The Specification, Design, and Implementation of an Abstract Processor

Daniel Davis - John Yurchak

* NPS, Code 52vv, Monterey, CA 93942
** VAW-123, FPO, New York, NY

Abstract

This paper describes the specification, design, and implementation of an abstract processor as the illustration of a methodology that describes computer resources in an implementation independent manner. The methodology uses algebraic semantics, but emphasizes practical over theoretical issues. This work seeks to develop a strategy for portability and reusability of computing resources through resource abstraction. This paper presents the definition of a hardware resource; the methodology is implementation independent, however, and equally capable of defining resources realized in hardware, firmware, or software.

Introduction

The basis of the strategy described here is that all computing systems provide, at essentially all levels and in many forms, a variety of resource abstractions. This term refers to the idea that computing resources at all levels are a priori mental concepts, that is, abstract constructions, that guide the way we configure actual physical resources.

For example, at a relatively primitive level, the instruction opcodes of a processor are abstractions of the effects of the circuitry realized by the underlying device. The form of the instructions is determined by the requirement that the physical device emulate various abstractly conceived operations. At a higher level, the service calls provided by an OS are further abstractions of the resources provided by the system, including processors, memory, and peripheral devices. At a still higher level, the linguistic features of a high level programming language are abstractions not primarily of lower-level resources, but constructs required to express an algorithm in some problem domain. There are physical resource abstractions and problem solving abstractions.

Abstraction is the process of separating the essential properties of an object from the nonessential. In particular, abstraction involves viewing something independently of its representations. Abstract objects are the natural objects of thought in general problem solving.

Essentially, we try to work with abstract objects, but are forced to use the objects that represent them. This dilemma is the source of most of our problems with portability and reusability of computing resources. We may have only a vague notion of the abstract objects we use or want to use in a given application, and even if we have a precise notion, our representations may not be true. A classic example in computing is, of course, the way we use mathematics. Mathematics provides a rich supply of abstract objects that we can use to solve practical problems, but it is difficult to accurately represent these objects in practice. We even have great difficulty in accurately representing the basic concept of a number. The result is usually some admixture of the abstract object and features of the mechanism used to represent it.

We will discuss here the problem of describing the abstract objects, that seem to arise naturally in computing, as "things in themselves", independent of their representation, and we will set up a method for doing this. The method is a result of work begun in Davis. We will illustrate the method by using it to design an "abstract processor." Then we will discuss an implementation of this processor as described in Yurchak.

Specifying Resources Abstractly

A major problem that we must solve is to find a unifying feature among all the different objects that we could view as a "computing resource." Such resources include the linguistic constructs of a high level language, the features of a processor chip, and the features of an operating system. Is there a single principle underlying all these when they are viewed as providing resources?

For this problem we basically adopt a functional view. We assume that every such resource can be described in functional terms. This assumption is our essential working hypothesis.

A consequence of this assumption is that to describe an abstract resource it is sufficient to describe its functions and their essential properties. Researchers who work on the theory of abstract data types make this same assumption. It is also the principle behind the concept of functional programming languages. We have found, however, that the objective of current research in this area differs somewhat from our objective, so that although we have borrowed from much of this work, we have also modified it to a more practical purpose.

A description of an abstract resource has two aspects: the "form" of the description, (its syntax), and the meaning to be associated to its form, (the semantics). Although syntax can vary and yet have the same semantics, we have tried here to choose a form that is simple, precise, and suggestive. The basic form is:

```
Resource (name) is
  Operand Types
    (operand types)
  Operators
    (operators)
  Properties
    (properties)
```
A name is an identifier for the resource. The operand types are names, separated by commas, for the types used by the functions (operators), and must be different from the name of any resource. The operators are the names of the functions provided by the resource, together with the types of its operands and result; again in a list. For example, a description commonly used in the theory of abstract data types is:

**Resource** Boolean is

**Operand Types**

| Boolean |

**Operators**

| Not: Boolean -> Boolean, And: Boolean, Boolean -> Boolean, True: -> Boolean, False: -> Boolean |

**Properties**

| Not(True()) = False(), Not(Not(X)) = X, And(True(), X) = X, And(False(), X) = False() |

Note the use of a nullary operator (function with no arguments) to describe constants that must be provided by the resource. The reason for using this form is to avoid a description of either the number or form of the set of values that represent a given operand type. In this manner we strive for representational independence. Intuitively, the function True is the function we invoke to return the representation of "True" in this resource. Moreover, the fact that there are only two constants of type Boolean is not something directly required by the above specification of Boolean. As we shall see, it is a consequence of the meaning associated to the form.

Now that we have introduced the form of an abstract description, how do we define its meaning? Put simply, meaning is always determined by associating form to real objects. We say that a real object is a realization of an abstract object defined by a specification if three conditions are satisfied.

**Condition 1.** For each operand type of the spec there is a corresponding set of values in the real object (the operands of that type), and to each operator in the spec, there is a corresponding operation in the real object that is defined on values that correspond to the operand types of the operator.

But this is not all. This correspondence between forms in the spec of the abstract object and operand sets and operations (functions) of the real object induces other correspondences. To illustrate what we mean consider the following example:

Suppose we have a "real object" that purports to provide the resources of what we know as Boolean. Suppose this "real object" consists of the set: 

| Switch = (on, off) |

and the operations:

| Nand: Switch, Switch -> Switch |

where Nand is defined by the usual table. Consider the correspondence between the formal and the actual:

| Boolean: -> Switch, True(): -> on, False(): -> off, Not(X): -> Nand(X, X), And(X, Y): -> Nand(Nand(X, Y), Nand(X, Y)) |

(where if $X_B$ is any Boolean term, $X_S$ is the Switch term corresponding to it.)

Note that we allow formal operators to be associated to any operation in the real object, as long as it has the correct number and types of operands. As a consequence, we now have a correspondence between any correctly formed "formal term" and operations in the real object. For example:

| And(Not(True()), False()) -> Nand(Nand(Nand(on, on), off), Nand(Nand(on, on), off)) |

Of course, the right side of this correspondence evaluates to "on". Moreover, using the properties given in the specification of Boolean, we can "prove" that the left side is equal to True(). This leads to another condition that a real object must satisfy, in order to be a "realization" of the abstract object.

**Condition 2.** In the correspondence between formal terms and actual terms, two formal terms are provably equal if and only if their corresponding actual terms have the same value.

One might think this is sufficient. It is not, however, because it does not prevent the occurrence of values or operations in the real object that do not correspond to anything described in the formal specification — for example, if we added another value to Switch and extended the definition of Nand to include some extraneous properties. Thus:

**Condition 3.** To every value and every operation of the real object there must correspond some formal term.

These are the complete conditions that characterize a "realization" of the abstract object defined by a specification. Clearly there are many realizations of the abstract object Boolean. Further, there are clearly other specifications that have the same exact class of realizations. Thus we make the following definition:

**Definition.** Two resource specifications are equivalent if they have the same realizations.

In this sense then the specification is not the abstract object either. A specification is like a word for an abstract concept; it is a pointer to all the objects that are realizations of the concept. And just as there are words in other languages that mean the same thing, there can be more than one specification that defines the same abstract object.

There are other important consequences of these definitions. As a minimum, we claim that we have a reasonably precise way of describing an object as a "thing in itself".

But can this method be used to describe useful abstract resources?

**The Design of an Abstract Processor**

In using the idea above to describe an abstract resource, we are forced to thinking of a real object strictly in terms of its functional properties. We must think in terms of the operand types that the resource provides and the basic operations that characterize the properties of the resource independent of any realization. On the one hand, it forces one into an unfamiliar form of thinking. On the other hand
it forces us to consider the very essence of things. The specification for Boolean discussed above illustrates this dilemma.

In the practical application of this method we realized the need to add a number of syntactic features to the specification language to allow an abbreviated syntax and to incorporate one specification within another. Similar problems in the theory of abstract data types have been carefully treated. Here we will focus on the practical problem of applying these ideas.

We began with a basic model of the resources of a processor. First there are data types and associated data operations. Next there is memory, used to store data values. Then there are instructions used to perform data operations on data, move data in memory, and control the execution of instructions. Instructions are also another data type. Since this was a design experiment, we eventually added features to the processor, such as stacks and registers, that were not absolutely necessary.

The construction of the specification followed the above sequence. First, primitive data types were described. The types included were boolean, signed and unsigned integer, ascii characters, and ascii character strings. The specification of boolean discussed above is an example of one we actually used. Since data types were first described as logically separate resources, we then had to describe them as a single aggregate. The resulting specification, called Values, effectively captures the concept of "typing". An operand of a primitive data type is called an "atom". There are operations in Values (for example Valofbool), used to find the "value" that represents an atom in a previously defined data type (such as Bool). There are other operations (such as Atomofbool) used to find the data type atoms represented by a value. These operations have two kinds of properties in Values, expressed as:

\[
\text{Atomofbool}(\text{Valofbool}(X)) = X
\]

and

\[
\text{Atomofbool}(\text{Valofchar}(X)) = \text{Undefined}
\]

e.tc.

"Undefined" is a special syntactic object used to express the notion that in any realization of the abstract resource, any formal term equal to, or provably equal to an "Undefined", need not have an interpretation in the realization. Thus the meaning of such a formal term is left open. In a strict sense, Undefined simply defines a predicate on formal terms that affects the semantics. Terms tagged by Undefined are not required to correspond to any actual term. It is not an "atom" of any type. It is a device used to allow the interpretation map to be a partial function on formal terms. (This is an example of one variation of our use of standard algebraic semantics).

The Values resource captures the idea of a uniform representation through "values" of all the different data types. With this we can describe memory.

Memory is described in two stages. First "memory addresses" are described. Memory is defined by a number of segments each identified by an arbitrary name constant. Given such a name we can find the starting address of its segment. For any address, we can find the next address and the previous address.

Second, in the resource specification called Amstate, the abstract description of a machine state is described through the use of the previous resources, a new operand type called "state", and the operations:

- \text{Initam} \rightarrow \text{state}
- \text{Store: val, memaddr} \rightarrow \text{state}
- \text{Fetch: memaddr} \rightarrow \text{val}

One of the required properties is obviously:

\[\text{Fetch}(M, \text{Store}(V, M, 0)) = V\]

Less obvious but required is:

\[\text{Store}(\text{Fetch}(M, Q), M, Q) = Q\]

and

\[\text{Fetch}(M, \text{Initam}()) = \text{Undefined}\]

Note the precision of these definitions.

Instructions are described prior to and as part of Amstate, since they are also represented as "values", but their properties are described last, in a final specification, called simply Am, that incorporates all the previously defined abstract resources. In the original design this final spec also includes other resources not discussed here, such as stacks, and registers. This final specification requires no new operand types, but introduces two new important operations.

\[\text{Prog: memaddr, state} \rightarrow \text{state}.\]

\[\text{Xeq: instr, memaddr, state} \rightarrow \text{state}.\]

The most significant properties of these operations are illustrated by examining two sample properties. First there is the property:

\[\text{prog}(M, Q) = \text{Xeq}(\text{Atomofinstr}(\text{Fetch}(M, Q), M, Q))\]

Intuitively it says, "to find the state obtained from Prog, given that you are executing the instruction at \(M\) in the current state \(Q\), fetch the value at \(M\) in state \(Q\), find the instruction this value represents, and execute it in state \(Q\)." If, for example, the instruction was found to be the atom:

\[\text{Mov}_{m-m}: \text{memaddr, memaddr} \rightarrow \text{instr},\]

then applying another property found also in the Am spec:

\[\text{Xeq}(\text{Mov}_{m-m}(M_1, M_2), M, 0) = \]

\[\text{Prog}(\text{Nextmemaddr}(M), \text{Store}(\text{Fetch}(M, Q), M_2)),\]

which says "fetch the value from \(M_1\) in state \(Q\), and store it into \(M_2\) in state \(Q\), and then in the resulting state, evaluate the program beginning at the next address".

Obviously, we have left out much detail. The full specification covers about twenty pages. Yet, these examples illustrate the remarkable precision of the method, particularly if we are reminded of the "meaning" of the specifications as discussed in the first section of this paper. And although the form of a specification, and particularly the appropriate properties of its operators, are often not apparent they follow from the kind of reflection required to write the spec for Boolean.
Implementation of the Design.

The next issue is whether it is possible to actually implement the abstract resource from its specification. In theory, we have exact conditions that characterize a correct realization. Some researchers have used specifications as a language for rapid prototyping. Our objective, however, was more practical: to use the specification as a guide to building a realization. One immediate problem is that some of the resources as defined, would require an infinite realization in order to satisfy the three conditions discussed earlier. An example is the memory address resource. In the specification, we did not define any ‘final’ memory address. Similarly for the integer realization. Some researchers have used specifications as defined formal terms have the same property, it is impossible to deduce anything about the internal representation of data or instructions or the alignment or wordsize of memory. or program.

In other words, the interpretation function is a partial function on defined terms as well. In our case, there will be formal terms of the form Nextmemaddr(M) (for some formal term M of type memaddr) that do not have an interpretation in the realization. Equivalently, a partial realization defines an additional predicate on formal terms, called "Unimplemented". Clearly, if a formal term is undefined, it is already unimplemented.

The implementation included two parts: a partial software emulation of the abstract processor resources as described in the specifications, and an assembler for an assembly language defined for the processor. The assembler was never defined formally. It was developed as a tool to create programs for the abstract processor. Some of its properties, however, are of interest. First we will discuss the processor.

The processor emulation was written in C under Unix on a VAX 11/780. The C program was designed to emulate the specification as closely as possible. Operand types are almost uniformly defined using a typedef, and operations on these types are defined by C functions with parameters and return values of the appropriate type. For example:

typedef struct {
    short type;
    bool val;
} BOOL;

BOOL not(a) {
    a.val = !a.val;
    return(a);
}

A significant example is the function corresponding to the operation Prog described above:

STATE prog(m,q) {
    q = seq(atomofinstr(fetch(m,a),m,q));
}

The C program to emulate this operation is almost a duplicate of the operation's fundamental property. This is similar to equational programming.

The abstract processor, presently called the AM-1, went through some evolutions. Instructions with program relative addressing were added in the specification and implementation. A file data type and appropriate operations were added to create a more usable, and realistic resource.

The second part of the implementation, an assembly language and assembler, were designed and implemented with the help of the LEX and YACC tools on UNIX. The end result is a processor and assembly language somewhat on the level of the Motorola 68000 microprocessor. All data type operations are performed by instructions whose operands are in registers. In addition to these instructions there are fourteen instructions for moving data and controlling execution:

- MOVE (data with seven addressing modes)
- PUSH (a value on a stack)
- POP (a value off a stack to a destination)
- POPX (clear a value off a stack)
- JMP (jump absolute)
- BRA (branch relative)
- IF (conditional jump)
- IITE (conditional binary switch)
- JSR (jump absolute subroutine)
- BSR (branch relative subroutine)
- RTS (return subroutine)
- LINK (link a frame and allocate)
- UNLINK (unlink and free)
- STOP (halt execution)

In addition there are ten addressing modes, and a number of storage and preprocessor directives for the assembler.

Although the assembly language is not part of the formal specification, it has a number of interesting properties. Since it is used to program a processor defined independently of any realization, and since its data types have the same property, it is impossible to deduce anything about the internal representation of data or instructions, or the alignment or wordsize of memory, from an AM-1 program. In AM-1, memory is a logical resource used to hold logical values representing logical data types. It takes one memory cell to hold a value whether or not that value is a boolean, an integer, a string or an instruction. Thus programs for the AM-1 are inherently highly portable. Also, each data type is represented in pure form. For example, stacks are defined as abstract stacks, and the stack operations operate on stacks only, not on random access memory. In conventional machines, stacks are overloaded onto random access memory because that is how they are implemented. Logically, they are a different resource.
This leads to another property of AM-1 programs: they unambiguously express the intent of the programmer. If the program uses stacks, the programmer must mean he intends to use a stack. If a programmer uses a value in a memory cell as an integer, he must represent a logical integer atom and the same value cannot be referenced as a boolean, say. In fact, the AM-1 enforces this strong typing throughout. For example, it is impossible to execute data as an instruction. In conventional assembly languages, the logical intent of instructions is mixed with questions of implementation and the internal representation of instructions and data, and often the programmer's intentions are obscured. If we attempt to move a program to another machine, we cannot distinguish between what the program intends to do and artifacts of the implementation. This problem even exists in some "virtual" machine languages designed to achieve portability. In our case, the inherently abstract quality of the specification method precludes this difficulty.

Conclusion

We have described the fundamental elements of our approach to the problem of describing, designing, and implementing a practical abstract resource. There are limitations. We are as yet uncertain about how to treat the "modality" of a resource. We have taken the same approach to "time" and order of execution as taken in functional programming semantics. Operators must be evaluated before the application of an operation. There are approaches that specify sequencing. Also, we have no practical means for establishing the correctness of a realization or validating a resource specification. We doubt the existence of general methods but expect there are practical methods based on the theory we have described that can assist.

Although we have not given a complete description of the substance of our methods, we hope that we have managed to communicate its spirit. This spirit is consonant with the desire to bring an element of rigor and precision to an important applied problem. It is consistent with some of the tenets expressed in Wegner[13] and Turski[14] for a sounder approach to the problems of portability and reusability of computing resources.

References