A Neighborhood of Infinity

Sunday, May 03, 2009

The Three Projections of Doctor Futamura

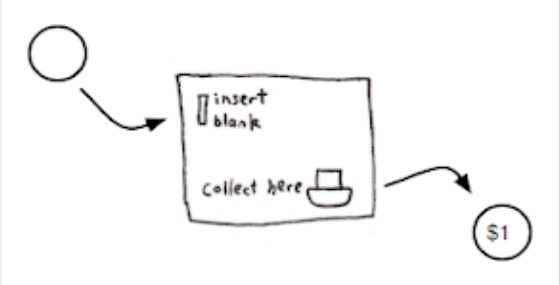
Introduction

The Three Projections of Futamura are a sequence of applications of a programming technique called 'partial evaluation' or 'specialisation', each one more mind-bending than the previous one. But it shouldn't be programmers who have all the fun. So I'm going to try to explain the three projections in a way that non-programmers can maybe understand too. But whether you're a programmer or not, this kind of self-referential reasoning can hurt your brain. At least it hurts mine. But it's a good pain, right?

So rather than talk about computer programs, I'll talk about machines of the mechanical variety. A bit like computer programs, these machines will have some kind of slot for inputting stuff, and some kind of slot where output will come out. But unlike computer programs, I'll be able to draw pictures of them to show what I'm talking about. I'll also assume these machines have access to an infinite supply of raw materials for manufacturing purposes and I'll also assume that these machines can replicate stuff - because in a computer we can freely make copies of data, until we run out of memory at least.

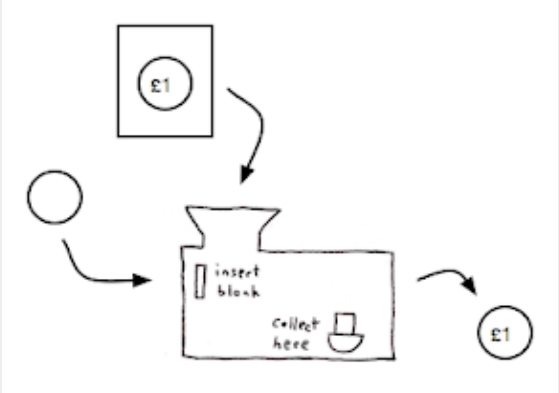
Minting coins

A really simple example of a machine is one that has a slot for inputting blanks, and outputs newly minted coins:



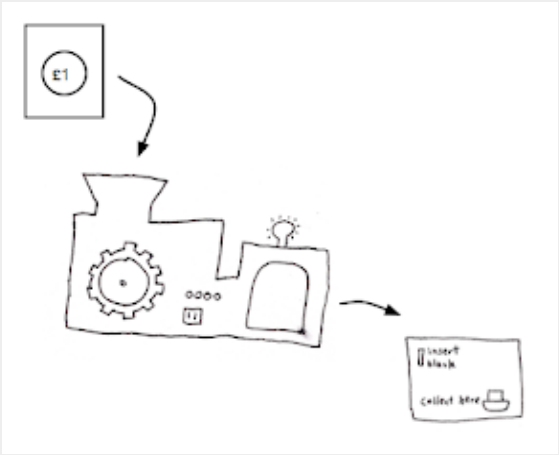
That's a dedicated $1 manufacturing machine. We could imagine that internally it stamps the appropriate design onto the blank and spits out the result.

It'd be more interesting if we could make a machine with another input slot that allowed us to input the description of the coin. By providing different inputs we could mint a variety of different coins with one machine. I'm going to adopt the convention that when we want to input a description we input a picture of the result we want. I'll draw pictures as rectangles with the subject inside them. Here's a general purpose machine manufacturing pound coins:



The same machine could make dollars, zlotys or yen. You could imagine this machine works by taking the description and then milling the coin CNC style. We call such a machine an interpreter. It interprets the instructions and produces its result.

The interpreter has a great advantage over the dedicated dollar mint. You make make any kind of coin. But it's going to run a lot slower. The dedicated minter can just stamp a coin in one go. The interpreter can't do this because every input might be different. It has to custom mill each coin individually. Is there a way to get the benefits of both types of machine? We could do this: take the coin description and instead of milling the coin directly we mill negative reliefs for both sides of the coin. We then build a new dedicated minting machine that uses these negatives to stamp out the coin. In other words we could make a machine that takes as input a coin description and outputs a dedicated machine to make that type of coin. This kind of machine making machine is called a compiler. It takes a set of instructions, but instead of executing them one by one, it makes a dedicated machine to perform them. Here's one in action:



So here are the two important concepts so far:

Interpreters: these take descriptions or instructions and use them to make the thing described.
Compilers: these take descriptions or instructions and use them to make a machine dedicated to making the thing described. The process of making such a machine from a set of instructions is known as compiling.

The Projections of Doctor Futamura help make clear the relationship between these kinds of things.

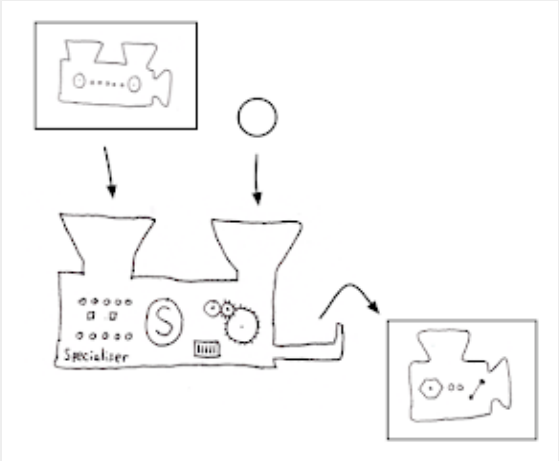
Specialisation

We need one more important concept: the specialiser. Suppose we have a machine that has two inputs slots, A and B. But now suppose that when we use the machine we find that we vary the kind of thing we put into slot B, but always end up putting the same thing into slot A. If we know that slot A will always get the same input then we could streamline the machine using our knowledge of the properties of A. This is similar to the minting situation - if we know we're always going to make $1 coins then we can dedicate our machine to that purpose. In fact, if we know that we're always going to input the same thing into slot A we don't even need slot A any more. We could just stick an A inside the machine and whenever the user inputs something to slot B, the machine would then replicate the A and then use it just as if it had been input.

In summary, given a machine with two slots A and B, and given some input suitable for slot A, we could redesign it as a machine with just a B slot that automatically, internally self-feeds the chosen item to A. But we can often do better than this. We don't need to self-feed stuff to slot A. We might be able to redesign the way the machine works based on the assumption that we always get the same stuff going into slot A. For example, in the minting example a dedicate $1 minter was more specialised than just a general purpose minter that interpreted the

instructions for making a $1 coin. This process of customising a machine for a particular input to slot A is called specialisation or partial evaluation.

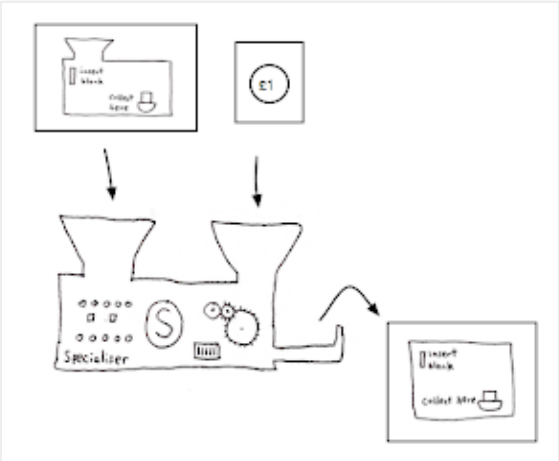
Now imagine we have a machine for automatically specialising designs for machines. It might have two slots: one for inputting a description for a two slot machine with slots A and B, and one for inputting stuff suitable for slot A. It would then print out a description for a customised machine with just a slot B. We could call it a specialisation machine. Here is one at work:



It's converting a description of a two input machine into a description of a one input machine.

The First Projection

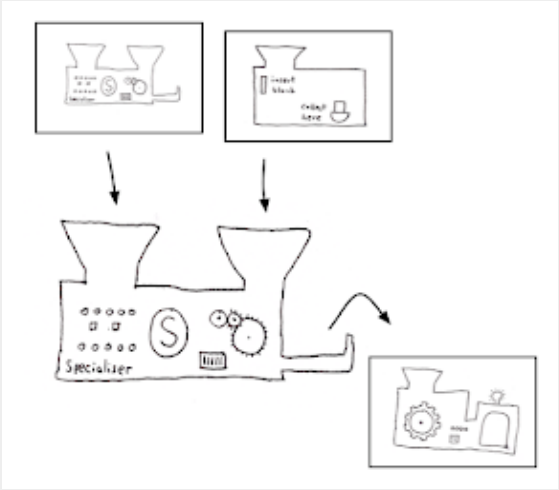
The process of specialisation is similar to what I was talking about with dedicated minting machines. Rather than just have a similarity we can completely formalise this. Note that the interpreter above takes two inputs. So the design for an interpreter could be fed into the first input of a specialiser. Now we feed a description the coin we want into slot B. The specialiser whirrs away and eventually outputs a description of a machine that is an interpreter that is dedicated to making that one particular coin. The result will describe a machine with only one input suitable for blanks. In other words, we can use a specialiser as a compiler. This is the first of Doctor Futamura's Projections. Here's a picture of the process at work:



What this shows is that you don't need to make compilers. You can make specialisers instead. This is actually a very practical thing to do in the computing world. For example there are commercial products (I'm not connected with that product in any way) that can specialise code to run on a specific architecture like CUDA. It's entirely practical to convert an interpreter to a compiler with such a tool. By writing a specialiser, the purveyors of such tools allow third parties to develop their own compilers and so this is more useful than just writing a dedicated compiler.

The Second Projection

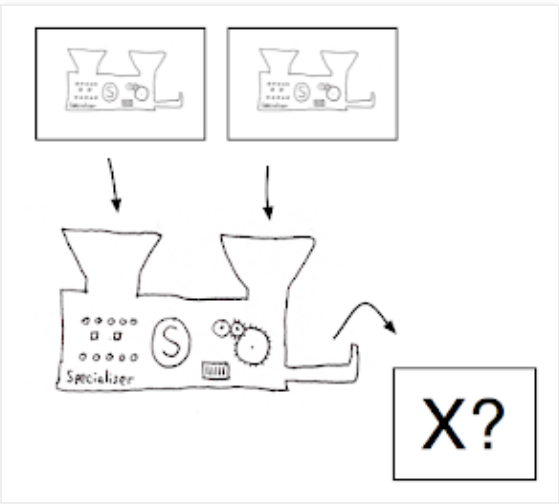
Time to kick it up a notch. The first input to the specialiser is a description of a two input machine. But the specialiser is itself a two input machine. Are you thinking what I'm thinking yet? We could stuff a description of a specialiser into the specialiser's own first input! In the first projection we provided an interpreter as input to the specialiser. If we know we're always going to want to use the same interpreter then we could streamline the specialiser to work specifically with this input. So we can specialise the specialiser to work with our interpreter like this:



But what is that machine whose description it has output? An interpreter takes as input a description of how to operate on some stuff, like turning blanks into coins. In effect, the output machine has the interpreter built into it. So it takes descriptions and outputs a machine for performing those instructions. In other words it's a compiler. If the specialiser is any good then the compiler will be good too. It won't just hide an interpreter in a box and feed it your description, it will make dedicated parts to ensure your compiler produces a fast dedicated machine. And that is Doctor Futamura's Second Projection.

The Third Projection

But we can go further. The specialiser can accept a description of a specialiser as its first input. That means we can specialise it specifically for this input. And to do that, we use a specialiser. In other words we can feed a descrption of a specialiser into *both* inputs of the specialiser! Here we go:



But what is the X machine that it outputs? In the second projection we pass in an interpreter as the second argument and get back a compiler. So the third projection gives us a dedicated machine for this task. The X machine accepts the description of an interpreter as input and outputs the description of a compiler. So the X machine is a dedicated interpreter-to-compiler converter. And that is the Third Projection of Doctor Futamura.

If we have a specialiser we never need to make a compiler again. We need only design interpreters that we can

automatically convert to compilers. In general it's easier to write interpreters than compilers and so in principle this makes life easier for programmers. It also allows us to compartmentalise the building of compilers. We can separate the interpreter bit from the bit that fashions specific parts for a task. The specialiser does the latter so our would-be compiler writer can concentrate on the former. But who would have guessed that passing a specialiser to itself *twice* would give us something so useful?

Summary

So here are the projections:

1. Compiling specific programs to dedicated machines.
2. Making a dedicated machine for compilation from an interpreter.
3. Making a machine dedicated to the task of converting interpreters into compilers.

There are lots of variations we can play with. I've just talked about descriptions of things without saying much about what those descriptions look like. In practice there are lots of different 'languages' we can use to express our descriptions. So variations on these projections can generate descriptions in different languages, possibly converting between them. We might also have lots of different specialisers that are themselves optimised for specific types of specialisation. The Futamura projections give interesting ways to combine these. And there are also variations for generating dedicating machines for other tasks related to compiling - like parsing the descriptions we might feed in as input. If you want to read more on this subject there's a whole book online with example code. They're not easy things to design. I think that specialisation is a killer feature that I'd like to see more of. Present day compilers (and here I'm talking about computers, not machines in general) are hard-coded black boxes for the task of compilation. They're not very good at allowing you to get in there and tweak the way compilation occurs - for example if you want to generate code according to a strategy you know. Specialisation is a nice alternative to simply bolting an API onto a compiler. It would make it easy for anyone to write optimising and optimised compilers for their own languages and combine such compilers with interpreters for interactive instead of offline compilation. I learnt about this stuff, as well as lots of other stuff in my blog, from the excellent Vicious Circles. The theory is closely related to the theory of writing quines that I used for my three language quine. And if you keep your ears to the ground you can hear rumours of a fabled fourth projection...

Appendix

I wanted to address some of the comments so I've added an appendix where I use the Haskell type checker to tighten up the statements I make above. There are some places I made some choices and I decided to make the specialiser output machines rather than pictures. This code doesn't actually do anything.

One important thing to note is that with these definitions the first projection is a function describing the action of a machine after it has been given one input. The second and third projections are dedicated machines.

```
> module Futamura where
```

I'm using `P a` to represent the type of a Picture (or Plan, or Program) of how to perform an operation of type `a` and `M a` to represent a Machine (or executable) that executes such an operation.

I use `M` because I want to make explicit what is actually a machine. In Haskell a type `a -> b -> c` can be thought of as a machine that takes an input of type `a` and an input of type `b` and outputs a `c`, or as a machine that takes as

input an `a` and outputs another machine that makes a `c` from a `b`. I distinguish those by using `M (a -> b -> c)` for the former and `M (a -> M (b -> c))` for the latter.

I'm not actually going to built a Futamura specialiser so the right hand sides hare are just filler:

```
> data P a = P
> data M a = M
```

Running a machine gives a way to perform what the machine is designed to do. We're not really running machines in Haskell so we have an undefined right hand side.

```
> run :: M a -> a
> run = undefined
```

A specialiser is a machine that takes as first input a picture of a process mapping an `a` and a `b` to a `c`. It also takes as argument the specialised value for the input for the process. It then outputs a dedicated machine for the specialised process:

```
> specialise :: M (P (a -> b -> c) -> a -> M (b -> c))
> specialise = undefined
```

We actually need the picture of the specialiser as it's going to be specialised:

```
> specialisePicture :: P (P (a -> b -> c) -> a -> M (b -> c))
> specialisePicture = undefined
```

For the first projection we'll need an interpreter. An interpreter is a general purpose machine that takes pictures of how to map an `a` to a `b`, as well as an actual `a`, and can then give you a `b`:

```
> interpreter :: M (P (a -> b) -> a -> b)
> interpreter = undefined
```

And what we really need is a picture of an interpreter:

```
> interpreterPicture :: P (P (a -> b) -> a -> b)
> interpreterPicture = undefined
```

The first projection turns a picture into a dedicated machine. So it functions as a compiler. But note that it's not itself a dedicated machine. It's a general purpose machine which acts as a compiler when given (a picture of) an interpreter as first input:

```
> proj1 :: P (input -> output) -> M (input -> output)
> proj1 = run specialise interpreterPicture
```

The second projection is a dedicated machine that does the task of `proj1`. So it's a compiler:

```
> proj2 :: M (P (input -> output) -> M (input -> output))
> proj2 = run specialise specialisePicture interpreterPicture
```

An interpreter is something that can take a computer program and some input and generate the output you expect from the program. A compiler, on the other hand, converts programs into dedicated machines to process the input into the output. And that's exactly what the third projection does:

```
> proj3 :: M (P (program -> input -> output) -> M (program -> M (input -> output)))
> proj3 = run specialise specialisePicture specialisePicture
```

---

sigfpe at Sunday, May 03, 2009

25 comments:

Martijn Monday, 04 May, 2009

That's really interesting.

I bet the higher projections are progressively harder to implement. Could it be that they are so hard to implement that they are not worth implementing anymore?

Reply

Josef Monday, 04 May, 2009

Thanks for the post, I really like it.

I think it's worth clarifying one thing though. You say "If we have a specialiser we never need to make a compiler again". Although it's technically correct it might be read as if we will never ever need a compiler as long as we have specialisers. But that's not true. A compiler is the only machine which turns a description of a machine into an actual machine. So it's not specialisers all the way down, we need a compiler at the bottom.

Reply

sigfpe Monday, 04 May, 2009

Josef,

Or you use one of the language changing specialisers I mentioned - In particular one that targets assembly language.

Reply

Anonymous Monday, 04 May, 2009

I wonder how many others read that as Futurama and thought of Dr. Zoidberg.

Reply

Josef Monday, 04 May, 2009

Ah, yes, of course.

Anyhow, it's still the case that your pedagogy breaks down in the particular case for machine language, when the description and the actual program is the same. It's a shame, because I think it's very clear otherwise.

Reply

Arnar Birgisson Monday, 04 May, 2009

Excellent article, as usual. Although I do have a tiny bit of feeling that there is a even better analogy out there somewhere.

Reply

Arnar Birgisson Monday, 04 May, 2009

Sorry for a second comment, but what does the specializer depend on? I.e. when we build a specializer (a partial evaluator) for a programming language, I imagine it depends on the programming language in question, right? Does it also depend on the target language?

Reply

sigfpe Monday, 04 May, 2009

Arnar,

I tried to avoid talking about multiple languages to make things easier to understand.

But any individual specialiser is typically written in language A, takes an input to specialise written in language B, and outputs something in language C. (Or directly outputs executables rather than source.) It's now kinda fun to work out schemes for using combinations of specialisers to bootstrap specialisers from one language to another.

You can also consider specialisers that only work with a subset of a language - for example a subset just big enough to write a specialiser. Or specialisers good enough to specialise an embedded DSL if not the full host language.

Reply

Doug Merritt Monday, 04 May, 2009

It seems to me that there is a clear connection with reflective towers, e.g. as discussed in extraordinary detail in Lisp in Small Pieces, but I don't recall whether the latter (or anything else) examines the connection.

As with reflective towers, mathematically it seems clear that the most natural theory actually does have not just a fourth projection, but actually an infinite number of projections (although one could axiomatically chop it short to suit one's tastes), but the connection of each level to some understandable concrete concepts seems questionable -- perhaps not unlike the general theory of N-Categories?

That thought is not identical to Martijn's, but is similar in flavor.

Reply

syoyo Tuesday, 05 May, 2009

FYI, I'm been working writing RenderMan Shader Language compiler in Haskell with specialization facility.

I believe shader specialization is killer application of Futamura projection.

See my demo if you are interested in this field.

http://vimeo.com/3294472

Reply

Dave Tuesday, 05 May, 2009

Handwaving wildly, it seems like there ought to be a topological problem here: if one curries f(stat,dyn) to f(stat)(dyn), then a specializer ought to produce f_stat(dyn). This is like saying that one can, instead of traversing two sides of a triangle, wind up at the same place by traversing the third. In some cases we can smoothly map a path from the first two sides to the third, in other cases -- particularly when the implementation is hardware based -- we run into an exponential blowup. Could it be useful to classify these situations into the presence or absence of the triangle "face"?

Reply

augustss Tuesday, 05 May, 2009

Writing a specializer that actually does something interesting when applied to itself is a really tricky. It took the DIKU group something like 10 years to figure out how.

You can bypass some of the problems by writing cogen (the 3rd Futamura projection) by hand.

Reply

sigfpe Tuesday, 05 May, 2009

augustss,

I'd expect it to be really hard. I can see that it'd be easy to specialise many kinds of purely numerical code, say, because you're just simplifying mathematical expressions. But specialising a piece of code that takes source code as input, and hence specialising all of the decisions made during the process of, say, parsing and interpreting that code, seems pretty scary to me.

Reply

augustss Tuesday, 05 May, 2009

Partial evaluators that are not to be self applied are very useful too, but they don't tickle the imagination in the same way.

In fact, I would almost claim that most of the commercial Haskell code I've written have been partial evaluators of one kind or another. But only one actually went by that name.

Reply

Guy Steele Wednesday, 06 May, 2009

Very nice! This blog entry is a delightful metaphorical introduction to the concepts---I love the drawings---and seems to be technically sound. One very slight glitch: in the last two drawings, I believe that the right-hand input to the specializer should be not a picture of a machine, but a picture of a picture of a machine. This is a subtlety that is all too easy to gloss over.

Reply

Peter Berry Wednesday, 06 May, 2009

```
> {-# LANGUAGE EmptyDataDecls #-}
> data Pic a
> type Specialiser a b c = (Pic ((a,b) -> c), a) -> Pic (b -> c)
>
> specialise :: Specialiser a b c
> specialise = undefined
>
> spec_pic :: Pic (Specialiser a b c)
> spec_pic = undefined
>
> test = specialise (spec_pic, spec_pic)
```

*Main> :t test
test :: Pic (Pic ((a, b) -> c) -> Pic (a -> Pic (b -> c)))

test is the picture of X that you get by feeding the specialiser with two pictures of itself. So X is a machine that takes a picture of a machine - _not_ a picture of a picture of a machine.

Reply

sigfpe Wednesday, 06 May, 2009

Peter,

We can have a bit of fun with Pic. If we have a picture of an a, it tells us how to make an a. So we have a map

Pic a -> a

Now if we have an a, it's hard to make a picture of it, because it involves reverse engineering all of its internals. In fact, the machine might be booby trapped. So we have no map, in general, a -> Pic a. But we do have a map

Pic a -> Pic (Pic a)

because if you have a complete picture of an a there's nothing hidden that prevents you fully describing that in a higher order picture.

So Pic is very close to a comonad. In fact, it's the modal operator from the S4 logic: http://www.cs.cmu.edu/~fp/talks/scottfest02.pdf (see p.16)

So if I wasn't feeling so stupid I could probably turn the Futamura projection into something interesting in S4.

Reply

Peter Berry Wednesday, 06 May, 2009

Guy, I see now I misread your criticism. But it's still a picture of a machine, not a picture of a picture of one, as you can see from the type.

Dan, that's pretty interesting. Somehow I'd never thought of applying Curry-Howard to modal logic before. By 'very close to a comonad' do you mean 'looks like a comonad but I haven't proved it yet' or 'actually isn't a comonad'? Not that I have much of a grasp on just what a comonad is, mind.

Reply

sigfpe Thursday, 07 May, 2009

Peter,

If Pic were a comonad we'd have a function

(a -> b) -> Pic a -> Pic b

but we only have

Pic (a -> b) -> Pic a -> Pic b

The latter corresponds to the fact that we can draw a diagram for constructing a b by showing first how to make an a and then showing how to make a machine to transform an a into a b.

Reply

Guy Steele Thursday, 07 May, 2009

Peter Berry,

Thanks for your nice type analysis, and I think it illuminates the problem. In the sigfpe essay, the diagram under the heading "Specialisation" indeed corresponds to your type

type Specialiser a b c = (Pic ((a,b) -> c), a) -> Pic (b -> c)

But the next diagram, under the heading "The FIrst Projection", has a different type:

type Specialiser a b c = (Pic ((a,b) -> c), Pic(a)) -> Pic (b -> c)

and this is the type that I relied on when making my original criticism. So the best I can say now is that there does seem to be a slight type inconsistency somewhere.

Reply

Anonymous Sunday, 10 May, 2009

I had to follow to the Wikipedia link before I read it properly and noticed it was Doctor *Futamura* rather than Doctor *Futurama*.I feel somewhat disappointed.

Reply

Zachary Vance Tuesday, 09 February, 2010

Dear Guy,
In the second diagram, under the heading "The First Projection", the type is still
type Specialiser a b c = (Pic ((a,b) -> c), a) -> Pic (b -> c)
where a = Pic(d). If you look at the top, you'll see the original machine takes a picture of a coin, which makes sense for coins but has no "program as data" connotation, possibly the source of your confusion. The diagrams are quite consistent.

Reply

solrize Wednesday, 14 July, 2010

The Google Books link to "Vicious Circles" is broken. Could you post the title and author info? Thanks.

Reply

Bobby Jameson Saturday, 07 August, 2010

It looks like you guys are building some type of slot machine or something. I'm not sure I understand, is that what it is, a slot machine???

Reply

Sonny O. Wednesday, 02 February, 2011

I'm sorry to say but I believe I'm one of those non-programmers whom you failed to explain the three projections to a level of understanding to. It did make my brain hurt a bit but I'll have to give you props for the effort.

I do understand the GIGO (Garbage In Garbage Out) process but any internal processes prior to output is still a bit of a blur for me. Even trying to picture it out as a scene in the game incredible machines didn't do the trick.

In any case, I will keep reading your blog for further comment exchange and resources until I get this. Thanks.

Reply

To leave a comment, click the button below to sign in with Google.

SIGN IN WITH GOOGLE

‹ Home ›

View web version

About Me

sigfpe

View my complete profile

Powered by Blogger.