *parse* function is the identity, and for any sequence of commands p' from P', the characters with cursor movement, we simply take p to be *before* where:

$$\{\ before, after\ \} \quad = \quad I'(\ p'\ )$$

The projection then merely maps an effect in E into the first component of E':

$$proj_{p'}(\ e\ ) \quad = \quad \{\ e, after\ \}$$

This says that for any period when the cursor commands are not used, the text before the cursor acts as if generated by a simple typewriter.

Returning to the general case of a bundle, the fact that there are different projections does unfortunately allow anomolies. It is quite possible for one projection to take the text "abc" to "abc" as one would expect, but for another to take the same text to "def": we would obviously like them to agree on the "unextended" part of the effect. In the section on 2-morphisms we will introduce a further condition that will enforce such consistency.

The bundle also does not help with more complex extensions like the MARK/JUMP editor. This is because even if no more MARKs or JUMPs are entered, the projection function cannot keep track of the position of the MARK.

2.10.6 Abstraction

The case of the 1-morphism where *parse* and *proj* are both surjective, we call abstraction. It is probably the most important of the PIE relations. It is called abstraction because the two PIEs have identical behaviour except that PIE distinguishes more commands and effects than PIE′. Again we could look at it the other way round, and say that PIE refines PIE′.

Consider the typewriter with delete from §2.4.3, $< P_\nabla, I_\nabla, E_\nabla >$, and the simple counter $< P', I', E' >$ defined by:

$$\begin{array}{lcl} C' & = & \{ -1, +1 \} \\ E' & = & \{ 0, 1, 2, 3, \cdots \} \end{array}$$

$$\begin{array}{lcll} I'(\ null\ ) & = & 0 & \\ I'(\ p::+1\ ) & = & I'(\ p\ ) + 1 & \\ I'(\ p::-1\ ) & = & I'(\ p\ ) - 1 & I'(\ p\ ) > 0 \\ & = & 0 & I'(\ p\ ) = 0 \end{array}$$

The counter PIE is an abstraction of the typewriter when they are related by the maps:

$$\begin{array}{lcll} parse(\ c\ ) & = & +1 & c \neq \nabla \\ parse(\ \nabla\ ) & = & -1 & \end{array}$$

(the rest of *parse* is the semigroup homomorphism generated from these)

$$proj(\ e\ ) \quad = \quad length(\ e\ )$$

This is the abstraction whereby the counter just keeps a tally of the number of characters typed.

In various places in the book we will mention the idea of something being an abstraction of another, referring to the general idea of having one or more abstraction functions. However, some instances deal with models other than the PIE model and thus cannot be handled explicitly by the above. For example, in Chapter 5, when we consider time-dependent systems, we find that systems may have some facet such as a clock which would never reach steady state, but we can look at some abstraction of the system which would stabilise. We could formalise this abstraction as desiring the existence of a *proj* map between the actual display (with clock) and the abstracted display (without). We can then apply all the analysis which we develop concerning steady-state behaviour to this abstracted temporal model.

We can investigate some of the properties of a PIE by looking at the simplified PIE′. One result that is useful here is that if *parse* is a semigroup homomorphism and PIE is strong (mega) reachable then so is PIE′. This would tend to be useful in the negative: if we were to find that an abstraction of our system is not reachable then we would know that some modification of the original system is necessary. The relevant modifications may become obvious when considering the abstraction. Abstraction is also useful when considering how one system is built in layers upon another, a point we will return to in Chapter 7.

If we allow abstractions to have exception conditions as in §2.9, we can have bundles of abstractions in a similar way to bundles of extensions. This would enable us to handle the relation between the MARK/JUMP and the plain cursor editor. The *parse* function simply throwing away *additional* MARK commands, with an exception being raised for JUMPs. The projection then just strips the MARKs from the effect. That is, for any p in P_{mk}, we choose p' in P_{cursor} to be *before* and define *parse* and $proj_p$ to be:

$$\begin{aligned} parse:\ & P_{mk} \rightarrow P_{cursor} \\ & parse(\ p\ ) \quad = \quad strip(\ p\ ) \\ proj_p:\ & E_{mk} \rightarrow E_{cursor} \\ & proj(\ (\ b, a\ )\ ) \quad = \quad (\ strip(\ b\ ), strip(\ a\ )\ ) \end{aligned}$$

where *strip* removes any MARKs or JUMPs. These form an abstraction with exceptions for each p. We notice that the exception condition obeys the "nothing silly" condition, since every time we have an exception, the cursor editor can be seen as being in the same state as if the relevant p' had been entered.

2.10.7 2-morphisms, implementation

In a 2-morphism the two functions *parse* and *proj* go in opposite directions:

$$\begin{tikzcd}
P \arrow[r, "I"] \arrow[d, "parse"'] & E \\
P' \arrow[r, "I'"] & E' \arrow[u, "proj"']
\end{tikzcd}$$

The commutativity condition is therefore:

$$I \quad = \quad proj \circ I' \circ parse$$

We have seen an example of this already, namely the relation between any PIE and its associated PIE generated from $E^{\dagger}$ and $P^{\sim}$:

$$\begin{tikzcd}
P \arrow[r, "I"] \arrow[d, "parse^{\sim}"'] & E \\
P^{\sim} \arrow[r, "I^{\sim\dagger}"] & E^{\dagger} \arrow[u, "proj^{\dagger}"']
\end{tikzcd}$$

We will call the 2-morphism an *implementation* because if we had coded $< P', I', E' >$ already, one way to implement $< P, I, E >$ would be using the functions *parse* and *proj*. For example, if we had implemented the calculator with memory for PIE′, we could implement the simple calculator by making both *parse* and *proj* identity maps. Although this seems a very useful construction it does not live up to its promise. This will become apparent when we consider such constructions in detail in Chapter 7. But now we will show how a 2-morphism *can* be useful in combination with an extension.

We recall that in order to have a better concept of a restricted system, we introduced a bundle of extensions parametrised over the extended command sequences P'. Each extension had a separate projection function $proj_{p'}$. These needed to be different, as the "extended" part of the effects might differ, but we wanted a way of enforcing consistency on the "unextended" part. We can ensure this by having an additional function *proj* from E' to E, the same for all the PIEs in the bundle, which makes all the PIEs in the bundle into 2-morphisms when taken with the original (constant) *parse*:

$$\begin{tikzcd}
P \arrow[r, "I_p"] \arrow[d, "parse"'] & E \arrow[d, shift right, "proj_{p'}"'] \\
P' \arrow[r, "I'_{p'}"] & E' \arrow[u, shift right, "proj"']
\end{tikzcd}$$

This extra implementation projection function ensures the consistency between the different extension projections. To be precise, the separate commutativity conditions imply that:

$$proj \circ proj_{p'} = identity$$

For example, consider extending the simple editor with cursor movement. Recall for any p' from the extended commands we chose p to be *before*, and defined $proj_{p'}$, where:

$$\begin{aligned} \textbf{let} \quad \{\ before, after\ \} &= I'(\ p'\ ) \\ proj_{p'}(\ e\ ) &= \{\ e, after\ \} \end{aligned}$$

The projection *proj* is then simply the function that extracts the "before" half of the effect:

$$proj(\ (\ b, a\ )\ ) = b$$

2.10.8 What we can and cannot capture in PIE-morphisms

In this short analysis we can see some success, but also some problems in describing the relationship between systems. We will see in Chapter 7 that the *abstraction* relation is particularly important and has wide generality when decomposing systems. Difficulties in relating formally systems that we believe to be similar can arise for two reasons.

First, the problem may be a strictly technical. Our models miss out the significant features that would enable us to express the relationship. If this is the only barrier, a more extensive model will allow us to proceed. For instance, the *pointer space* model described in Chapter 8 gives us just such enhanced expressive power. The price we may pay for such power will typically be a more complex model, and therefore a model which is more difficult to analyse (although which is capable of more extensive analysis). However, if we are careful the model may not necessarily be more complex, just more appropriate for the relationship we wish to express.

The second, more fundamental problem is that the similarity we perceive informally may be in some way unformalisable. Its expressiveness may be related to a deep understanding of different situations that is lost if we try to tease out the similarity too precisely. This is the contrast between metaphor on the one hand, and allegory on the other. When we formalise a relationship, we effectively allegorise it; this feature in the one situation corresponds to that feature in the other, etc. If we are unaware of this problem attempts to formalise a metaphor may be counterproductive, capturing the peripheral parts whilst missing the central issues entirely. However, if we keep these limitations in mind it can be useful to find just how far a similarity can be captured formally, and thus highlight where we do rely on intuition.

2.11 PIEs – discussion

Even using the very simple PIE model, we have been able to investigate some non-trivial properties of interactive systems. We have introduced the principles of predictability, observability and reachability, and the important notion of monotone closure. We have also examined problems of undo, and dealing with exceptions in interactive systems. Finally, we have seen how we can relate different systems to one another. This was done at a largely theoretical level, but gives us the machinery with which to discuss layered design and implementation in Chapter 7.