A theoretical view of Human-Centered Computing

Clayton Lewis
University of Colorado, Boulder
DRAFT: Please send comments and suggestions to
clayton dot lewis at colorado dot edu

Note: These notes are being developed to help introduce undergraduate students to HCC, as a contribution to the HCC Research Cluster of a Broadening Participation in Computing Alliance, headed by North Carolina A&T State University.

A theoretical view of Human-Centered Computing 1
1. Introduction 2
2. Representations and the World of Information Technology 3
3. Making the idea of representation formal 7
3.1 Theory of Measurement 7
3.2 From measurement to representation 10
3.2.1 Expressiveness and effectiveness: “Human-Centered” comes in. . 11
3.2.2 Usefulness 12
3.3 A economical formal analysis of representation: category theory.......... 12
3.4 What does a representation represent? 15
4. Implementations.. 15
4.1 What it means to implement a mapping 15
4.2 Implementation by people 16
4.2.2 Input representations: Representations created by people............ 17
4.2.3 Actions in context... 18
4.2.4 HCC and social systems... 19
4.2.5 HCC and particular user groups 20
4.4 Mappings implemented by computer... 20
4.4.1 What it means for a computer to implement a mapping.................. 20
4.4.2 Human-Centered Shapes and Sizes 21
4.4.3 Unintentional Communication 21
4.5 How to Create Computer Implementations 22
4.5.1 Programs and Programming Languages 23
4.5.2 HCC and programming 24
4.5.3 The two faces of representations in computational systems........... 25
5. So, What is Human-Centered Computing? 26
APPENDIX: More about category theory. 28

1. Introduction

Human-Centered Computing, or HCC, is a very broad area, containing many fascinating areas of work. Here is the synopsis of the program description for the Human-Centered Computing Cluster at the National Science Foundation:

> This cluster, Human-Centered Computing (HCC), encompasses a rich panoply of diverse themes in Computer Science and IT, all of which are united by the common thread that human beings, whether as individuals, teams, organizations or societies, assume participatory and integral roles throughout all stages of IT development and use.
>
> HCC topics include, but are not limited to:
>
> * Problem-solving in distributed environments, ranging across Internet-based information systems, grids, sensor-based information networks, and mobile and wearable information appliances.
> * Multimedia and multi-modal interfaces in which combinations of speech, text, graphics, gesture, movement, touch, sound, etc. are used by people and machines to communicate with one another.
> * Intelligent interfaces and user modeling, information visualization, and adaptation of content to accommodate different display capabilities, modalities, bandwidth and latency.
> * Multi-agent systems that control and coordinate actions and solve complex problems in distributed environments in a wide variety of domains, such as disaster response teams, e-commerce, education, and successful aging.
> * Models for effective computer-mediated human-human interaction under a variety of constraints, (e.g., video conferencing, collaboration across high vs. low bandwidth networks, etc.).
> * Definition of semantic structures for multimedia information to support cross-modal input and output.
> * Specific solutions to address the special needs of particular communities.
> * Collaborative systems that enable knowledge-intensive and dynamic interactions for innovation and knowledge generation across organizational boundaries, national borders, and professional fields.
> * Novel methods to support and enhance social interaction, including innovative ideas like social orthotics, affective computing, and experience capture.
> * Studies of how social organizations, such as government agencies or corporations, respond to and shape the introduction of new information

technologies, especially with the goal of improving scientific understanding and technical design.

Many people, perhaps including you, will see topics on that list that are interesting and important, and that you want to work on. By all means, dive in and get involved.

But for some people, and perhaps you are in this category, it can be hard to get your head around an area that's as broad as this, and that seems to be defined by examples rather than by unifying ideas. If you are the kind of person who likes better defined, more formal ideas, can HCC work for you, or you for it?

The answer is yes. There are formal perspectives in HCC. These aren't (yet) very well represented in the literature... maybe you will play an important role in developing and spreading these ideas.

An idea that is central to all of computing, and also communication, and mathematics as well, is ***representation***. As I'll try to show, many things about the usefulness of computing, and how it relates to other disciplines, can be understood by focusing on the role of representations. I'll also try to show that the idea of representation can be made sharp and formal. Finally, I'll use this idea to define what HCC is, and in the process, show that HCC is not, as some have thought, a peripheral aspect of computing, but rather a central aspect, not just in its practical importance, but also in the intellectual content it shares with other areas of computing.

2. Representations and the World of Information Technology

Let's get started without pausing to develop any definition of representation. We'll use our everyday ideas and experience, and we'll begin with an example. We can represent the length of a steel beam by a number. We can represent this number by printed digits on a piece of paper, or by a pattern of glowing colored dots on a CRT screen, or (inside a computer) by a pattern of currents flowing through transistors in a circuit in a computer, or by a pattern of magnetized spots on a rotating disk.

This example, simple though it is, brings out some of the special characteristics of the world of information. One is its ***layer structure***: lengths are represented by numbers, numbers are represented by (say) magnetized spots. Another is ***generality***: numbers are a suitable representation for many different things-- not just lengths but also weights, prices, temperatures, and on, and on. Another is ***interchangeability***: for some purposes it does not matter much whether numbers are represented by marks on a page, or by magnetized spots, and for a

great many more purposes it matters little whether they are represented by magnetized spots or by currents in tiny transistors. Often, different representations can be substituted for one another without changing what can be represented.

Physical representations. Among the example representations we've talked about, some, like magnetized spots, or glowing dots, are ***physical***: you can point to them out there in the world. Others, like numbers, aren't physical in the same way. Where's the number 2.5? It can be represented physically in the world, but it can also exist as an idea, or ***mental*** representation, just in your head. Interchangeability means that for a given mental representation, like a number, there can be many *different* possible physical representations.

While choice of physical representation for something like numbers does not affect what can be represented, it does profoundly affect the practical value of the representation. Some physical representations allow information to be stored much more cheaply, and in far less space, than others, or allow information to be retrieved or transmitted far more rapidly. The recent explosion of interest in computing and communication is due in large part to rapid progress in developing physical implementations of representations that have these advantages.

These advantages have important effects in spreading the uses of representations. The ability to store information cheaply, to retrieve it quickly, and to transmit it sufficiently rapidly to permit new ways for people to share information, have made possible new methods for commerce and scholarship, to name just two examples.

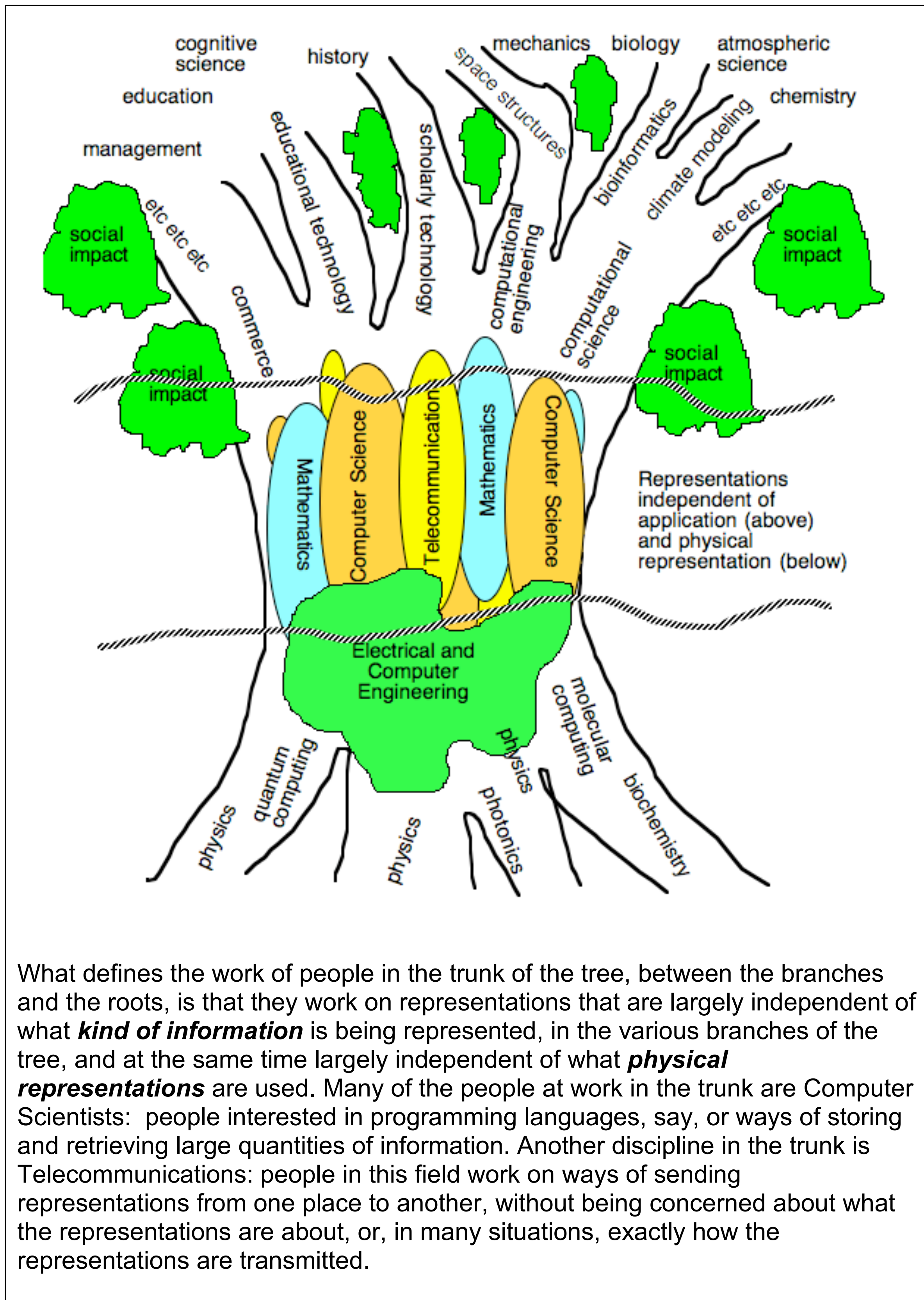
How do Computer Science, Electrical Engineering, and Mathematics, and all those other fields fit together?

The layered structure of representations, together with interchangeability, determine the outline of the traditional disciplines in computing and communication. Roughly speaking, the province of Electrical and Computer Engineering is the physical implementation of representations, and of devices for manipulating them (for example, processor chips), while the creation and manipulation of representations defined "on top of" these implementations (so-called "software") is the province of Computer Science. The separation is broadly workable precisely because one can specify how to manipulate numbers (for example) without caring whether the numbers are represented by magnetized spots or currents in transistors or whatever.

"On top of" the common representations of software one finds representations supported by a myriad of application programs for work of all kinds. Layering and interchangeability are at work here, too, ensuring (for example) that a user of a word processing program has no reason to know what methods it uses for finding

or changing text, what programming language was used to create the program, or what kind of processor it is running on, or how that processor represents letters in its memory. Thus a word processor user need know nothing, happily, of Computer Science or Electrical and Computer Engineering.
One can picture this situation as a kind of layer cake. At the top is the domain of word processing, containing documents and operations on them. Below that is the layer of Computer Science, in which one sees how documents are represented as collections of characters, and how these characters are stored and manipulated. Below that is the layer of Electrical and Computer Engineering, where one sees how characters are represented and manipulated by the action of electronic circuits of various kinds.

If we broaden our view, to take in other activities than word processing, we see that there are vertical divisions that cut the horizontal layers in the cake. In fact, we will do better to replace our image of a cake by an image of a tree, with branches at the top, a trunk in the middle, and roots at the bottom. Towards the top of the tree, among the branches, people interested in the broad area of text processing can be separated from those interested in (say) programs to support analysis of reinforced concrete structures, or programs for creating ultrasound images of the heart, or programs for processing information about inventory. In universities it's usual for people with these differing interests to be in different departments.
Towards the bottom of the tree, among the roots, people working on implementations that use magnetics or electronics would be found in Electrical and Computer Engineering departments, while work on photonics would be found there or in Physics, work on molecular computing in Chemistry or Biochemistry departments, and work on quantum computing in Physics. Layering and interchangeability mean that it’s possible for these groups to make progress working separately.



What defines the work of people in the trunk of the tree, between the branches and the roots, is that they work on representations that are largely independent of what ***kind of information*** is being represented, in the various branches of the tree, and at the same time largely independent of what ***physical representations*** are used. Many of the people at work in the trunk are Computer Scientists: people interested in programming languages, say, or ways of storing and retrieving large quantities of information. Another discipline in the trunk is Telecommunications: people in this field work on ways of sending representations from one place to another, without being concerned about what the representations are about, or, in many situations, exactly how the representations are transmitted.

Also at work in the trunk of the tree are mathematicians. Like Computer Science, Mathematics is concerned with representations, and usually considers representations without worrying about what the topics being represented are. The key distinction between Computer Science and Mathematics is that computer scientists usually restrict their attention to representations that, in theory at least, can be given physical form, while mathematicians usually aren't subject to this restriction.

3. Making the idea of representation formal

3.1 Theory of Measurement

So far we've been using an intuitive idea of what a representation is. It's time to get more specific, and more formal. Our goal is to develop a ***theory of representations***, within which we can state and answer definite questions. A good starting point is a theory of ***measurement***, a limited kind of representation in which the representations are simple things like numbers. Here we can draw on organized theoretical work such as that reviewed in the book series *Foundations of Measurement*, by Krantz, Luce, Suppes, and Tversky. Let's take as an example the measurement of length. In a measurement system we have two collections of things that are related: objects of some kind, say steel beams, and their lengths, which are numbers. We'll call the beams our ***subject domain***, and we'll call the numbers the ***representation domain***. We're interested in the numbers because they represent something about things in the domain we are really interested in, the target of our attention, the beams.

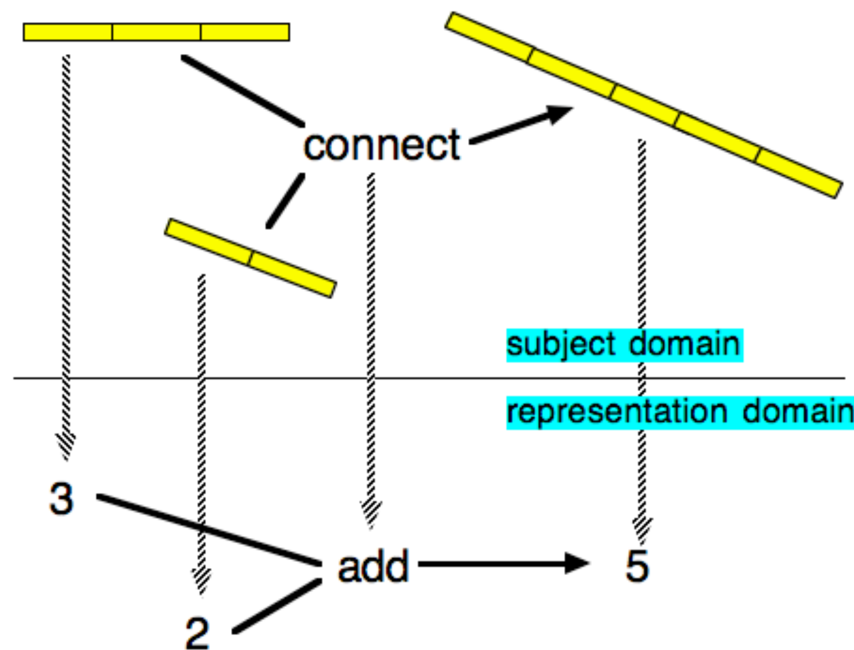
We also have some ***operations***, things we can do with the beams, such as putting two beams side by side to see which is longer, or connecting two beams together. The key idea of measurement is that we want to be able use operations in the representation domain, that is, operations on numbers, to *substitute* for operations in the subject domain. For example, suppose we want to see which of two beams is longer. We could lay the two beams out side by side, slide them so that they are lined up at one end, and see which one sticks out past the other. But this may involve a lot of work, if the beams are large, or if they are already fastened to some structure, or for many other reasons. It's much easier to *measure* the two beams, getting two numbers, and then just *compare the numbers* to see which is bigger. Thus the operation of *comparing two numbers* can ***substitute*** for *comparing two beams*.

The key requirement for a measurement system is that this substitution gives correct results. If I don't assign numbers to beams in the right way, for example, if count the rivet holes in beams, instead of measuring their lengths, I'll get some

wrong answers when I compare the numbers that I get. A beam with six rivet holes may or may not be longer than a beam with none.

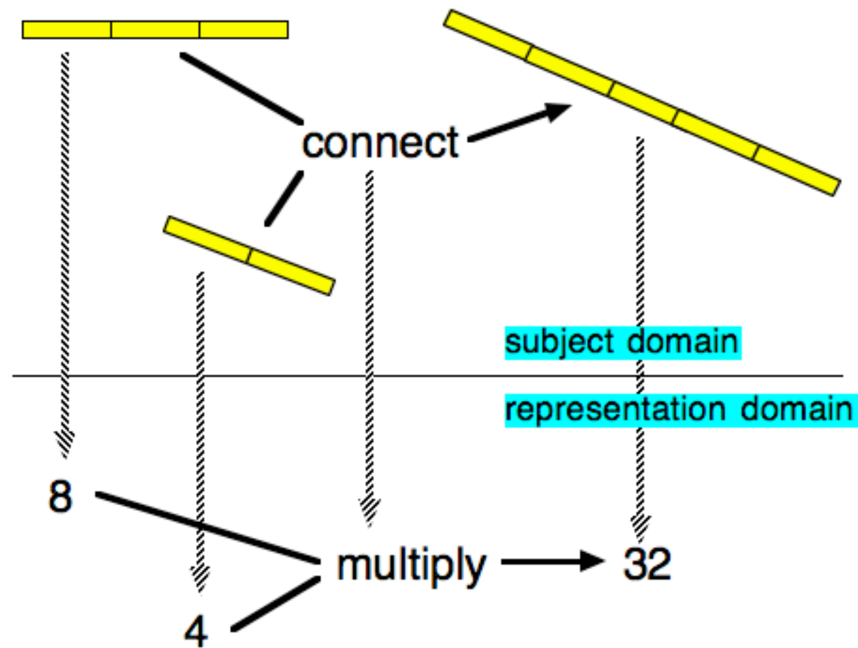
Another operation on beams is connecting them together. Suppose we have two beams, want to know how long a beam we will get if we connect them end to end in a straight line. We could actually connect them, and measure the resulting beam. But it would be easier to measure the two beams separately and then *add* the resulting numbers. Here we are substituting addition, an operation on numbers, for an operation on beams, connecting them end to end. If our measurement system works, we'll get the same answer whether we connect the beams and then measure, or measure the beams and then add.

Here's a diagram that shows the situation. It shows the subject domain, beams; the representation domain, numbers; operations in these two domains; and the *correspondences* between the two domains (shown as dashed arrows).

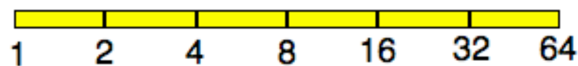


> ***What's special about adding lengths?***
>
> Do you think you could have a way of measuring lengths in which you would *multiply* the lengths of beams to find their combined lengths, instead of adding them? If this were possible, the lengths would have to be different from the familiar ones. Allowing this, could you come up with a system that works? You may want to try out some ideas before you read further.

In fact, you *can* define such a measurement scheme. This picture shows an example:



Here is the kind of ruler you have to use. Notice that the length of no beam at all has to be 1 in this system, not 0, so that adding "no beam" to a beam doesn't change the length when you multiply. The markings on the ruler were determined by using the familiar additive length as the exponent in a power of two: familiar length 0 becomes $2^0=1$, familiar length 1 becomes $2^1=2$, and so on.



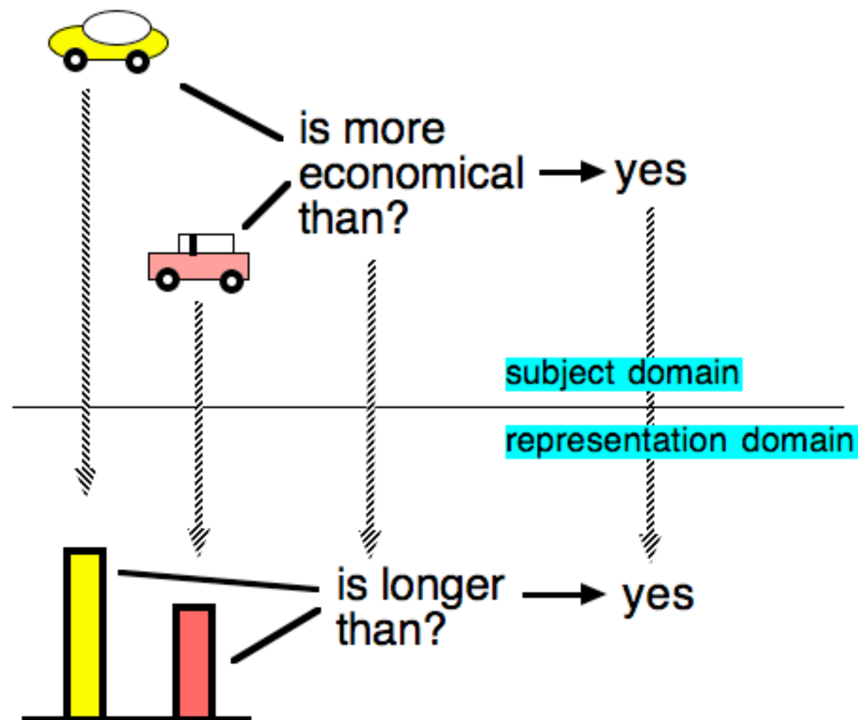
Multiplicative Ruler

Now, do you think you could come up with a system for measuring lengths in which you *subtract* to find the length of a combined beam? *Hint:* You can't. Why not?

3.2 From measurement to representation

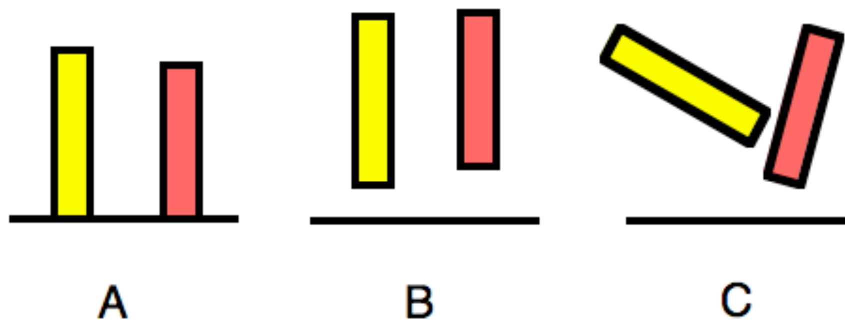
Jock Macinlay and Michael Genesereth took the first steps in generalizing this account of measurement, to produce an analysis of how representations in general work [Mackinlay, J. and Genesereth, M. R. 1985. Expressiveness and language choice. *Data Knowl. Eng*. 1, 1 (Apr. 1985), 17-29; Mackinlay, J. 1986. Automating the design of graphical presentations of relational information. *ACM Trans. Graph*. 5, 2 (Apr. 1986), 110-141; both papers available at http://www2.parc.com/istl/projects/uir/publications/author/Mackinlay.html. See also Wehrend, S. and Lewis, C. 1990. A problem-oriented classification of visualization techniques. In *Proceedings of the 1st Conference on Visualization '90* (San Francisco, California, October 23 - 26, 1990). A. Kaufman, Ed. IEEE Visualization. IEEE Computer Society Press, Los Alamitos, CA, 139-143, available at http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=949553#]. While theory of measurement covers relatively simple mathematical things in the representation domain, like numbers, we can open up the idea to include more complicated structures, such as bar charts. Suppose, for example, that we are interested in comparing the fuel economy of various cars. We could *measure* the fuel economy of the cars, using numbers (say miles per gallon). But we could also *represent* the fuel economy of the cars using *bars* for each car. The crucial thing is to assign the bars to the cars in such a way that if one car is more economical than another, then its bar is longer. When we want to compare two cars, we just compare their bars.

You can see that this setup isn't very different from the measurement setup. We have a subject domain, cars, a representation domain, bars, an operation on cars (determining if one car is more economical than another), and an operation on bars (determining if one bar is longer than another). The setup works if comparing the bars that correspond to two cars always gives the same answer as comparing the cars. Here’s a picture showing this:



3.2.1 Expressiveness and effectiveness: "Human-Centered" comes in.

With this example we can see where being "human centered" comes into the picture. Look at these three bar charts, all based on the same data.



The first one, in Panel A, is what you expect. The other two are unusual. Why? First, note that all three charts satisfy the requirement that comparing bars gives the same answer as comparing the cars. In fact, the bars in the three charts have the same lengths. So, logically, all three charts are acceptable. But the familiar chart seems better than the other two: it's much easier to read. Getting more specific, we can see that comparing bars is easy when they are *parallel*, and

when *their bases are lined up*. The chart in Panel C violates both of these conditions, and is pretty hopeless, even though it is logically "correct".

Notice that the problems with the chart in Panel C are problems with *human* perception. A robot with good image processing might have no trouble comparing the bars, but people do have trouble. So the goodness or badness of representations for human use is a ***human-centered*** judgment.

Macinlay and Genesereth noted that there are therefore two requirements that a good representation has to meet, if it is to be useful for people. There's the logical requirement that the operations on the representation have to give the same answers as the corresponding operations in the subject domain. Macinlay and Genesereth call this ***expressiveness***. But in addition, the operations used in the representation domain have to be *easy for people to carry out*. Macinlay and Genesereth call this ***effectiveness***. Effectiveness is intrinsically a human-centered attribute: it can't be assessed without knowledge of *human* processing capabilities, in this example, facts about what makes bar lengths easy or hard for people to compare.

3.2.2 Usefulness

There's one more feature of representations that's important, along with expressiveness and effectiveness. In a good representation system the operations that are supported are ***useful***: they are representations people want to work with, or need to work with. If no one cares about comparing the fuel economy of cars, there'd be no point in designing a bar chart to make the comparisons easier. As we'll see, judgments about usefulness play an important role in developing human-centered computing systems.

3.3 A economical formal analysis of representation: category theory

The logical part of the analysis of representations, the ***expressiveness*** requirement, can be expressed very neatly using the concepts of ***category theory***, a powerful mathematical framework that's catching on in some areas of mathematics and theoretical computer science. These notes will introduce some of these concepts as a way of sharpening up our ideas about representations. If the ideas make sense, great. If not, you'll find that you can still work with representations.

A ***category*** is a bunch of collections of things, such as cars and bars, or beams and numbers, and operations on them, called ***mappings***. A mapping takes a thing as input and produces a thing as output. For example, we could have a mapping, call it "nl", from the collection of numbers from 1 to 26 to the letters, in alphabetical order: we give it 7 as input and the mapping gives us G as the

output. Notice that there are usually many different mappings that connect the same collections: I could have a mapping from the numbers to the letters that counts from the end of alphabet, so that 1 gets mapped to Z, 2 to Y, and so on, so that 7 gets mapped to T rather than G. All we ask is that for any given input, the mapping has to give us just one output.

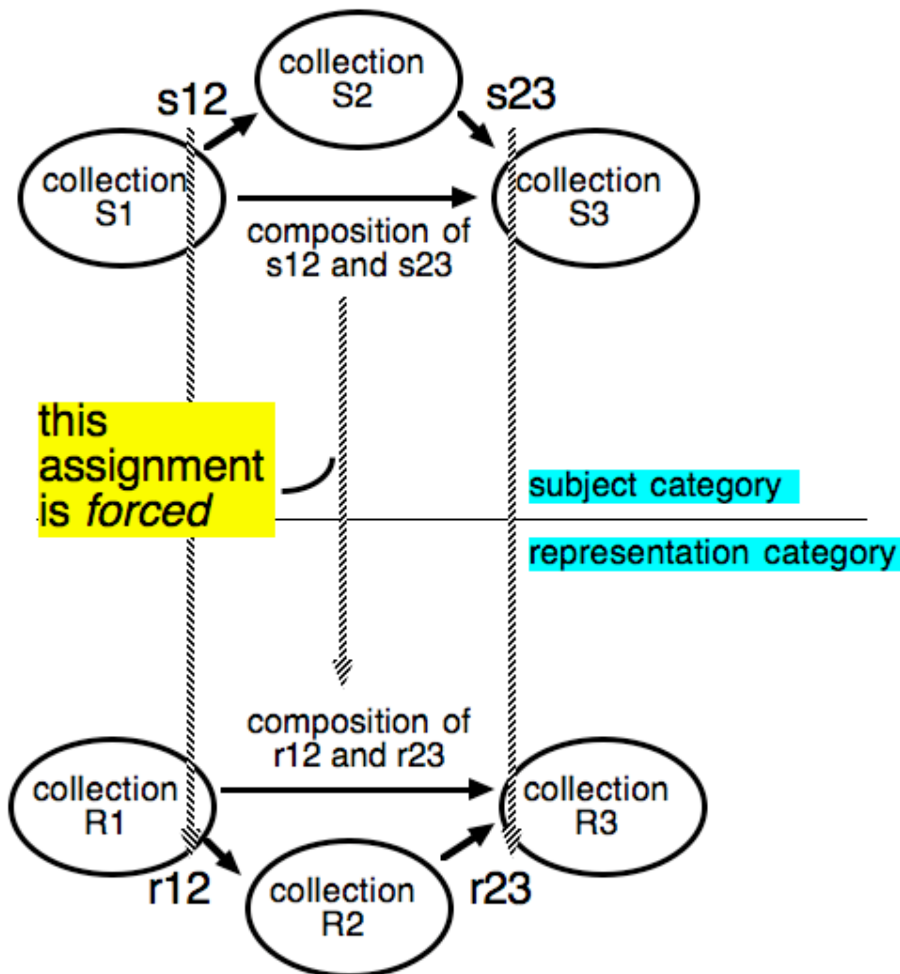
If it happens that we have a mapping ab from some collection A to another collection B, and a mapping bc from B to some other collection C, we can ***compose*** ab and bc to get a mapping ac that goes directly from A to C. The output of ac for any input is just the output bc would give, when it is given as its input the output ab would give. Composition is a simple idea, once you understand it, but it's the key to the workings of category theory.

As an example, suppose we have a mapping (call it "lw") from letters to words that works by taking a letter as input, and producing the last word in some particular dictionary that starts with that letter, as input. If we provide G as input the mapping gives "gyve" as the result (for the dictionary on my desk). Now suppose we compose nl and lw. What output will it give if we give it 7 as input? The answer is "gyve", because nl gives G when we give it 7, and lw gives "gyve" when we give it G.

To use category theory to understand the theory of representations, we'll have a category representing the subject domain, and another category representing the representation domain.The connection between the subject category and the representation category is called a ***functor***. A functor assigns a collection of things in the representation category (say a collection of bars) to each collection in the subject category (the cars). It also assigns a mapping in the representation domain to each mapping in the subject domain.

A key job for the functor is to tell us which mappings in the representation domain can be used to ***substitute*** for mappings in the subject domain. In the example of the lengths of beams we talked about earlier, the the functor tells us that we can substitute adding numbers for connecting beams, when we want to find out how long two beams will be when we stick them together.

In assigning corresponding mappings, a functor has to ***preserve composition***. That is, suppose in the subject category we have mappings s12,s23, and their composition, s13. Then if a functor assigns mapping r12 in the category to s12, and r23 to s23, it *must* assign the composition of r12 and r23 to s13. Here's a picture:



If we set up our subject and representation categories as just described, it turns out that we can show that any legal functor creates a working representational scheme, in the sense that mappings in the representation category can substitute for mappings in the subject category, and give the corresponding answers.

You might think we would have to have special conditions on the functor, to require that it assign bars to cars in such a way that comparing bars gives the same answers as comparing cars. But, remarkably, the basic requirements on functors are enough to guarantee the behavior that we need. You can see the details of the argument in the Appendix. You'll see a few more category theory concepts in the argument, including how to deal with pairs of things, something we commonly need to do (in putting together two beams, or comparing two cars), and a more complete definition of functor. The Appendix also suggests some further reading if you are interested in category theory.

3.4 What does a representation represent?

In familiar language we talk about the bars in the bar chart representing something, in this case the fuel economy of the cars. But that's really a loose way of talking, and can lead to confusion. In fact, the bars *on their own* don't represent *anything*. The very same bars could represent fuel economy, or weight, or price, and for that matter something that has nothing to do with cars, like number of teeth.

Bars, and other representations, only work in the context of a representation system, in which the correspondences between things in the representation category and the subject category are spelled out. Actually, it's this fact that makes possible the ***generality*** of representations that we talked about earlier as a big plus. The same representations, say numbers, can be used in different representation systems, to represent very different kinds of things. We don't need new representations every time we encounter a new subject domain.

So a representation, say a number, doesn't say on its own what it represents. It can represent different things in different representation systems. At the same time, a thing in a subject domain doesn't say what its representation has to be. One single thing, say the fuel economy of a car, can be represented by *different* representations in *different* representation systems, say by a bar or by a number. That's crucial to ***interchangeability***, another plus we talked about earlier.

4. Implementations

We can talk about representations just as ideas, but to put them to work we need ***implementation***, some way of actually carrying out the operations that form part of an representational system. We're going to use the operations in the representation domain to substitute for operations in the subject domain, and that means we actually want to put in inputs and get outputs. That means we'll want to implement the mappings, and also create the representations that the mappings work on.

4.1 What it means to implement a mapping

An implementation of a mapping is a thing with two ends, an input end and an output end. There must be some means of setting the input end so as to correspond to something in the input collection for the mapping. Then, after some delay, the output end gets set to correspond to something in the output collection for the mapping.

A person can implement a mapping. The person perceives the input, in some way, and then creates some representation of the output. Mechanical, electronic, or other technical devices can also implement mappings, as we'll discuss later.

Notice that implementing a mapping required creating representations, in two ways. Someone or something has to set up the representation of the input. If a person does that, that set up has to be something that a person can easily and accurately do. Whatever implements the mapping itself has to set up the output representation.

4.2 Implementation by people

4.2.1 Mappings

Some of the operations, or mappings, in a typical representation problem have to implemented by people. In many cases, including the bar chart example, technology can be used to support these operations, by creating and presenting representations on which people can operate. But it's the people that have to do the work after that. Here are some examples.

Bar charts: The bars are drawn by computer, and then people can make the actual judgements. When the a person implements a mapping in this context, they perceive the bars, and then create an output. If the bars are base-aligned and parallel, the length comparison will be easy and accurate.

Picture processing: Recognizing objects or people, or judging aesthetic qualities, or many other everyday operations, can be done from pictures. These operations can all be seen as mappings from collections of things, or scenes, in the world to Yes or No, or more complex collections. You provide a picture as input, and the person can provide a Yes or No output (Yes, this thing is familiar) or an aesthetic judgement (The aesthetic value of this scene is "ugly"). If the real things or scenes aren't at hand, they can be represented by pictures, and judgements made from the pictures can substitute for the judgements that would be made from the real things or scenes. Conveniently, suitable pictures can be created and displayed by computer, taking into account the following characteristics of human vision.

> *Color perception*: For *homo sapiens* the brightness of just three colors can be chosen to reproduce perfectly any color (but see http://www.purveslab.net/main/ for complications that are at work in color vision). A display for dogs would only need two colors (dogs only discriminate short from long wavelengths.)
>
> *Limited spatial resolution, and blending*: Below a certain size, closely spaced dots merge into a smooth-appearing area. Exploiting these two

principles, most displays work by presenting a large number of tiny, closely spaced dots, of three colors. Even though this arrangement produces images that are physically very different from "real" images seen in the world, to the human visual system these images look realistic, and they are processed effectively and easily.

Depth perception: There are many cues for depth, that is, distance from the viewer. Nearly all of these are included in two-dimensional images. For example, ***perspective***, the effect by which more distant objects form smaller images, or ***aerial perspective***, the effect by which more distant objects appear fainter or more hazy, because they are seen through more intervening atmosphere, are preserved in two dimensional images. A depth due that is important for objects near at hand is ***stereopsis***: the fact that the images seen by the two eyes differ in a way that is related ton distance. Displays that exploit this indication of depth present different images to the two eyes, for example by using differently polarized light for the two images, and placing differently polarized filters over the two eyes. As for images made of dots, notice that the resulting pattern of light is very different from the pattern of light in a real scene. But it is different in a way that the human visual system doesn't detect. There could be organisms whose visual system is sensitive to the polarization of light. For such organisms these displays wouldn't work.

Moving from pictures to moving scenes, presentation of movies and videos relies on further facts about how people see. Just as closely spaced dots merge into the appearance of smooth surfaces, so images closely spaced ***in time*** merge into the appearance of smooth motion. As we all know, a movie consists of a series of frames, each a static picture, changed very rapidly, but we don’t see it that way. One could imagine a Martian in a movie theater wondering why Earthlings like to sit in the dark and watch long series of very similar pictures flashed on a screen. Video is more complicated, in that the pictures aren’t even complete: they are collections of closely spaced lines, with only half the lines included in each frame. But this ridiculous presentation looks quite good to the human visual system.

If you are interested, you can extend this list by adding information about how sound is presented, how letters, words, and characters are seen, and the like. In each case, computers can support people in performing important operations, like enjoying a melody, or understanding a scholarly, by creating representations of these things that are *matched to human perceptual abilities*.

4.2.2 Input representations: Representations created by people.

In the examples just given the mappings are carried out by people on representations that are presented to them by a computer. From the point of view of the computer these are ***output*** representations. But it’s usually also necessary

for people to produce representations that are given to the computer, so as to provide data for the computer to operate on, or to control what the computer does. From the point of the computer these are ***input*** representations. Just as for output representations, input representations have to be shaped to fit people's capabilities. While for output representations the human abilities that matter center on perception, since input representations have to be produced by people, the key capabilities are those of ***action***. What are the actions people can control, to produce representations usable by a computer?

Keys and keypresses. Most people have dexterous fingers that can be moved quickly and accurately so as to apply force to appropriately sized objects. Keys and buttons are such objects, and are designed to fit people's fingers in size, the force required to activate them, and (for high quality keys) the feedback they provide to confirm that they have been activated. Most people can move their fingers so as to press more than one key at a time, and this ability is exploited in many musical instruments and a few computer input devices, such as chord keyboards.

Text entry. Often keys are used in groups to allow people to specify sequences of characters making up a text. For people who know a written language, many pieces of text are familiar, and can be generated quickly and accurately, whereas random sequences of keypresses can be entered only slowly and with high error rates.

Drawing. Most people can use their hands and fingers to grasp a pointed object of appropriate size and shape, and move the point along a desired path. With more difficulty, most people can move an object that has no point (a mouse) so as to control a pointed marker whose movement traces a desired path. In either case, the path can be sensed and act as input to a computer.

You can add many examples to this list, and perhaps invent new ways to use actions to communicate. For example, HCC researchers are developing ways for people to use facial expression, or tone of voice, to create input representations.

4.2.3 Actions in context.

In any of these examples, what is being represented by the user's actions is determined by context . As we've seen, being a representation means being part of a system, and without knowing about the system, one can't tell what is being represented.

This is crucial for using input representations: the user has to know enough about the system to understand what will be represented by a given action. To know what key to press, I have to know what pressing a given key will mean. Early computer systems were weak in this respect. They relied on users to understand

what their inputs would mean, based only on weak cues like order of input: the first thing you type will be a command, the next thing the name of a file, and so on. Current designs provide output representations that describe what associated input representations will mean. For example, a dialog box like this one

Do you want to save your changes? Yes [ ] No [ ]

represents for the *user* what two actions (clicking the box labeled Yes, and clicking the box labeled No) will represent to the *computer*.

This situation involves an interesting combination of human and machine roles. In order for the user to create an input representation on which the machine will operate, the machine has to create an output representation on which the user can operate.

4.2.4 HCC and social systems

Humans are social animals: most things that people do are done in *groups*. This influences human-centered computing in a number of ways.

There are usually multiple people in a computational system. The designers of such systems therefore have to understand not just what individual people are likely to do, and can do, but what people working in groups will and can do. For example, sometimes systems fail because some users don't do things, like entering information into a data base, that are needed to support other users. But sometimes, as in the Wikipedia, people working strictly as volunteers put huge amounts of effort into really useful contributions. If you are designing a system for a lot of people to use, you have to try to understand what shapes these differences.

Decisions about what a computational system should represent has to reflect the needs and wants of groups of people, not just those of individuals. Historically, before computers became cheap enough for individuals to buy them, nearly all computational systems were created to meet the needs of organizations, especially businesses. Today, even though it is common for individuals to own one or more computers, facilities for communicating with other people, especially via the internet, are used almost all the time. People need to communicate, in many situations, and want to communicate in many others. The design of future computational systems will continue to be shaped strongly by these needs and wants. Think of MySpace, or FaceBook.

4.2.5 HCC and particular user groups

People have different capabilities. Because computational systems are almost always shaped by users' capabilities, differences in capabilities need to be reflected in different designs. ***Assistive technology*** uses representations that are suited to the needs of people with limited vision, or limited ability to read text, to mention just two examples. ***Universal design*** seeks to create representational systems that can be used by people with the widest possible range of capabilities.

4.4 Mappings implemented by computer

We've seen that some operations in the use of representations, such as comparing the lengths of bars in a bar chart, are carried out by people. We've also seen, though we haven't looked at the details, that the representations on which people operate may be created by computer. Let's extend our fuel economy example to bring out more fully what the computer can do.

Let's assume that the subject domain includes a collection of cars, as before, but that there is also a collection of ***trips***, one for each car. For each trip we've kept track of distance traveled and the fuel used. Our plan is to use this information in comparing the fuel economy of the cars.

To process the information about the trips we associate their lengths and fuel consumed with numbers; this is a familiar measurement idea. We'll then do something new: we'll ***calculate***, or ***compute***, the quotient of the distance traveled trip and the fuel consumed, for each trip. The resulting number is a measure of fuel economy, and we'll use this to create a bar with corresponding length for the car that made each trip.

So now we have two mappings, the calculation of miles per gallon, and the creation of a bar from each mileage we calculate, that we want the *computer* to carry out. To do this, these mappings have to be ***implemented***. That is, we don't want just abstract ideas, we want something to *happen*. Those bars have to actually appear on our screen, as patterns of glowing dots, say.

4.4.1 What it means for a computer to implement a mapping

As we've seen, an implementation of a mapping is a thing with two ends, an input end and an output end. In the case of computer implementation, the thing will be a technical device of some kind. For example, an ***adder*** is a device whose input end can be set to correspond to a pair of numbers, say 2 and 3. After a little delay, the adder sets its output end to correspond to the sum of the two numbers on the input end, 5 in this case. Any computer will have one or more adders in it.

Similarly, to draw a bar chart we will want a device of some kind whose input end can be set to represent a length, and whose output end will produce a colored bar. In practice, you won't find a bar-drawer as a separate device inside a computer. Rather, simpler devices are hooked together so as to act like a bar-drawer when needed.

From the point of view of creating a representation system, we don't care very much about the nature of the devices that implement the needed mappings, as long as they work. That's part of the basis of ***interchangeability***, one of the virtues of representational systems that we've discussed. It means that if someone invents a better way to draw bars, we can incorporate the new idea without changing how we do everything else.

Communication and Storage as Implemented Mappings

Many of the mappings one thinks of are like addition, in that the output is different from the input. But there are two very useful families of implementations of the ***identity*** mapping, the mapping whose output is the same as its input. Because the ends of a physical device can be separated in space, we can create a ***communication*** system, whose input end is in New York and whose output end is in Singapore. Here it is an advantage that the output is identical to the input.

Similarly, the ends of a device can be separated in *time*. A device whose input end is in September, 2006, but whose output end is in September, 2016, can be a ***storage*** system. Again, we are happy if the output is the same as the input.

4.4.2 Human-Centered Shapes and Sizes

We've just said that we often don't care how the devices work that implement mappings in a representational system. But we often care a lot about how big they are, and what their shape is, for human-centered reasons. ***Mobile devices*** are useful for many purposes, and good mobility means they have to be carried by people. Being easily carried means more than being light enough and small enough; ***wearable devices*** are shaped by the need to carry the device on the body in such a way that it is always available for use (and doesn't have to be extracted from a pocket.) A wearable device for a human is different from a wearable device for a dog.

4.4.3 Unintentional Communication

In most interactions with computers, people's actions are deliberate, and intended to represent information or intentions. But advances in the development

of small, readily deployed ***sensors*** has made possible communication in which people convey information to a computer just by performing ordinary actions with no intent to communicate. For example, when a person walks around a room that has a motion detector, they are conveying information to a computer about their presence, or their activity level.

Even though a person in this situation is not deliberately creating a representation of anything, the logic of representation is still relevant to design. The challenge for the designer of a system for sensing people's activities is create a useful correspondence between the representations in the machine and the things of interest in the subject domain, which in this case is the domain of people in the room, where they are, and what they are doing.

4.5 How to Create Computer Implementations

As we've argued, a computation is an implemented mapping, that is, a process embodied in a physical system. One could imagine that a machine to solve a given problem would have to be created using physical parts and physical tools, like wires and gears, and soldering irons and wrenches. In fact, much of the usefulness of computers in representation systems comes from three facts:

Configurability. It's possible to build a system in which the particular mappings that are implemented are specified by a ***configuration***: changes to part of the system that specify what mappings the system as a whole implements. The changes that make up a configuration have to be reversible, so that different configurations can be used at different times, and they have to be easy to make, say by throwing a switch rather than soldering something. Configurability means that you don't have to keep building new computational systems every time you have a new problem. Rather, you can just set up a new configuration.

Universality. One might think that any given computational system, even if configurable, could only be used for some particular class of problem. To some extent this is true: a system that doesn't have a color display can't be reconfigured to present color pictures. But to a remarkable extent a computational system that has some basic facilities can be configured to implement a very wide range of mappings. Some examples: a system that can implement addition can be configured to implement multiplication; a system that can represent mappings on whole numbers can be configured to represent operations on fractional numbers; a system that can represent whole numbers can produce a representation of a color picture that could be shown on a color display. This kind of flexibility is called ***universality***.

Self-configuration. While configuration can be done by a user doing something like throwing switches, it is possible, and very useful, to create systems that can *control their own configurations*. There are two parts to this. First, there needs to

be a representation of configurations that the system itself can manipulate. Second, there has to be way for the system to impose one of these configurations on its own operation, that is, to change its own configuration. These two capabilities mean that a user of the system doesn't always have to understand the configuration process in order to apply the system to a problem. Instead, the user can provide a representation of some kind that the system itself processes in such a way as to produce a configuration for a solution to the problem. The system can then impose this configuration on itself, and solve the problem.

How do configurability, universality, and self-configuration relate to layering, generality, and interchangeability?

These are two lists of virtues that explain much of the power of computer systems, that is, implemented representational systems, in our lives. Configurability, universality, and self-configuration, the virtues just presented, are important facts about *implementations*, facts that make computers cheap rather than expensive. Because of them, you don't have to buy a new computer every time you face a new problem, and, equally important, you don't have to design and build a new computer every time you have a new problem. Your old one will usually work fine.

The other three virtues are facts about the logic of representations themselves. They make it possible for representations to be thought of, with our limited human minds, and they have value even when the mappings involved can't be implemented. Just as we don't need a new computer for every new problem, we usually don't need a whole new mathematics (or a whole new conceptual system of some other kind) for every new problem. Familiar mathematics will do for lots of problems. Given how hard it is for us to create and learn mathematics, that's a really good thing.

4.5.1 Programs and Programming Languages

Representations of configurations, and representations from which configurations are produced, are both called ***programs***. Those representations of configurations that the system can impose on itself are called ***machine language programs***, and those representations provided by users, from which machine language programs are produced, are called ***higher level programs***. A notation used to express higher level programs is called a ***higher level programming language.***

Why not just use machine language programs to configure computational systems? There are two key advantages of higher level programs. First, it is

possible to use the same higher level program to control quite different computational systems. There are programs created in the 1960's to control the computers of that era that can still be used today to control the very different computers we have now. This results in a huge saving of work, since all the programs in the world do not have to recreated every time there is a improvement in computers. Second,and even more important, the descriptions of configurations that computers can impose on themselves, machine language programs, aren't easy *for people* to understand and manipulate. Higher level programs can be much easier for people to work with. For example, here is a tiny machine language program written for the IBM 7090 (about 1959), and then a corresponding program, expressed in FORTRAN, one of the earliest higher level programming languages.

```
0500 00 0 1000
0400 00 0 1001
0601 00 0 1002
```

```
A=B+C
```

Both programs add two numbers and store the result for future use. Comparing the programs you may be able to see that the labor required to produce a program in FORTRAN, especially a complicated one, is vastly less than for a corresponding machine language program.

4.5.2 HCC and programming

Because the point of a higher level program is that it should be easier *for people* to understand and work with, the design of systems for representing and manipulating these representations of configurations is inherently human centered. Just as the design of something like a bar chart has to be based on facts about human perceptual abilities, so the design of a higher level programming language has to be based on the abilities of humans to carry out necessary operations with and on the language. What are some of these operations?

Creating representations of things and operations in the subject domain. Programs have to create and manipulate things like collections of bars.

Problem decomposition. Representation systems nearly always have lots of collections and things, and lots of mappings. It's essential to build up a system like this piece by piece, not all in one big tangle.

Understanding what a program represents. Once a program has been created, it's important that the person who created it, or other people, be able to tell what it does and how it works. This is needed if some change is required, which almost always happens (such as adding cars to the collection whose economy is to be compared, to take an easy case.)

Modifying a program. When a change is needed, it should be easy to make it.

These operations will rely on people's ability to reason about complicated things and situations. At the same time, the design should as much as possible steer clear of weaknesses people have, such as difficulty in discriminating things with closely similar meanings or appearance, or keeping track of things that have no clear meaning. Unfortunately, programming languages often require just these kinds of efforts, such as distinguishing commas from semicolons, or managing different uses of equals signs, or putting in punctuation that's needed for machine processing of the program, but not for human comprehension of it.

4.5.3 The two faces of representations in computational systems

A higher level program gets used in two ways. First, it is operated on (and usually created) by people. As we've just seen, this requirement has to shape its design. But second, it is operated on by the computational system so as to produce a representation of a configuration that can be imposed on the system's physical structure, that is, a machine language program. This requirement, too, shapes its design. Thus the higher level program faces two ways: towards the people who create it, and perhaps must understand and modify it, and towards the machine configuration it must be converted to. In case of conflict it can't lean too far in either direction.

Because people are more flexible than machines, the design of higher level languages, while clearly motivated by human uses, leaned strongly toward the machine. As mentioned just now, people are terrible at managing fine distinctions in meaning, and in following conventions that aren't well motivated for them. But higher level programming languages are full of requirements of this kind, because they simplify the conversion of higher level programs to machine language programs.

This happens partly because the simple mechanics of translation are easier for languages that have simple grammar rules, and for which the meaning of a collection of parts is easily determined from the meanings of the parts. Natural human communication is not like this; the meaning of almost anything people say or understand is strongly and subtly influenced by context. Many sentences containing the word “dog” have nothing to do with dogs, for example “Give me a dog with mustard”.

A further complication arises not from the translation process itself but from the problem of describing machine configurations in such a way that they correspond to useful computations. For example, computations very often involve operating on collections of things, as we've seen. But configuring a physical system so as to represent these collections and operations on them is not easy. People have

developed different ways of doing this, and the higher level language features used with these different ways are different. FORTRAN used (and uses) consecutive blocks of bits in memory, called arrays, and used numbers to select a given item in the collection. Processing a collection was done by changing the numbers used for selection so as to carry out an operation successively on all the items in the array. LISP, another pioneering higher level language, used (and uses) linked lists, with ways of referring to the first element in a list, or to the list formed by all the elements except the first. Processing a collection is done by recursion: defining what to do with the first element of list, and how to process the remaining elements, including the treatment of an empty list. Both of these ways of working with collections works quite well, but they are different. Thus user of higher level languages are asked to deal with different ways of representing similar or identical ideas, and with the different language features provided for working with them.

5. So, What is Human-Centered Computing?

Computational systems are representational systems. Representational systems substitute operations in a representation domain for operations in a subject domain. To work properly, the representation domain has to reflect the structure of the subject domain, in the sense captured by category theory. Also, it has to be possible to implement the operations in the representation domain.

HCC deals with representations to be used or created by people, and creates systems in which some operations are carried out by people, singly or in groups, as well as by computers. In these situations design is shaped by insight into what people can do, physically or mentally, easily and accurately, and into people’s wants and needs, including the capabilities, tendencies, wants, and needs of people acting in groups and organizations. Here’s a tabular summary of these ideas:

Computational systems are representational systems.
Representational systems can be defined formally.
Representational systems are defined by the mappings they support.
The mappings should support the wants and needs of users, as individuals or in groups and organizations.
Computational systems include machines and people as implementing agents.
Implementation includes producing representations and carrying out mappings.
The demands of implementation have to be matched to the capabilities of machines, people, and groups of people, and what they naturally do.

Historically, work in computing was machine centered rather than human centered, for quite a long time.. This made perfect sense: the technical challenges in creating machine implementations of mappings, and the associated high cost and limited performance, justified the development of mixed human-machine systems in which the humans had to do a good deal of difficult, tedious, and error-prone work, using the primitive representations that were all that were available in the machine technology of the era.

Today, enormous advances in machine technology have made it possible to create more balanced systems, in which machine effort supports the use of representations that are much easier for people to work with. As a result, we can see human-centered computing not as an island off the shore of computing, as it seemed for a long time, but rather as a large province, along with machine-centered computing, of the country of computing.

The two provinces, machine-centered and human-centered are closely connected by the logic of representations. What makes representation systems work, whether their mappings are implemented by machine or human, or some of each, is the correspondence between the mappings in their subject and representation domains, the accuracy and speed of the implementations (whether based on human capacities or technology), and the value of the mappings in the subject domain.

APPENDIX: More about category theory.

We can't do more than introduce some of the ideas of category theory here. If you are pretty comfortable with math, you may find the paper "When is one thing equal to some other thing?", by Barry Mazur (available at http://www.math.harvard.edu/~mazur/), helpful and interesting. If you are less comfortable, the book *Conceptual Mathematics: A First Introduction to Categories*, by F. William Lawvere and Stephen Schanuel, may be a better place to start.

To push a little farther in these notes, let's set two goals. First, let's develop the fuel economy example in a way that sets up everything we'll need in the category theory treatment. As you'll see, this involves a couple of changes from the informal presentation in the body of the notes. Second, let's use that setup to justify the key claim made in the notes, that any legal functor will produce a working representational system, in that the operations in the representation category produce valid answers.

The basic setup for the fuel economy example will be two categories, the subject category, containing cars and operations on them, and the representation category containing bars and operations on them. So the subject category will have a collection of cars in it, and the representation category will have a collection of bars. (Collections like these are called ***objects*** in category theory, as you'll find if you seek out further readings. But this is confusing to people who know about objects in Computer Science, so I'm not using the term here.)

You may want to look at the picture below, which shows the finished set up, as you read along. You can see the two categories, and the collections of cars and bars, in the picture.

The operations we find in the example, comparing cars and comparing bars, both act on pairs of things, not individuals, so we'll need to add another collection to each category: a collection of pairs of cars, in the subject category, and a collection of pairs of bars, in the representation category. What makes these collections collections of pairs is that there are two mappings from each collection of pairs, `first` and `second`. When you apply `first` to a pair you get the first thing in the pair as your result, and when you apply `second` you get the second thing. Thus if you have a pair of a Chevy and a Ford, `first` gives the Chevy and `second` gives the Ford.

Looking at the picture, you can see the collections of pairs of cars and pairs of bars. You can also see the mappings, `first` and `second`, that connect these collections to the collections of cars and bars. (Actually, the two mappings I've called `first` are two *different* mappings: one connects pairs of cars to cars, and

the other connects pairs of bars to bars. So I shouldn't really give them the same name. But using the same name for both makes the picture simpler, as long as you don't get fooled into thinking the two are the same mapping.)

Because we're going to be asking whether the first car in a pair is more economical than the second, we need to be able represent the answers we can get, Yes or No. So we'll add a collection with Yes and No in it to the subject category. We'll be comparing bars in the representation category, so we'll add a similar collection to the representation category. You can see these collections in the picture.

Now that we have a collection of pairs of cars, and a collection of possible answers, we can put in a mapping for the operation of comparing two cars. In the picture it's the arrow from the collection of pairs of cars to the collection of answers, labelled "is more economical than?". Similarly, we put in a mapping for comparing the lengths of bars.

Because we need to have bars correspond to cars, we'll also need to be able pick out individual cars or bars from the collections in the two categories. We'll need to be able to pick out individual pairs from the collections of pairs, too. To do these things we need to have a ***terminal object*** in each category. You can think of a terminal object as a collection with just one single thing in it, so that any mapping from it into any another collection picks out just one thing. (Think about it: if a mapping has only one possible input, it has only one possible output.) In the picture the terminal objects are shown as collections with just 1 in them (it doesn't matter what is in a terminal object as long as there's just one thing.)

Once we have terminal objects in both of our categories, we'll put in a whole bunch of mappings from them to the other collections in each category. In fact, for every individual car, we'll have a mapping from the terminal object to the collection of cars that picks out just that car. Similarly, for every pair of cars we'll have a mapping from the terminal object to the collection of pairs of cars that picks out that pair. We'll do the same for the answers (in both categories), the bars, and the pairs of cars.

If we drew in the arrows for all of these mappings the whole picture would be covered. So the picture just shows a few arrows coming from each terminal object. Keep in mind that there's actually one of those arrows for each of the individual things in any of the collections in each category.

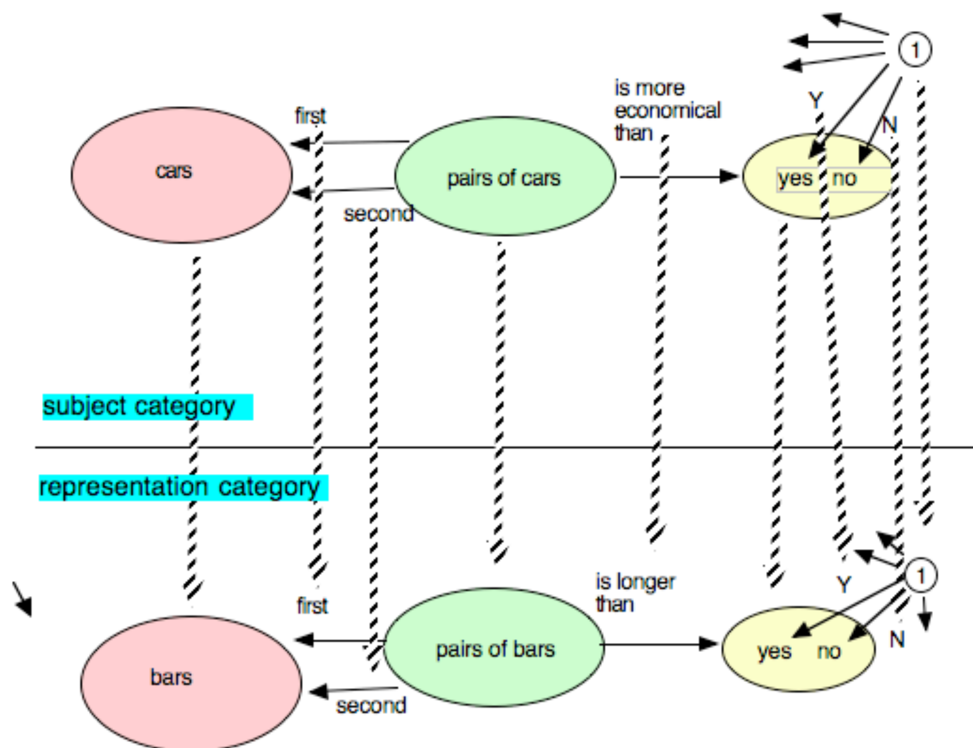
There are some other arrows not shown in the picture. Category theory requires that you can compose any two mappings, if the output collection for the first mapping is the input collection for the second mapping. There are a lot of these compositions in our categories, and some will play a part in our analysis later on, but they aren't shown, to cut down on clutter. Finally, for each collection there is an ***identity mapping*** that connects that collection with itself. For each input the

identity mapping gives the same thing as its output. The arrows for these identity mappings are also not shown.

The final piece of the setup is a ***functor*** connecting the two categories. For each collection in the subject category the functor assigns a corresponding collection in the representation category. These correspondences are shown as shaded arrows between the categories. Also, for each mapping (arrow) in the subject category, the functor assigns an arrow in the representation category. To avoid clutter, only a few of these correspondences are shown.

On the right hand side of the categories notice that a few of the mappings from the terminal objects are filled in, and the correspondence assigned by the functor is shown. These show that we are assuming that Yes in the representation category corresponds to Yes in the subject category, and the same for No. We wouldn't have to do this, but if we allow No to correspond to Yes we're likely to get confused in interpreting the answers we get when we compare two bars.

Here is the completed setup:



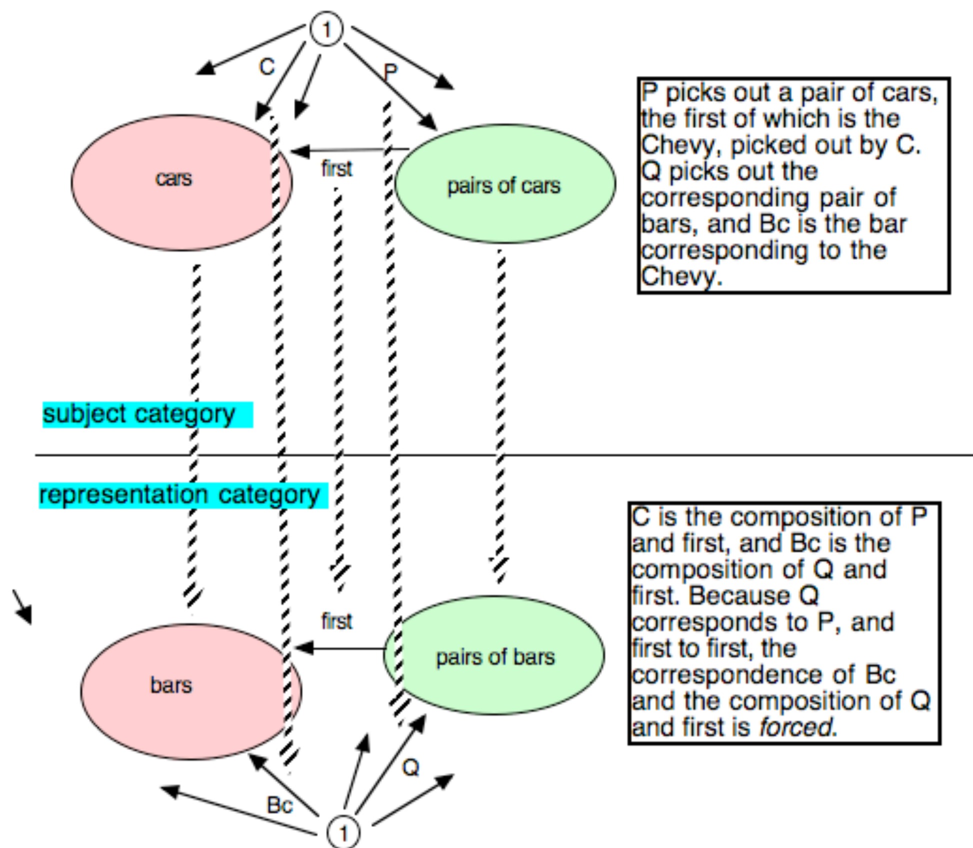
Now we turn to determining whether the setup gives us a working representation system, in the sense that comparing two bars can substitute for comparing two cars. Our analysis will hinge on the fact that a legal functor can't assign correspondences between the two categories in just any old way, but must

satisfy two conditions. First, it has to preserve identities. For example, since the collections of bars corresponds to the collection of cars, the functor has to make the identity mapping on bars correspond to the identity mapping on cars. We'll assume that happens.

More important, as mentioned in the body of the notes, the functor has to preserve composition. That means, as described earlier, that what the mapping the functor makes correspond to the composition of two mappings is *forced* by the assignments it made for those mappings.

Now we're ready to ask the big question. If we compare a pair of cars, and we compare the corresponding pair of cars, will we get the same answer?

The first thing we need to check is that when we pick out a pair of cars, the corresponding pair of bars contains the bars that correspond to those two cars. This picture focuses on the relevant part of the setup, and adds some mappings we'll be discussing:



Something you may find hard to follow in the argument is that in category theory we only work with mappings, not with things. We can get away with this because

each thing corresponds to a mapping from the terminal object to the collection the thing is in. So we use that mapping instead of the thing.

Let's say we have a pair that consists of the Chevy and the Ford. Let's call the mapping from the terminal object to collection of pairs of cars that picks out this pair P. Let's call the mapping from the terminal object to the collection of cars that picks out the Chevy C, and the mapping that picks out the Ford F. There's some bar that corresponds to the Chevy, and so there's a mapping from the terminal object to the collection of bars that picks it out. Let's call that mapping Bc, and the mapping that picks out the bar that corresponds to the Ford Bf. Finally, there's some pair of bars that corresponds to the pair of the Chevy and the Ford, and let's call the mapping that picks this out Q.

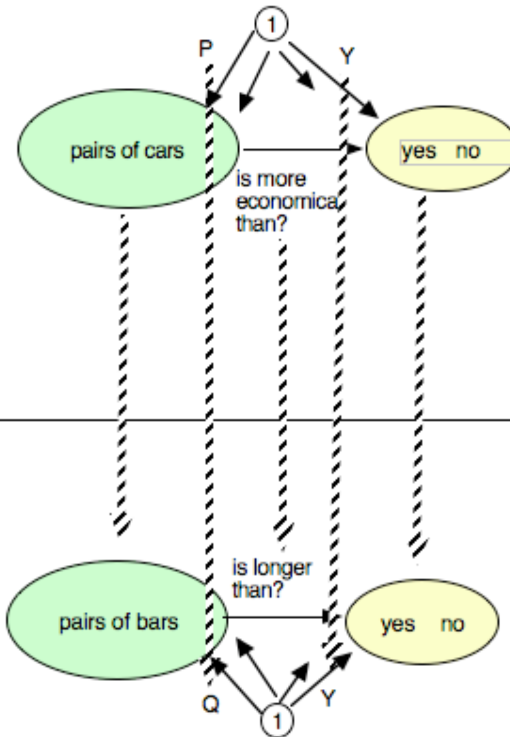
We can now state our question: does the pair of bars that Q picks out contain the bars that correspond to the Chevy and the Ford? If it doesn't, we're in trouble.

Let's just consider the first bar in the pair. We can get this by composing `first` with Q: this will give us a mapping from the terminal object to the collection of bars. We're hoping that this mapping will turn out to the same as Bc, because that picks out the bar that corresponds to the Chevy, and the Chevy is the first car in the pair of cars we started with.

We can show that any legal functor in our setup has to give us this result, as follows. Consider the mappings P and `first`, in the subject category. When we compose them we get C: that's what we mean by saying the first car in the pair is the Chevy. We've said that the functor assigns Q to P, and `first` to `first`. Then because it has to preserve composition, it is forced to assign the composition of Q and `first` to C. We've already said that it assigned Bc to C, so the composition of Q and first must be Bc, just as we hoped.

We can make a similar argument that shows that the second bar in the pair of bars is forced to correspond to the Ford. So any legal functor in our setup has to keep the cars and bars in corresponding pairs lined up.

Now, what happens when we compare cars or bars? Let's say that the Chevy is more economical than the Ford, so that the answer to our comparison is Yes. The question is, what answer will we get when we apply the mapping "is longer than?" to Q? Here's a picture that focuses on the relevant part of the setup.



subject category

representation category

Here P picks out a pair of cars, and Q picks out the corresponding pair of bars. The mapping "is longer than?" corresponds to "is more econominal than?". Then the composition of Q and "is longer than" must be Y, since that corrsponds to Y in the subject category, the composition of P and "is more econonmical than?"

The fact that comparing the pair of cars gave the answer Yes means that the composition of "is more economical than?" and P is the mapping Y, that picks out Yes. We know that the functor assigns Q to P, and "is longer than?" to "is more economical than?" So it is forced to assign the composition of Q and "is longer than?" to Y. But in our setup we required the functor to assign Y (in the representation category) to Y (in the subject category). So the composition of Q and "is longer than" must be Y, as we want.

All of this will likely seem like heavy going the first few times you work through it. And anyway, what's the payoff from all this argumentation? It's the fact that working representation systems are defined by legal functors. It might seem that the requirement of substitutability of operations in the representation domain for operations in the subject domain would require stronger and more complicated conditions that just preserving composition, but in fact it doesn't.

Then again, it may be that instead of being surprised at how little has to be required to guarantee a working representation system, you're surprised at how *much* has to be said. Isn't it obvious that you just have to match up the bars and the cars? In a simple case like this, it is pretty obvious. But if you think back to the length example, you probably didn't find it obvious that you can match up beams and numbers in such a way that you determine the lengths of beams using multiplication rather than addition. And how about using subtraction? The analysis we've gone through here can be applied to that more complex situation, as well as to the simple one we've examined here.

Actually, there's another un-obvious problem about representations that our analysis may clarify, though it doesn't solve it. We've shown that if we can create a legal functor connecting our subject and representation categories, then we have a working representation system. But how do we know there *is* a legal functor? How do we know that either addition or multiplication will actually work to predict the lengths of connected beams in the real, physical world? This is a question debated by philosophers, mathematicians, and physicists... how is it that the abstract things of mathematics fit the physical things of the world? Category theory doesn't tell us, but it does tell us what "fit" is.

There may be another payoff from understanding the category theory view of representations. While using mappings from the terminal object instead of talking about things may seem odd and unnatural, there's some conceptual benefit. As we'll discuss further below, things on their own can't represent anything: they only work as representations in context. A mapping, unlike a thing, carries some context with it: it specifies what collection it maps into. Using a mapping from a terminal object, rather than a thing, makes clear that to represent a thing you have to be able to *pick it out from a collection of similar things*, and that you have to know *what that collection of similar things is* in order to do that. A number *on its own* can't represent a length, but a mapping that picks out one number from a collection of numbers that represent lengths can. And, once you know that you've got a collection of numbers that represent lengths, that lone number suffices to pick out one length from the collection. That explains why, even though a lone number can't represent a length, or anything else, it can do the job when interpreted in the right way, that is, as a mapping.