DataScript- A Specification and Scripting Language for Binary Data

Conference Paper *in* Lecture Notes in Computer Science · October 2002

DOI: 10.1007/3-540-45821-2_4 · Source: DBLP

Godmar Back
Virginia Tech
53 PUBLICATIONS **1,813** CITATIONS

DataScript–a Specification and Scripting Language for Binary Data

Godmar Back

Stanford University
gback@stanford.edu

Abstract. DataScript is a language to describe and manipulate binary data formats as types. DataScript consists of two components: a constraint-based specification language that uses DataScript types to describe the physical layout of data and a language binding that provides a simple programming interface to script binary data. A DataScript compiler generates Java libraries that are linked with DataScript scripts.
DataScript specifications can be used to describe formats in a programmatic way, eliminating the vagaries and ambiguities often associated with prosaic format descriptions. The libraries generated by the DataScript compiler free the programmer from the tedious task of coding input and output routines. More importantly, they assure correctness and safety by validating both the input read and the output generated. We show examples that demonstrate that DataScript is simple, yet powerful enough to describe many commonly used formats. Similar to how scripting languages such as Perl allow the manipulation of text files, the libraries generated by the DataScript compiler can be used to quickly write scripts that safely manipulate binary files.

1 Introduction

Currently, binary data formats are typically described in English prose in design documents and standards. This method is inefficient, prone to ambiguities and misunderstandings, and does not lend itself to easy reuse and sharing of these metadata specifications between applications. As a result, some applications may produce incompatible output that cannot be read by other applications, or worse, causes other applications to crash.

Casual scripting of binary data is also cumbersome. While there are many tools to validate and manipulate high-level textual data formats such as XML, there are no tools for binary formats. Currently, a programmer has to choose a language, convert the prosaic description into language-specific data structures such as C structures; then he must implement input routines to read the data into memory, and output routines to write the data back to external storage. The programmer has to account for alignment and byte-order issues, which are notorious for their potential for bugs when such programs are ported to different architectures.

Even where libraries or tools to read a given file format are available, a programmer has to learn those libraries' APIs first, and there may be limitations in what they can do. The necessary feature to perform the task at hand may not be provided, because the developers of the tool or library did not foresee a need for it. Where libraries are not available, programmers tend to "whip up" input and output routines that often only provide partial implementations, do not validate their input, or produce nonconforming output that causes problems when it is processed in the future.

Many binary formats include constraints that guarantee the consistency of the data, such as parameter ranges, length constraints, checksums and others. Like the layout of the data itself, these constraints are also typically expressed in prose rather than in a form suitable for automatic translation into programs.

DataScript's specification language solves these problems by describing binary data formats as types. The specification language has the following properties:

– **simple and intuitive.** DataScript is designed to be a simple solution to what is conceptually a simple problem. DataScript types are easily readable; we kept the number of abstractions needed to understand a DataScript specification to a minimum.
– **architecture-independent.** Unlike C header files, DataScript types are described in a architecture-independent way. The developer is responsible for specifying the size and byte order of any and all data elements—since DataScript is a language to describe the physical layout of data, no attempt is made to hide such details from the programmer.
– **constraint-based.** Each component of a DataScript type can have constraints associated with it. These constraints must be fulfilled if a given piece of binary data corresponds to a type. They can be used to discriminate between types, or they can be used to express invariants that hold if the data is consistent.
– **declarative.** DataScript types are purely declarative, i.e., the specification of a type does not include any side effects. Purely declarative descriptions allow the same format description to be used for input (parsing binary data) and output (generating binary data according to a given format).
– **practical.** DataScript provides direct support for commonly used idioms. For instance, some file formats store length and offset parameters in headers. Such dependencies can be directly expressed in DataScript.

2 The DataScript Language

A typical situation in which binary data are read and processed is the loading of Java class files by a class loader in a Java Virtual Machine (JVM) [4]. Java is often used for mobile code, which a virtual machine may load from untrusted sources, hence the class loader must verify the well-formedness of the class file. It must be prepared to handle corrupted or malformed class files gracefully.

The loader reads a class file's bytes into memory, parses them and creates and populates data structures that correspond to the data stored in the file. This process involves identifying constants that are stored as multibyte integers in the file, reading parameters such as lengths and offsets, reading arrays of simple and variable-record types, reading flags and indices and checking that they are valid and within range, and the verification of constraints that ensure the internal consistency of a class file.

The set of rules that govern the internal structure and consistency of a class file are described in the DataScript specification shown in Fig. 1. A class file starts with magic integers that describe its version, which are followed by the constant pool, which is a linear array of variable-sized structures of `union` type "ConstantPoolInfo". A variable record or union type can take on different alternatives, which must be discriminated. In this case, the choice taken depends on the value of a "tag" field contained in each of the options, which include class entries, utf8 (string) entries, and others not shown in the figure.

Fields in a composite DataScript type can have constraints associated with them. For instance, the "magic" constant at the beginning of `ClassFile` must always be equal to 0xCAFEBABE. However, DataScript allows for the expression of more complex constraints: for instance, the Java virtual machine specification requires that the "super_class" field in a class file is either zero or the index of a constant pool entry with a tag value of "CONSTANT_Class". In general, a field's constraints can be an arbitrary boolean predicate. Predicates that are used in multiple places, such as "clazz", can be defined using the `constraint` keyword. "clazz" uses the `is` operator, which checks whether the union representing the specified constant pool entry represents an external reference to another class.

The DataScript compiler takes this specification and generates a set of Java classes and interfaces that can read and write Java class files. On input, the generated code checks that all constraints are satisfied. When reading union types, it uses constraints to discriminate between the different choices of a union, similar to how a top-down parser uses look-ahead tokens to decide on the next nonterminal during parsing. On output, it performs the same set of checks, which ensures the well-formedness of the generated output.

2.1 DataScript Types

In this section, we provide a more formal description of DataScript's types. DataScript types are defined recursively as follows. A DataScript type is either

– a primitive type, or
– a set type, which can be either a enumerated type or a bitmask type, or
– a linear array of a type, or
– a composite type, which can either be a record type or variant-record type.

We describe each of these types in turn.

```
const uint8    CONSTANT_Utf8  = 1;
const uint8    CONSTANT_Class = 7;      // ...

const uint16   ACC_PUBLIC    = 0x0001;
const uint16   ACC_ABSTRACT  = 0x0400;  // ...

ClassFile {
    uint32           magic = 0xCAFEBABE;
    uint16           minor_version = 3;
    uint16           major_version = 45;
    uint16           cp_count;
    ConstantPoolInfo constant_pool[1..cp_count];

    bitmask uint16 ClassFlags {
        ACC_PUBLIC, ACC_FINAL, ACC_ABSTRACT, ACC_INTERFACE, ACC_SUPER
    } access_flags;
    uint16        this_class  : clazz(this_class);
    uint16        super_class : super_class == 0 || clazz(super_class);
    uint16        interfaces_count;
    {
        uint16 ifidx : clazz(ifidx);
    } interfaces[interfaces_count];
    uint16        fields_count;
    FieldInfo     fields[fields_count];
    uint16        methods_count;
    MethodInfo    methods[methods_count];
    uint16        attributes_count;
    AttributeInfo attributes[attributes_count];

    constraint clazz(uint16 x) { constant_pool[x] is cp_class; }
};

union ConstantPoolInfo {
    {
        uint8 tag = CONSTANT_Class;
        uint16 name_index;
    } cp_class;

    Utf8 {
        uint8 tag = CONSTANT_Utf8;
        uint16 length;
        uint8 bytes[length];
    } cp_utf8;
    // ... other choices ...
};
```

Fig. 1. DataScript description of Java class files. The complete description is 237 lines long.

Primitive Types Primitive types form the basic blocks upon which more complex types can be built. They include bit fields, 8-bit integers (bytes), 16-bit, 32-bit, 64-bit, and 128-bit integers. All primitive types are interpreted as integers, which can be either signed or unsigned. The size and signedness is encoded in the keyword, i.e., `uint32` is a 32-bit unsigned integer, while `int16` is a signed 16-bit integer.

The byte order for multibyte integers can be specified by an optional attribute prefix, which can be either `little` or `big`. By default, a primitive type inherits the byte order attribute of its enclosing composite type. If the enclosing composite type has no byte order attribute, big endian or network byte order is assumed (as is the case for Java class files).

Set Types DataScript supports two types of set types: enumerated types (`enum`) and bitmask types (`bitmask`). A set type forms a subset of an underlying primitive type, which specifies the byte order and signedness used to store and interpret its values. This example was taken from the DataScript specification of IEEE 802.11b [2] packets:

```
enum bit:4 ControlPacketType {
   CTRL_PS_POLL     = 1010b,     // Power Save (PS) Poll
   CTRL_RTS         = 1011b, ... // Request To Send
};
```

Composite Types DataScript uses a C-like style to describe composite types. DataScript's composite types are similar to C structs and unions, and the intuitions match. However, unlike in C, the `struct` keyword can be omitted, because it describes the default composition mode. Unlike C structs and unions, however, there is no implementation-dependent alignment, packing, or padding. As such, the size of a union may vary depending on which choice is taken.

DataScript unions are always discriminated. The first choice in a union whose constraints match the input is taken, hence the textual ordering in the specification is important. DataScript does not require that the constraints associated with the different choices of a union be disjoint, which allows for a default or fall-back case that is often used in specifications to allow room for future extensions.

Composite types can be lexically nested, which provides each type with its own lexical scope. Scoping provides a namespace for each type, it does not provide encapsulation. If a type is referenced from outside its confining type, a full qualification can be used, such as `ClassFile.ClassFlags`. Anonymous composite types can be used to define a fields whose types are never referenced, such as "cp_class" in Fig. 1.

Array Types DataScript arrays are linear arrays that use integer indices. Like record types, there is no alignment or padding between elements. An array's lower and upper bounds are given as two expressions: `[lower.. upper]`, which includes `lower`, but excludes `upper`. If `lower` is omitted, it defaults to zero.

The expressions used to specify the lower and upper bounds must be constructed such that their values can be determined at the time the array is read. Typically, an array's length will depend in some form on fields that were previously read. Hence, frequently used idioms such as a length field followed by an array of the specified length can be easily expressed.

2.2 DataScript Constraints

Each field in a composite type can have a constraint associated with it, which is specified as an arbitrary boolean predicate separated from the field name by a colon. Constant fields form a special case of constraints that can be written like an initializer in C and Java. Thus, the field definition `uint32 magic = 0xCAFEBABE` is a shorthand for `uint32 magic : magic == 0xCAFEBABE`.

Constraints are used in three cases: first, constraints that are attached to fields in a union type are used to discriminate that type. Second, constraints are used to express consistency requirements of a record type's fields. In a record type, a specification designer can choose the field to which a predicate is attached, with the restriction that a predicate can only refer to fields that lexically occur *before* the field to which the constraint is attached. This restriction was introduced to facilitate straightforward translation and to encourage readable DataScript specifications.

Third, constraints can be attached to array elements to limit the number of items in an array when its length is not known beforehand. Arrays with unspecified length grow until an element's constraints would be violated or until there is no more input.

Because DataScript constraints cannot have side effects, the DataScript compiler is free to generate code that evaluates constraints as soon as their constituents are known; the generated code can also evaluate a constraint predicate multiple times if needed.

DataScript predicates are boolean expressions. DataScript borrows its set of operators, associativity, and precedence rules from Java and C in the hope that doing so will facilitate adoption by tapping into programmers' existing skill sets. It includes arithmetic operations such as integer addition, multiplication, etc.; logical and arithmetic bit operations, relational and comparison operators, and so on.

DataScript requires additional operators. In many formats, the interpretation of a piece of data depends on descriptors that occurred earlier in the file. If the descriptor was modeled as a union type, the constraints used to discriminate the data will need to inquire which alternative of the descriptor was chosen. The `is` operator is a boolean operator that can be used to test whether an instance of a union type took on a specified choice.

Another common situation is for a field to express the length of an array not as the number of elements, but as its total size in bytes. The `sizeof` operator can be used to compute the number of elements. `sizeof` returns a field's or type's

```
ClassFile {
  constraint compare_utf8(uint16 idx, string str) {
    constant_pool[idx] is cp_utf8;
    constant_pool[idx].cp_utf8.bytes.compare_to_string(str);
  }         // rest omitted ...
}

union AttributeInfo {
  Code {
    uint16 name_idx: ClassFile.compare_utf8(name_idx, "Code");
    uint32 length;
    uint16 max_stack;
    uint16 max_locals;
    uint32 code_length;
    uint8  code[code_length];
    uint16 exception_table_length;
    {
      uint16  start_pc;
      uint16  end_pc;
      uint16  handler_pc;
      uint16  catch_t: catch_t == 0 || ClassFile.clazz(catch_t);
    } exception_table[exception_table_length];
    uint16 attributes_count;
    AttributeInfo attributes[attributes_count];
  } code : sizeof(code) == sizeof(code.name_idx)
                         + sizeof(code.length) + code.length;

  LineNumberTable {
    uint16 name_idx: ClassFile.compare_utf8(name_idx,"LineNumberTable");
    uint32 length;
    uint16 line_number_table_length;
    {
      uint16  start_pc: start_pc < Code_attribute.code_length;
      uint16 line_number;
    } line_number_table[line_number_table_length];
  } lnr_table: sizeof(lnr_table) == sizeof(lnr_table.length)
                    + sizeof(lnr_table.name_idx) + lnr_table.length;
};
```

Fig. 2. Recursive description of attributes in Java class files.

size in bytes, similar to the same-named operator in C. If `sizeof` is applied to a field, it will evaluate to the actual size of that field after it is read. If `sizeof` is applied to a composite type, the type must have a fixed size. A record type has a fixed size if all its elements have fixed sizes; a union type's size is fixed if all its elements have the same size, and an array type has a fixed size if all its elements have the same size and the number of elements is a constant. Figure 2 provides an example for the `sizeof` operator: the "length" field in a `Code` type must contain the length of the attribute, not including the number of bytes used for the "name_idx" and the "length" fields.

DataScript provides a `forall` operator that can be attached to arrays, and whose use is necessary when constraints cannot be attached to an array element because they refer to the array itself, whose definition occurs lexically after the element's definition. For instance, the following construction can be used to ensure that an array is sorted:

```
   uint32 a[l] : forall i in a : i < l-1 ? a[i] < a[i+1] : true;
```

The index check is necessary because any expression that causes out-of-bound array accesses will evaluate to false.

Constraints that are used in multiple places can be defined once and reused as part of predicates. A constraint defined in this way can only be applied to instances that are contained inside an instance of the type defining the constraint. For example, the constraint "compare_utf8" shown in Figure 2 can be used from inside `AttributeInfo` if the instance is contained in a `ClassFile` instance shown in Figure 1. Because both `ClassFile` and `AttributeInfo` are top-level definitions, this decision cannot be made statically, but must be made at run time. If at run time the instance is not contained, the expression containing the reference will evaluate to false. The DataScript compiler will reject type references if the specification provides no possible way for the referring type to be contained inside the referred type. This property can be checked by considering reachability in the graph that results from the "contained-in" relation between types.

2.3 Type Parameters

Each composite type can have a list of parameters associated with it. Parameters can be arbitrary DataScript types, either primitive or composite. Consider

```
Padding(uint32 size, uint32 align) {
  uint8 padding[(align - (size % align)) % align];
};

SomeDirectoryEntry {
  uint16 namesize;
  uint8  fname[namesize];
  Padding(sizeof fname, 4) pad;        // align to 4-byte boundary
  ...
};
```

Fig. 3. Use of type parameters for padding.

the example in Figure 3, which shows how type parameters can be used to implement a padding type that can be inserted to ensure that alignment constraints are not violated. `Padding` takes two parameters: the size of the unaligned structure that needs alignment, and the boundary at which it should be aligned. A variable-length byte array whose length is computed from these parameters serves to insert appropriate padding.

2.4 Labels

Many data formats are not sequential, but contain headers or directories whose entries store offsets into the file at which other structures start. To support such constructs, DataScript allows fields to be labeled. Unlike C labels, DataScript labels are integer expressions. When they are evaluated at run time, the resulting value is used as an offset at which the labeled field is located in the data.

Figure 4 shows part of the DataScript specification for an ELF [9] object file. ELF object files contain a number of sections with symbolic and linking information necessary to load a program or library into memory and to execute it. These sections are described in a section header table. However, the location of the section header table is not fixed; instead, the ELF specification states that the offset at which the section header table starts is given by the field "e_shoff", which is stored at a known offset in the ELF header at the beginning of the file. The DataScript specification expresses this by labeling the "sh_header" field with the expression "e_shoff".

Labels are offsets relative to the beginning of the record or union type in whose scope they occur. They can only depend on fields that occur lexically before them. On some occasions, it is necessary for a label to refer to a different structure. In these circumstances, a label base can be specified before the label

```
Elf32_File {
  Elf_Identification {
    uint32 magic = 0x7f454c46;
    ...
  } e_ident;

  ...

  uint32 e_shoff;
  uint32 e_flags;
  uint16 e_ehsize;
  uint16 e_phentsize;
  uint16 e_phnum;
  uint16 e_shentsize;
  uint16 e_shnum;
  uint16 e_shtrndx;
  ...
e_shoff:
  Elf32_SectionHeader {
    uint32 sh_name;
    uint32 sh_type;
    uint32 sh_flags;
    uint32 sh_addr;
    uint32 sh_offset;
    uint32 sh_size;
    uint32 sh_link;
    uint32 sh_info;
    uint32 sh_addralign;
    uint32 sh_entsize;
  } hdrs[e_shnum];

  ElfSection(hdrs[s$index]) s[e_shnum];
};
```

Fig. 4. Use of labels in ELF object file description.

```
ElfSection (Elf32_File.Elf32_SectionHeader h) {
Elf32_File:: h.sh_offset:
  union {
    { } null : h.sh_type == SHT_NULL;
    StringTable(h) strtab : h.sh_type == SHT_STRTAB;
    SymbolTable(h) symtab : h.sh_type == SHT_SYMTAB;
    SymbolTable(h) dynsym : h.sh_type == SHT_DYNSYM;
    RelocationTable(h) rel : h.sh_type == SHT_REL;
    ...
  } section;
};

SymbolTable(Elf32_File.Elf32_SectionHeader h) {
  Elf32_Sym {
    uint32    st_name;
    uint32    st_value;
    uint32    st_size;
    uint8     st_info;
    uint8     st_other;
    uint16    st_shndx;
  } entry[h.sh_size / sizeof Elf32_Sym];
};
```

Fig. 5. Use of a label base to express offsets that are relative to another structure.

that refers to the structure relative to which the label is to be interpreted. Figure 5 provides an example.

Figures 4 and 5 also demonstrate a more involved use of type parameters. Since each section is described by its header, the definition for the `ElfSection` type is parameterized by the `Elf32_SectionHeader` type. The type definition then uses fields such as "sh_offset" and "sh_type" to compute the offset and to discriminate the section type. The parameter "h" passed to the type can be passed on to other types. For instance, the `SymbolTable` type uses the "sh_size" field to compute the number of entries in the table if the section contains a symbol table.

3 Java Language Binding

We chose Java for our reference implementation because its object-oriented nature provides for a straighforward mapping of DataScript types to Java classes, because it provides architecture-independent primitive types, and because it has rich runtime support for reading from and writing to data streams. The DataScript compiler generates a Java class for each composite type, which includes accessor methods for each field. Each DataScript type's class provides a constructor that can construct an instance from a seekable input stream.

3.1 Prototype Implementation

Our current prototype code generator does not perform any optimizations. When parsing input, it simply sets a mark in the input stream and reads the necessary numbers of bytes for the next field, performs byte-order swapping if necessary, and initializes the field. It then checks the constraints associated with that field. If the constraint is violated, an exception is thrown. A catch handler resets the stream to the earlier set mark and aborts parsing the structure. For union types, a try/catch block is created for each choice. If a constraint for one choice is violated, the next choice is tried after the stream is reset until either a matching choice is found or there are no choices left, in which case the constructor is aborted.

3.2 Using Visitors

The DataScript code generator also creates the necessary interfaces to implement the visitor design pattern [1]. This generated visitor interface contains methods for each composite and primitive type in the specification. A default visitor implementation is generated that provides a depth-first traversal of a type. For a record type, it visits all fields in sequence. For a union type, it visits only the incarnated choice.

4 Related Work

PacketTypes. McCann and Chandra present a packet specification language called PacketTypes that serves as a type system for network packet formats [6]. It provides features that are similar to DataScript's: composite types, union types, and constraints. In addition, it provides a way for a type to overlay fields in other types, as is needed for protocol composition. PacketTypes's focus is on performing one operation well: matching a network packet received from the network to a specified protocol. For this reason, it generates stubs in C. Unlike DataScript, PacketTypes does not address generating data, it also does not provide an visitor interface for writing scripts the way DataScript's Java binding does. PacketTypes also does not support different byte orders, labels for non-sequential types such as ELF, nor type parameters.

OMG's Meta Object Facility. The OMG's Meta Object Facility or MOF [8] is a framework for standardizing the exchange of metadata. It defines a framework with which to describe metadata; a Java language binding [3] in the form of interfaces has been proposed recently. However, it does not address the physical layout of the actual data nor does it provide a way to automatically generate code to read and write data.

XML. XML is an extensible markup language used to store structured information as marked-up text while providing support for syntactic and limited semantic checking. In a world in which everybody used XML to exchange data, there would be no need for DataScript, yet this scenario is unlikely not only because of the large number of existing non-XML formats, but because XML has shortcomings, such as lack of space efficiency.

Interface Definition Languages (IDL). IDL languages such as RPC-XDR [5] or CORBA-IDL [7] provide a way to exchange data between applications. They typically consist of a IDL part that is independent of the physical layout of the data, and a physical layer specification such as IIOP that describes the physical layout of the data. By contrast, DataScript does not prescribe how applications lay out data, it merely provides a language to describe how they do.

5 Conclusion

We have introduced DataScript, a specification language for describing physical data layouts, and we presented a Java language binding for DataScript specifications. DataScript specifications allow casual scripting with data whose physical layout have been described in DataScript. DataScript is a simple-to-understand constraint-based language. We believe that such a specification language should become a standard part of the tool and skill set of developers, in the same way that compiler-compilers such as yacc or the use of regular expressions in languages such as Perl have become tools that are in everyday use.

References

1. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.
2. IEEE Standards Board. *IEEE Std 802.11–1997*, 1997.
3. Iyengar, S. *Java Metadata Interface (JMI Specification)*, Nov. 2001. Version 1.0 Update Public Review Draft Specification.
4. Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Jan. 1997.
5. Lyon, B. Sun remote procedure call specification. Tech. rep., Sun Microsystems, 1984.
6. McCann, P. J., and Chandra, S. Packet Types: abstract specification of network protocol messages. In *Proceedings of the SIGCOMM '00 Conf.* (Stockholm, Sweden, Aug. 2000).
7. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, June 1999. Revision 2.3. OMG Document formal/98–12–01. Part of the CORBA 2.3 specification.
8. Object Management Group. *Meta Object Facility (MOF) Specification*, Mar. 2000. Revision 1.3 OMG Document 00-04-03.
9. Tools Interface Standards (TIS) Committee. *ELF: Executable and Linkable Format: Portable Formats Specification, Version 1.2*, May 1995.