A system to implement primitive data types

Alan Mitchell Durham
Instituto de Matemática e Estatística da USP
alan@ime.usp.br

Ralph Johnson
University of Illinois
{johnson@cs.uiuc.edu}

Abstract
When compiling a high level language like Java, Lisp or Smalltalk, the implementation of primitive data types is a large part of the task. The powerful data types make most of the virtual machine, and code generation for them must be rewritten if any of the specifications of the data types change. Moreover, most of the differences between different implementations of the same data types are in the format of their representations. By describing format and semantics separately we can create more modular and reusable specifications, and therefore more modular and reusable interfaces with the back-end of the compiler. This paper presents a model for specifying data type implementations in a compiler, and a prototype system that was built following that model. This system automatically generates a compiler component from the specifications of a data type, assuring consistency between implementation and specification. The system also provides a visual interface to inspect and specify data type formats.

Keywords: Primitive Data Types, Compilers, Virtual Machines

1 Introduction
The common and straightforward way to implement a primitive data type is to provide a library of routines, or to just create code generation statements inside the compiler. Unfortunately, this unstructured approach can make the implementation convoluted and with
a lot of redundant code replicated across the routines. One solution is providing implementation guidelines that will produce more modular code. However, the best solution is to provide a model for implementing data types that generates modular implementations. With this model we can develop a formalism to describe data types. This formalism can be used to automatically generate data type implementations. A good formalism will make the specifications easier to understand and to change. Finally, automatically generating code from the specifications prevents the implementor from having to keep the specification and implementation consistent.

This paper presents a model for data type implementation and a formalism for data type specification based on this model. These are part of a prototype system that uses this framework to generate implementations of primitive data types.

2 Partitioning the Implementation

What is the real problem of implementing primitive data types? If we look at Shebbs' survey on primitive data type implementations [9] we can see that, in general, the semantics of each primitive operation is the same across different implementations. What varies between different implementations of data types is their run-time format. One good example is Java, where the Virtual Machine specification is very particular about the semantics of the operations but puts no constraint on their representation, not even mentioning data type run-time format [5]. Even across different virtual machines, there are data types that have invariant semantics, such as "SmallIntegers", i.e. machine-operable integers, which exist in the virtual machines of Lisp, Smalltalk, Java, Scheme, etc.

Since most of the variance in primitive data type implementations is in the format of the data, not on the details of the operations, separating the specification of the format from the specification of the operations on the data makes both more reusable.

Finally, some common operations of virtual machines depend strictly on data type format and can have their code automatically generated from declarative format descriptions. These include routines for accessing and updating information, routines for run-time type checking, routines for copying, and other routines for supporting garbage collection, including routines for marking objects and for performing pointer-searching. If we develop a good specification language for data type format, all these operations can be automatically generated.

3 The Description of a Data Type Implementation

Following the reasoning presented in the last section, we separate the description of a data type implementation in two tasks: describing the format and describing the semantics. All details about the data type format are abstracted out of the semantics descriptions and vice-versa.

3.1 Specifying the Semantics of Basic Operations and Constants

The basic operations on primitive data types can be arbitrarily complex in virtual machines of modern languages. Languages like Snobol4, APL, Smalltalk and Java can have primitive operations that are equivalent to whole programs of other languages.
Since our main concern is to produce a tool that produces compiler components to generate efficient implementations, it is important to provide a fine level of control over the code being produced. Implementing the basic operations efficiently is absolutely crucial for the performance of the virtual machine as a whole. Taking this into consideration, our prototype system uses an abstract assembly code, named RTL [6], as the specification language. RTL is a language that offers an infinite number of 32-bit registers on top of which all the basic hardware operations apply (e.g. arithmetic, logic and bit manipulation). The low level of the language, associated with a register allocator and a code optimizer permits the specification of efficient code\(^1\).

We parameterize the description of the primitive operations by abstracting out format information, which give us a more modular approach that facilitates code reuse. This is done in the semantic specifications by naming the various parts of the data type representation and using these names in the RTL code, instead of writing down the access sequences. We will see how this is done in the next section.

### 3.2 Describing the Format of a Data Type

We should keep in mind that the format of data types can be arbitrarily complex. To illustrate this we will look at three examples. But, first, it is important to make clear our use of the words "reference" and "pointer":

- a *pointer* is a set of bits containing some memory address.
- a *reference* is a register-sized group of bits that is used by the virtual machine of a language to refer to an instance of a data type. In other words, a "reference" is a run-time "handle" to an instance of a data type. References can contain pointers but are not the same thing.

Our first example is an implementation of a *small integer*, that is, an integer that can be operated directly by hardware functionality. Figure 1 illustrates a possible representation of a small integer, where bottom 31 bits are used to store the number (indicated by 31 "*"), and the top bit contains a tag '1'.

![Figure 1: Format of Small Integer](image1.png)

Our next example is an implementation of a LISP *cons* cell. A *cons* cell is a data type containing two references to objects of arbitrary data types. We call these the *car* and the *cdr* of the LISP cell. Figure 2 shows a typical representation of a LISP *cons* cell, with the reference containing a pointer to a data block that contains the *car* and the *cdr* fields. Both *car* and *cdr* are 4 bytes long. The reference has two tag bits, with contents '10'. All other bits in the reference can have arbitrary content, indicated by "*****...". One interesting characteristic of this tag is that it can be considered directly a part of the pointer, meaning that all objects are aligned in even numbered memory positions.
Our last example (fig. 3) presents a much more complex data type, a *Big Indexed Object* in Tektronix's Smalltalk. A Big Indexed Object is a data type roughly equivalent to a record that contains an array as one of its fields. The representation of the data type stores

- the elements of this "record",
- the elements of the indexed part,
- a pointer to the object’s "class" (a description of the user-defined data type that uses this primitive data type as a representation),
- some other miscellaneous information.
In Smalltalk-80 all instances of data types are accessed through a 32-bit reference, with its top bit set to the value "zero". Like in our LISP example, this tag is actually part of the address, so no decoding is needed for de-referencing. The address contained in the reference is the location of a header. The first word of this header contains the total size of the object (in the first 16 bits) and the number of fixed fields \( M \) (in bits 16 to 23). The second word of the header has a tag with value 1 at bit 23, distinguishing this as a "big object". The third word of the header is a reference to another instance of a primitive data type that describes the "class" of the object. Words 4 thru \( M+4 \) contain references to other objects, or the "regular" entries in our "record". Word number \( M+5 \) is a pointer to a remote header. This new header has only two 32-bit pointers, the first pointing back to the header and the next pointing to another data block with an indexed sequence of references to other objects (i.e. the "array" part of our "record").

As we can see in the example figures, formats of data types can be very complex, involving internal pointers, tags that are part of the useful information, etc. The code sequences to access all these parts are not trivial and may involve a series of indirections and possibly even the resolution of relative pointers (as in "cdr-coding") or double indirection, as is the case of object tables. However, we can describe this complexity out of only three entities:

- **the reference**, which is the "handle" used by the language to refer to a specific instance of the data type,
- **data blocks**, which are contiguous parts of memory that hold information. In our Lisp example we have just one data block, the one that holds both the car and the cdr. In the Smalltalk example we have three data blocks, the one that holds the "primary header" and the fixed fields, the one that holds the "secondary header", and the one that holds the indexed data.
- **fields**, which are the atomic units that we want to access inside the data type representation. In the Lisp case we have 3 fields, the one with the pointer to the data block, the car, and the cdr. In the Smalltalk examples we have many fields: the pointer to the primary header located in the reference, all the fields of the primary header (tags, class reference, size field, etc.) the two fields of the secondary header (two pointers), and an indexed field in the last data block.

In our framework, we describe the format of a data type describing:

- a field’s name and relative position in the data block or in the reference
- how to reach a data block given a reference
- which fields are contained in each data block and in the reference
- a field’s contents (if it is an absolute pointer, a relative pointer, a reference, or regular data)
- which fields are indexed fields (we call them aggregate fields)

Now, as each field receives a name, the code for specifying primitive operations can use the field names as identifiers instead of including the instructions to access or updated the fields. This is done using the format

\[
<\text{variable name}>\{<\text{data type name}>\}.<\text{field name}>
\]

As an example let's see the code necessary to implement the primitive operation `indexedAt`: 
in our Smalltalk Object. This operation receives as argument a small integer (remember the tag on bits 30 and 31) and returns the content of the respective field of the indexed part of the big object:

```plaintext
program indexedAt($0,$1){
    ; first get the index in R1
    ; to do this we have to extract the integer from the reference
    ; we suppose here integers are stored in the bottom 31 bits of the reference.
    ; Since the top bit is zero, no extraction is needed
    R1 := $1. ; get the reference in R1
    R1 := R1 & '7FFFFFFF'. ; remove the tag bit
    ; now we get the address of the first indexed field
    ; the reference to the big object is in '$0'
    R2 := *(R0) ; R2 stores the first word of the header
    R2 := R2 & '00FF0000'. ; R2 has m
    R2 := R2 $>> 14. ; shift right 14 bits to get integer
    ; (M*4 --we shifted only 14 bits)
    R3 := $0. ; R3 is the address of the header block
    R3 := R3+12. ; R3 is the address of the first fixed field
    R3 := R3+R2. ; R3 is address of ptr. to remote header
    R4 := *(R3). ; R4 has address of remote header
    R4 := R4 + 4
    R4 := *(R4). ; R4 points to first indexed field
    ; finally we compute the address of the desired field and return it
    R1 := R1 * 4.
    R4 := R4 + R1
    R5 := *(R4^ R5. returned value }
```

Now, if we can substitute the access sequences of fields and arguments for names we will have a version much easier to read and understand:

```plaintext
program indexedAt{
    ; first get the contents of the argument
    ; (indicated by '$1')
    ; to get its contents we access the field 'data'
    R1 := $1{SmallInteger}.data
    R1 := $0.IndexedField[R1].
    ^ R2. ; value to be returned
}
```

We classify fields by the type of information they can contain. Classifying fields this way makes it possible to automatically generate not only access and update code, but also memory management support code. Based on the information needed for automatically generating code for accessing information, moving objects, and supporting memory management, we have created three main types of field:

- **pointer field** - A field that holds a pointer to another data block of the data type representation. The system needs to know which are the pointer fields in order to automatically generate code for moving data objects and for returning the object's memory space to the free area when the object is garbage collected. Since a pointer can be stored in coded form (e.g. displacement from a known address), there are many subtypes of pointer fields, one for each type of coding.

- **reference field** - A field that stores a reference to an instance of a primitive data type. Being able to pinpoint internal references to instances of data types is crucial for finding which objects are live at garbage collection time.

- **data field** - A field that holds any arbitrary information that is not a pointer or a
reference. Data fields are used by the internal semantics of the data type's operations and do not concern memory management. The system only needs to generate code for accessing and updating these fields.

In addition to the classification above, any type of field can be an repetition aggregate field, a field that actually designates many positions in memory, each one accessed by an index. Examples of this are the fixed fields and the it indexed fields of figure 3. Each field that is an aggregate can have its multiplicity set at design time or at compile-time/run-time. If the multiplicity is set at design-time then the field is a fixed repetition aggregate field, otherwise it is a variant repetition aggregate field.

3.2.1 Formalizing Data Type Format Description

We can express the above description with the following grammar:

/*we separate format description from semantic descriptions*/
(1)PDTDescription à <name>
  ReferenceDescription
  OperationDescription
(2)OperationDescription à <name> <RTL code>
  /*we describe a reference by describing its fields*/
(3)ReferenceDescription à {FieldDescription}+
  /*fields can be single or can be multiple (repetition aggregate)*/
(4)FieldDescription à SingleFieldDescription || RepetitionAggregateDescription

/*describe the format of a field describing their position specifying any tags that they contain, and describing their type*/
(5)SingleFieldDescription à Position
  Tag
  FieldTypeDescription
/*we specify position by giving the starting word and bit in the data block*/
(6)Position à <word index>
  <first bit>
  <number of bits>
/*A tag describes the contents of each big of a field, which can be any (A), one (1), or zero (0). Therefore, a low tag of '10' in a 6 bit field can be described as AAAA10.*/
(7)Tag à {A||0||1}+
(8)FieldTypeDescription à DataFieldDescription || ReferenceFieldDescription || PointerFieldDescription
/*data fields do not need more information*/
(9)DataFieldDescription à <empty>
/*ref. fields do not need more information*/
(10)ReferenceFieldDescription à <empty>
/*Pointer fields describe the data block they point to Also, they may be encoded pointers, that is, either pointers into a table of addresses or relative pointers.*/
(11)PointerFieldDescription à DataBlockDescription
  [TableBaseInformation|| RelativeToFieldInformation]
/* to access a pointer table we must know the name of the global run-time variable that holds the 
table's base address */
(12) TableBaseInformation à <global variable name>

/* A relative pointer needs to specify the name of the field to which its pointer increments on. */
(13) RelativeToFieldInformation à <field name>

/* A data block has a description for each of its fields. Some blocks that can appear many times 
in a representation (e.g. representation of large integers in LISP) and need to indicate the 
field that points to its mirror image (the "recursive pointer"). */
(14) DataBlockDescription à {FieldDescription}+ [<recursive pointer>]

/* Repetition aggregates are used to describe indexed fields. An aggregate description must 
describe the field that should be repeated and the number of repetitions, which can be set at 
design time, or be stored in some other field */
(15) RepetitionAggregateDescription à

    FieldDescription
    {<repetition number>||
    <size field>}

There are some constraints not mentioned in the above grammar. In particular:

- the word index of a field inside the reference (rules 5 and 6) is always 0.
- a tag description has to specify all the bits of the field.
- the field name that describes the base for a relative pointer has to indicate a field 
  already defined in the data type.
- the '<recursive pointer>' entry in the data block description has to describe a pointer 
  field in that same data block. That pointer field has to refer to the mirror image of the 
  same data block.
- the '<size field>' entry in a repetition aggregate description has to refer to a field of the 
  same data type.
- no two fields can have the same name in a data type.

### 3.2.2 Specifying the Examples

These field types and the notation of reference and data block are enough to specify 
arbitrarily complex formats. In our example of the "big" objects, the classification of fields 
would be:

- **pointer fields**: field in reference, ptr. to remote field, ptr. to main header, ptr. to 
  indexable fields block.
- **reference field**: Class (points to another data type instance).
- **data fields**: Total Size, M.
- **aggregate fields**: FixedField, IndexedField.

These fields are put together using:

- three data blocks:, one for the indexed fields, one for the remote header, and one for 
  the main header.
- the reference: holds only one field, a pointer to the main header.

As another example, [figure 4](http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-...) shows how our cons cell implementation is specified using the 
various descriptors.
4 The Primitive Data Types Browser, a Primitive Data Type Specification Tool

Any virtual machine data format can be described using only these descriptor types, it is not necessary to make more sub-types. This makes it easy to build a language to describe data type implementations using only menus and templates.

However, all the above formalism would be of little help without a user-friendly specification interface. This interface will be provided by a specification tool called Primitive Data Types Browser, which uses a frame-based language to specify the format of data types. With this browser, compiler writers can build new sets of primitive data types for a compiler by just inspecting the existing ones and collecting a set of them. Through editing in the browser implementors can also create new customized specifications, if needed. A visualization of the primitive data type format, deduced from its specification, can be used as an aid for understanding individual implementations of a data type. The browser comes with cut and paste facilities that makes it easy to copy data type formats and operations from other data types, whether they are of the same virtual machine or not.

After the language implementor finishes the specification the tool generates a compiler component that can be used to generate code that implements the data types. Figure 5

Figure 4: Specification of cons cell design
shows an example of such browser.

![PDTBrowser](image)

**Figure 5: PDTBrowser**

We have built a frame-based specification language based on the grammar described above. This frame-based language has a different frame for each field type. The virtual machine architect uses browser to create new data types. Once a data type is created, the format is described by specifying each of the fields. To add a field, the architect only has to choose a field type and fill a template. There is a template for each field type. Data block descriptors are created automatically each time a pointer field is specified. There is only one reference descriptor per data type, which is automatically created with the data type. **Figure 6** shows two of the possible templates.
5 How the System Works

The prototype system we developed is designed not only for specifying of the data types but also for specifying the rest of the virtual machine of a language. Figure 6 illustrates the use of the system.

The language implementor (the "user") uses systems's tools to specify the implementation of the data types (step 1). He describes the semantics of the primitive operations on the data types using a register transfer language (RTL), and specifies the data type formats using a
template language. The templates embody the format specification language. From the specifications, the system generates a specification component (called "PDTDescriptor") for each of the data types (step 2). To build a compiler, one just links the compiler front-end to the back end and to these components. Then, at compile time, the front-end delegates to the new components all the code generation for primitive data types. The front end just specifies the operation to be generated, the location at run-time of a reference to that data type, and the arguments (step 3), the rest is done by the automatically-generated components, which communicate with the code generator (step 4) which then generates object code (step 5).

Details about the component implementation and the interfaces can be found in [3].

6 Related Work

The more frequently explored aspect virtual machine implementation in compilers is the implementation of primitive data types. However, most of the work has had as a goal constructing an inference system for designing the implementation of the primitive data types. Jalote [4] proposed a system based on standard list-based implementations of data types geared to allow rapid prototyping and testing of new data type specifications. Jalote's system is directed towards rapid prototyping when testing a specification for an ADT (abstract data type) and always generates a list-based implementation. The system as a rule does not generate efficient implementations.

The work done on the SETL compiler [8] used an inference system to produce the best implementation of sets based on an analysis of the source program, but it is restricted to the search of efficient implementations for sets, and is geared toward a choice among some pre-specified patterns.

Shebbs' work [9] is more ambitious: he enumerates design heuristics for implementing data types which include rules for developing tagging schemes, vector representations, etc. These heuristics are incorporated into a designer system that receives as an input the description of the primitive data types in the source language and the description of the target machine. Even though promising, Shebbs work is restricted by the magnitude of the search spaces involved. There are still too many uncertainties in the design of primitive data type implementations. Many of the existing heuristics have conflicting goals, and their number is too big. His system was only successful in generating implementations for the more simple data types and does not provide for storage management. Until we have a deeper understanding of the rules that govern primitive data type implementation, there seems to be little hope that we can get a totally automatic design system.

Finally, the idea of parameterizing primitive operation descriptions by the field accesses is not new. Miller's, in his code generation system [7] suggested this same idea. There are important differences, though. Miller was concerned with the low level details of code generation. Also, even though Miller recognized the usefulness of a declarative specification of data type formats, he did not propose one: all the format specifications in his system were, as he wrote, 'sequential' and not declarative. Moreover, he was concerned with the low level details of code generation. In our system, we abstract out all details about register allocation and other code-generation particularities. The declarative style of PDATIS's format
specifications permits generating many operations automatically, including operations for the support for garbage collection and run-time type checking, as well as generating a visual representation of each format.

7 Conclusion

We have presented a framework to implement simple and complex data types. This framework creates more modular implementations by separating format specification from semantic specification. We have also shown how we used this framework to produce a visual specification tool that can be used to produce compiler components to implement the code generation of the data types for a compiler. The component is automatically generated so the compiler implementation will always be consistent with the data type specification. Since the implementation of a given data type is generally the same but for its run-time format, writing a new implementation many times will involve only using menus and templates. In a future paper we will present a generic interface for the support of memory management systems that can be automatically generated from the data type specification presented here.

More details about the development of this system be found in [1] and [2] and also in [3].

References


1 In fact, any other language of the same level will be satisfactory, but one interesting feature of RTL is that it offers a higher-level flow of control constructs like 'if' and 'while'. This provides nice control abstractions that makes the code easier to read, without hiding the details of memory use and low-level machine operations.

2 Of course there are other possibilities like a low tag of zeroes (in which case the name is directly operable for addition and subtraction), etc.

3 This is used to enable automatic generation of memory management routines.

4 We designate by $0$ the first argument of the routine (which is the data type associated with the operation) and by $1$ the second argument to that routine.

5 Be careful with the difference between reference field and reference. In a data type description the reference is the description of the "handle" to an instance of that data type, and a reference field is a field that holds one such handle, but which can be a reference to another instance of a data type.

6 In our Smalltalk example both aggregate fields are variant repetition.

7 In the prototype system there is also a specification tool to describe the environment.

8 The front-end may specify the data type, but the system can automatically generate run-time type tests.

9 "[...] each of these has been solved by a shortcut that should be addressed in the future: [...] The designer encounters combinatorial explosions frequently so it has restrictive rules and can only be used on simple types [...] nothing is done about garbage collection or storage recovery in general [...]"[9](page120).}