First class representations (very early draft)

Patrick Prémont

April 10, 1996

1 Motivation for first class representations

The concept of a representation is one which is used abundantly in the design of computing systems. It tends, however, to be seen as something to be carefully considered and then fixed at the time the system is designed. This static view of representations imposes limitations similar to the limitations introduced by other static concepts in computing systems like static types, static definition of functions, static memory allocation, ... There are existing systems where representations can change dynamically but to my knowledge this is done in a very restricted way which allows, at best, to dynamically select a representation among a small statically determined set of representations. A system that allows dynamic creation and use of representations should prove considerably more powerful and provide remarkable capacities for reflection. A natural and versatile way of achieving this goal is to make representations first class objects in the language used to implement the system. The representations may then be constructed, manipulated and used with the whole power of the language.

2 Formalization of representations

Representing an object is writing information in some language so that a reader will be able to distinguish the signified object from the other members of the set of objects which could have been represented.

The reader and the writer must share the same set of objects which could be represented, we will call this set the type. They must also share the same language used for representation. Finally, they must share the same storage space which can be characterised by the set of states the it may be in. The language defines a bijection between the type and a partition of the states in the storage space. The states are grouped into equivalence classes because in many cases more than one state may represent the same type element. But the particular state in the class used to represent the element may still be relevant to some other representation occurring in the same storage space.

Let us define a formalism to talk about representations: If R is a representation, $F(R)$ is the associated bijection, $T(R)$ is the associated type and $S(R)$ is the associated state space. We can write a representation R as trio $\langle T(R), F(R), S(R)\rangle$.

Example :

Let's say the set of states is $Z = \{0000, 0001, 0010, \ldots, 1111\}$.
Let's have two types $A = \{a, b, c, d\}$ and $E = \{e, f, g, h\}$.
Let $R_1 = \langle A, f_1, Z\rangle$ where we will define f_1 to be :

$$\begin{aligned}
f_1(a) &= \{0000, 0001, 0010, 0011\} \\
f_1(b) &= \{0100, 0101, 0110, 0111\} \\
f_1(c) &= \{1000, 1001, 1010, 1011\} \\
f_1(d) &= \{1100, 1101, 1110, 1111\}
\end{aligned}$$

So to represent c with R_1, the state can be 1000, 1001, 1010 or 1011.
But we may want to represent something else :
let's say h with $R2 = \langle E, f_2, Z\rangle$ where f_2 is defined as :

$$f_2(e) = \{0000, 0100, 1000, 1100\}$$
$$f_2(f) = \{0001, 0101, 1001, 1101\}$$
$$f_2(g) = \{0010, 0110, 1010, 1110\}$$
$$f_2(h) = \{0011, 0111, 1011, 1111\}$$

So the state that represents both c with R_1 and h with R_2 is 1011.

The three components of representations are completely determined by the bijection. The type is the domain of the bijection and the state space is the union of the sets in the range of the bijection. So we could define representations as simply being bijections. I have not done that yet but I may choose to eventually. I don't think it matters since there is an obvious equivalence between the two options.

There is another interesting class of representations which require a slightly different approach. Let us call the previous representations exact representations. Inexact representations have a bijection not between type and partition of states but between a partition of the type and a partition of states. This means that many members of the type may be represented by the same states.

There may also be things which could be called representations which are neither exact nor inexact according to the previous definitions. Inexact representations can model some lossy representations but there may be some where the writing maps a subset C of the type to an equivalence class of states and the reading maps this class back to a subset of the type which is different but close to C. We will not consider these kinds of "representations".

Many representations may or may not be satisfiable concurrently. They are if and only if representing any combination of members of the types of each representation is possible (there is a state contained in all the equivalence classes (one class for each representation, the class corresponding to the element being represented)). This is difficult to express clearly in English because there is a variable number of quantifiers (one for each representation). It can be expressed formally as follows : if $R_1, R_2, \ldots, R_n$ are representations, they are concurrently satisfiable if and only if

$$\forall x_1 \in T(R_1)\ \forall x_2 \in T(R_2)\ \ldots\ \forall x_n \in T(R_n)\ \left[\bigcap_{1\le i\le n} F(R_i)(x_i)\right] \neq \{\}$$

For some set of representations, verifying that all pairs of representations from the set are concurrently satisfiable is *not* sufficient to conclude that all representations in the set are concurrently satisfiable. Here is a problematic example :

Let $Z = \{00, 01, 10, 11\}$ and $A = \{a, b\}$.
Consider the set S of representations $\{\langle A, f_1, Z\rangle, \langle A, f_2, Z\rangle, \langle A, f_3, Z\rangle\}$ where

$$\begin{aligned} f_1(a) &= \{00, 01\} \\ f_1(b) &= \{10, 11\} \\ f_2(a) &= \{00, 10\} \\ f_2(b) &= \{01, 11\} \\ f_3(a) &= \{00, 11\} \\ f_3(b) &= \{10, 01\} \end{aligned}$$

It is easy to see that the three representations cannot be concurrently satisfiable since there are not enough states in Z (2^3). However every pair of representations in the set is concurrently satisfiable.

2.1 A few implementation issues and problems

The requirement of concurrent satisfiability may be too strong in some circumstances. If there are constraints on the objects to be represented, like for example if we wish to represent a, b and $a + b$, then it is sufficient to verify that the representations are concurrently satisfiable under the constraints. This means that there are no representability requirements on object combinations which do not satisfy the constraints. If the predicate C characterizes the satisfaction of the constraints, concurrent satisfiability under the constraints becomes :

$$\forall x_1 \in T(R_1)\ \ \forall x_2 \in T(R_2)\ \ldots\ \forall x_n \in T(R_n)$$

$$C(x_1, x_2, \ldots, x_n) \Rightarrow \left[\bigcap_{1\le i\le n} F(R_i)(x_i)\right] \neq \{\}$$

In a computing system, to ensure that they are concurrently satisfiable, representations will have to be carefully selected. In current systems, this problem is solved by memory managers. They proceed by allocating disjoint memory regions to each representation. For a set of representations selected using this method, if every pair of representations in the set are concurrently satisfiable, so is the whole set. However, in some cases, it may be beneficial to consider representations which do not have exclusive access to their memory region. For example, if a is represented in some region and b in another, a representation of $a + b$ may be considered that would use the union of the two memory regions.

Can we represent infinite types knowing that the set of states is finite ? Not exactly but it is possible to do that inexactly.

What about representing integers without bounds ? Some programming languages allow that. But these languages allow operations to fail because of a lack of memory. Is this acceptable ? There are many other infinite types: trees, graphs,... Can we be expected to represent only finite subtypes ? Or do we allow them and find a way to deal with the out-of-memory problem ? Can the problem be dealt with before the computation is attempted, by a proof that it will not occur ? It seems this would be similar to bounding the types. Or is there no alternative but to fail when there is no memory left ? In that case, how does the system go on gracefully ?

How can we prove the correctness/safety of programs which use inexact representations ? This seems to imply being able to bound every floating-point number in an interval which may get very big after many operations. Or more generally, determine all possible values. This produces a combinatorial explosion which is not present in deterministic operations on exact representations. But then the explosion also occurs in nondeterministic operations on exact representations.

3 Operations

An operation needs to read arguments and write results. When input is read, it must know which representation is used. This must be achieved by a system which ensures compatibility with the operations which have previously written what is now being read.

But if representations are fixed in that respect, how do we allow for dynamic representations ? Dynamically creating operations with the desired argument representations is one way but it can be costly and can require a lot of management. A solution which allows a single statically defined operation to accept and produce dynamically determined representations would be preferable in many circumstances.

Many languages allow polymorphic functions which have input arguments of a type which is not fixed. In that case the type is just another input which is read before reading the argument and the representation of that type is known statically (in most cases, it is constant and global to the system).

We can get our solution by making the concept of polymorphic functions more general so that these functions are allowed not only to accept arguments of variable types but of variable representations. This means that the actual representation used becomes another argument which must be read first. Note that according to our definition of a representation, it includes the type so we are replacing the type argument by a representation argument. The representation of this representation argument can either be fixed of be passed previously. This process may be repeated any number of times but the operation must know the representation of the first thing it reads. We can still have an arbitrarily long chain of representations of representations by passing first an integer which gives the length of that chain. The representation of that integer must be known however. Again, it may be fixed or passed previously ...

We can safely pass representations as arguments since they are first class objects. But to do that, representations, as objects, must be represented.

This is where the formalization of representations which was done previously finds its application. We have identified three components of representations : a type, a bijection and a state space. As with everything else, we can represent the representations by representing their components. We will want first-class types also so finding a way (or many ways, remember we have first-class representations) to represent types is something we needed to do anyway. First class bijections would also be useful since, for example, we may want to apply operations like composition on them to get new bijections. First-class state spaces would be useful in a system where there are many such spaces. And that is the case in most systems which include video memory, hard drive space, main memory, caches, paper (through printing/scanning) or more generally, the physical world (through motors/senses).

Primitive operations like machine instructions will tend to assume fixed input and output representations. We will call these representations primitive. They will act, together with the primitive operations, as foundations upon which more representations and operations may be built. To do that, it will be necessary to be able to represent primitive representations, for example, by associating them to integers if integers have a primitive representation. The primitive representations will really be parametrized representations, to be discussed later.

For a representation to be of any use, operations must be defined that allow to perform computations on the elements of the representation's type.

**** The following paragraph is outdated and needs some rethinking and rewriting (that does not mean that I consider everything else to be immune from major changes) ****

There is a cyclical dependency problem in representing representations. A representation contains a type. So we will want to represent the type to represent the representation (R). But then we cannot represent the type explicitly (by listing its elements) because representing the elements requires R or another representation. So either we have dependency cycles or there is a representation whose type's elements are not represented. This makes sense because before the first representation for a type is created, there is no way of talking about the individual elements. So the type should be represented by what is know about it, like the fact that it is different (a new-object, maybe the same contents) from all other types (can be done by giving it a unique integer ID, its address for example). Its cardinality may also be known. It may even have a logical structure ...

4 Copying a type and creating a first representation with it

One can create the character type/representation from the unsigned 8 bit integer type/representation,

First we can create a new type Char.

Then we declare that there is a bijection between Char and the 8 bit unsigned integer type. At this point the system doesn't ask for what that bijection actually IS because there is no way of talking about Char's individual elements (because there is no representation with that type yet, but if there were, we couldn't just declare the bijection, we would have to define it).

The system automatically creates a representation with that type where the bijection is the one we have just declared composed with a bijection found in some representation of 8 bit unsigned integer. The effect is that we now have a representation for a character, we just represent it as a 8 bit number and treat it as a character. The bijection doesn't need to be represented since it hasn't been defined, just declared. The type is no more arbitrary, this should be represented (probably by referring to the representation(s) with that type).

Now the only way of talking about members of Char is by their corresponding number. So we could then create a routine to draw the characters but the only operation we could do on characters is to get the corresponding 8 bit unsigned integer (no translation machine code is required here).

5 Other type/representation constructors

One can create a new type which is the Cartesian product of n types. A default representation(s) with that type can then be created. Two options are : appending the representations or creating a linked list of the representations (if they are memory regions).

6 Parametrized representations

Often we wish to write many elements of some type. Each of these elements must have its own distinct representation if we want the representations to be concurrently satisfiable. When the state space is organized as a collection of identical but smaller state space cells (like memory words), a possible transformation on representations becomes obvious. The bijection of a representation can modified to map to other cells (since they are identical). If a representation used (I assume it is obvious what it means to "use" a cell) only some cells, then through this transformation, other versions may be created which use mutually exclusive subsets of the cells and are therefore concurrently satisfiable. From this we can create a parametrized representation. That is a function which takes some arguments and returns a representation. In this case, the arguments would be the cells used. Sufficient flexibility is often achieved with only one argument, an identifier for the first cell (an address). But of course in general the arguments need not be cell identifiers.

Parametrized representations are interesting because together with the arguments which are passed to it, they give a way of representing the representation returned by the parametrized representation when applied to these arguments. More concisely : a representation may be represented by a parametrized representation and arguments for it. Often this would mean that to express how something is represented you can give a start address and a description of the representation by referencing memory locations relatively to the start address.

7 A state space definition language

I will present a formalism the can be use to describe some state spaces. This description will suggest a representation for state spaces by representing the terms of the formalism.

Space(n) is a set of n elements. We will designate the elements by integers from 0 to $n-1$ but this is purely for identification; there are no operations defined on these identifiers. So Space(n) = $\{0, 1, \ldots, n-1\}$.

$$\text{Singleton} = \text{Space}(1) = \{0\}$$

With only one state, the current state contains no information.

$$\text{Bit} = \text{Space}(2) = \{0, 1\}$$

With two states, the current state can store up to one bit of information.

Space is sufficient to create unstructured state spaces but structured spaces are more interesting for representations. We will use set operators to build structured spaces :

Named$(N_1 : S_1, N_2 : S_2, \ldots, N_n : S_n)$ returns the Cartesian product S of S_1 to S_n. S can then be indexed by a name from N_1 to N_n to yield the corresponding set : $S.N_x = S_x$. Also, a state s in S can be indexed similarly to yield to corresponding component of the state ($s.N_x$ is the element of S_x that must be chosen to build s).

$$\text{Numbered}(n, S) = \text{Named}(0 : S, 1 : S, 2 : S, \ldots, (n-1) : S)$$

The names are integers on which operations may be performed.

A state space for a 386 machine with 16MB of RAM could then be formalised by :

$$\text{Machine} = \text{Named}(\text{mem} : \text{Numbered}(16 \times 2^{20}, \text{Byte}), \text{reg} : \text{Registers})$$

where

$$\begin{aligned}\text{Byte} &= \text{Numbered}(8, \text{Bit})\\ \text{Registers} &= \text{Named}(\text{eax} : \text{DWord}, \text{ebx} : \text{DWord}, \ldots)\\ \text{DWord} &= \text{Numbered}(4, \text{Byte})\end{aligned}$$

With this formalism we can now talk about (and represent) such spaces as :
Machine.mem.345.4 (bit 4 of byte 345 of main memory)
Machine.reg.ecx.0 (byte 0 of register ecx (cl))
Machine.reg.ecx.3.7 (bit 7 of word 3 of register ecx)

If the machine has more storage devices like caches, video memory, configuration EEPROMs, hard drives, ... larger state spaces may be described that include these memories.

We are now also able to describe basic representations for a computing system.

8 Some basic parametrized representations

Let $0, ..n$ be the type of integers from 0 to $n-1$.

Unsigned-8bit-integer(byte-address) $= \langle (0, ..2^8), f, Machine \rangle$
where $f(n) = \{s \in \text{Machine} | \forall m \in \{0, \ldots, 7\}. s.\text{byte-address}.m = \lfloor \frac{n}{2^m} \rfloor \bmod 2\}$

Unless we wish to prove the correctness of programs down to the semantics of the machine operations, we do not strictly need the system to understand the above description because the operations which operate on objects represented using it are the fixed machine instructions.

However, we want to be able to able to specify our own types and representations in terms of the basic ones. By formalizing how this can be done, we would be creating representations for types and representations.

9 Type and representation constructors

Building types from other types: (Let's assume that the types are disjoint. I'm not sure how to deal with non-disjoint types.)

Mirroring Creating a type who's elements are all new but mirror the elements of the old type. A bijection is available between the two types to define operations on the new type in term of the operations on the old one.

Union

Functions

Vectors/Lists Should be unified if semantically the same. (Ignoring operational aspects which belong in the representation)

Cartesian product Fixed length Vectors/Lists

Building representations from representations:

When a type constructor is applied to types to yield a new one, one or more representations with the new type may be created automatically from the representations with the old types. Also, operations should be created automatically to retrieve the constituents of the new type (but the creation may wait until the operation is needed, to not represent it if it is not used and to represent it with an appropriate representation otherwise.)

If the old representations use memory regions of fixed length, all the represented data may simply be placed in adjacent memory locations. If some are of variable length, length indicators may be added.

10 Representation of operations

Machine operations have their predefined representations. The representation of their inputs and outputs are also predefined.

**** I'm not as convinced as I used to be about everything that follows ****

Higher-level operations may be represented with sequences (usually) of machine operations. The representations of the inputs and outputs are part of the operation and not something to be decided internally in the operation as part of its representation. Different input and output representations yield different operations even if they implement the same input-type to output-type function. Deciding if timing is part of the operation or purely part of the representation (public or private) is something that may be done either or both ways, yielding different kinds of operations (public timing operations have much less freedom in deciding their internal representations because they must satisfy timing constraints). Another defining aspect of operations is the way in which control is passed to them (like the entry point). This can be considered to be another input.

One may want to have things which are similar to operations but which do not specify the representation of their I/O. Such a construction would be useful at even higher levels, where this kind of operations (let's call them functions or procedures) are seen as having input and outputs of a specified type but not representation. So these are not executable without help from an interpreter or a compiler which will pin down the I/O representations before execution.