DataScript Language Overview

Harald Wellmann <HWellmann@harmanbecker.com>

Revision History

Revision 0.1	20 July 2006 Initial version.
Revision 0.2	16 November 2006 Added section on Comments.
Revision 0.3	3 December 2006 Added section on Packages and Imports.
Revision 0.4	11 January 2007 Added section on Subtypes.

Table of Contents

1. Introduction

There are thousands of languages for modelling abstract datatypes. For some of these, the binary representation of the defined types is implementation dependent and of no concern. Others do provide a binary encoding, but there is usually no way to retrofit an abstract specification to an existing binary format.

DataScript [Back] is a language for modelling binary datatypes or bitstreams. It was designed by Godmar Back to unambiguously define complex binary data formats. His reference implementation of a DataScript compiler also includes a Java code generator producing Java classes which are able to read and write a binary stream that complies with a DataScript specification.

After evaluating different data modelling languages and toolsets, DataScript was selected by the authors as the most promising approach for formally defining a car navigation database format.

While Back's reference implementation [DataScript] provided a great start, it does not fully implement the semantics introduced in his specification. In addition, we found that some language extensions were desirable to better support our specific requirements. For this reason, we branched off our own DataScript project from Back's reference implementation.

The present document describes a DataScript dialect supported by our implementation. It is more of a User's Manual than a formal language specification. Some features of the original DataScript specification that were semantically ambiguous, not fully implemented or simply not very important to us have been silently suppressed. New features of our own have been added and are documented in this overview. Features from the original specification not documented here may or may not be functional, but we are not actively supporting them.

2. Literals

The DataScript syntax for literal values is similar to the Java syntax. There are no character literals, only string literals with the usual escape syntax. Integer literals can use decimal, hexadecimal, octal or binary notation.

Examples:

- Decimal: `100, 4711, 255`
- Hexadecimal: `0xCAFEBABE, 0Xff`
- Octal: `044, 0377`
- Binary: `111b, 110b, 001B`
- String: `"You"`

Hexadecimal digits and the `x` prefix as well as the `b` suffix for binary types are case-insensitive.

String literals correspond to zero-terminated ASCII-encoded strings. Thus, the literal `"You"` corresponds to a sequence of 4 bytes equal to the binary representation of the integer literal `0x596F7500`. The representation of strings which are not ASCII-encoded (e.g. ISO 8859-1 or UTF-8) is currently undefined and shall be added to a future version of DataScript.

3. Base Types

3.1. Integer Base Types

DataScript supports the following integer base types

- Unsigned Types: `uint8, uint16, uint32, uint64`
- Signed Types: `int8, int16, int32, int64`

These types correspond to unsigned or signed integers represented as sequences of 8, 16, 32 or 64 bits, respectively. Negative values are represented in two's complement, i.e. the hex byte `FF` is 255 as `uint8` or -1 as `int8`.

The default byte order is big endian. Thus, for multi-byte integers, the most significant byte comes first. Within each byte, the most significant bit comes first.

Example: The byte stream `02 01` (hex) interpreted as `int16` has the decimal value 513. As a bit stream, this looks like `0000 0010 0000 0001`. Bit 0 is `0`, bit 15 is `1`.

3.2. Bit Field Types

A bit field type is denoted by `bit:1, bit:2`, ... The colon must be followed by a positive integer literal, which indicates the length of the type in bits. The length is not limited. A bit field type corresponds to an unsigned integer of the given length. Thus, `bit:16` and `uint16` are equivalent.

Signed bit field types are not supported.

Variable length bit field types can be specified as `bit<`*expr*`>`, where *expr* is an expression of integer type to be evaluated at run-time.

3.3. String Types

A string type is denoted by `string`. It is represented by a zero terminated sequence of bytes.

4. Enumeration Types

An enumeration type has a base type which is an integer type or a bit field type. The members of an enumeration have a name and a value which may be assigned explicitly or implicitly. A member that does not have an initializer gets assigned the value of its predecessor incremented by 1, or the value 0 if it is the first member.

```
enum bit:3 Color
{
    NONE  = 000b,
    RED   = 010b,
    BLUE,
    BLACK = 111b
};
```

In this example, `BLUE` has the value 3. When decoding a member of type `Color`, the decoder will read 3 bits from the stream and report an error when the integer value of these 3 bits is not one of 0, 2, 3 or 7.

An enumeration type does not provide its own lexical scope. The member names are and must be unique in the enclosing scope. Thus, if `Color` is defined at global scope, other enumerations at global scope may not contain a member named `NONE`.

5. Compound Types

5.1. Sequence Types

A sequence type is the concatenation of its members. There is no padding or alignment between members. Example:

```
MySequence
{
    bit:4    a;
    uint8    b;
    bit:4    c;
};
```

This type has a total length of 16 bits or 2 bytes. As a bit stream, bits 0-3 correspond to member `a`, bits 4-11 represent an unsigned integer `b`, followed by member `c` in bits 12-15. Note that member `b` overlaps a byte bound-

ary, when the entire type is byte aligned. But `MySequence` may also be embedded into another type where it may not be byte-aligned.

5.2. Union Types

A union type corresponds to exactly one of its members, which are also called branches.

```
union VarCoordXY
{
    CoordXY8    coord8  : width == 8;
    CoordXY16   coord16 : width == 16;
};
```

In this example, the union `VarCoordXY` has two branches `coord8` and `coord16`. The syntax of a member definition is the same as in sequence types. However, each member should be followed by a constraint. This is a boolean expression introduced by a colon. The terms involved in the constraint must be visible in the scope of the current type at compile time and must have been decoded at runtime before entering the branch.

The decoding semantics of a union type is a trial-and-error method. The decoder tries to decode the first branch. If a constraint fails, it proceeds with the second branch, and so on. If all branches fail, a decoder error is reported for the union type.

A branch without constraints will never fail, so any following branches will never be matched. This can be used to express a default branch of a union, which should be the last member.

5.3. Constraints

A constraint may be specified for any member of a compound type, not just for selecting a branch of a union. In a sequence type, after decoding a member with a constraint, the decoder checks the constraint and reports an error if the constraint is not satisfied.

There is a shorthand syntax for a constraint that tests a field for equality. *Type fieldName* `=` *expr*`;` is equivalent to *Type fieldName* `:` *fieldName* `==` *expr*`;`

5.4. Optional Members

A sequence type may have optional members:

```
ItemCount
{
    uint8    count8;
    uint16   count16  if count8 == 0xFF;
};
```

An optional member has an `if` clause with a boolean expression. The member will be decoded only if the expression evaluates to true at run-time.

Optional members are a more compact and convenient alternative to a union with two branches one of which is empty.

6. Array Types

6.1. Fixed and Variable Length Arrays

An array type is like a sequence of members of the same type. The element type may be any other type, except an array type. (Two dimensional arrays can be emulated by wrapping the element type in a sequence type.)

The length of an array is the number of elements, which may be fixed (i.e. set at compile-time) or variable (set at run-time). The elements of an array have indices ranging from 0 to *n*-1, where n is the array length.

The notation for array types and elements is similar to C:

```
ArrayExample
{
    uint8    header[256];
    int16    numItems;
    Element  list[numItems];
};
```

`header` is a fixed-length array of 256 bytes; list is an array with *n* elements, where *n* is the value of `numItems`. Individual array elements may be referenced in expressions with the usual index notation, e.g. `list[2]` is the third element of the `list` array.

Constraints on all elements of an array can be expressed with the `forall` operator, see Section 8.3.2, “Quantified Expression”.

6.2. Implicit Length Arrays

An array type may have an implicit length indicated by an empty pair of brackets. In this case, the decoder will continue matching instances of the element type until a constraints fail or the end of the stream is reached.

```
ImplicitArray
{
    Element    list[];
};
```

The length of the list array can be referenced as `lengthof list`, see Section 8.2.4, “lengthof Operator”.

7. Labels and Offsets

The name of a member of integral type may be used as a label on another member to indicate its byte offset in the enclosing sequence:

```
Tile
{
    TileHeader    header;
    uint32        stringOffset;
    uint16        numFeatures;

stringOffset:
    StringTable   stringTable;
};
```

In this example, the byte offset of member `stringTable` from the beginning of the `Tile` instance is given by the value of `stringOffset`.

The offset of a label is relative to the enclosing sequence by default. If the offset is relative to some other type containing the current one, this is indicated by a global label, where the type name is used as a prefix, followed by a double colon:

```
Database
{
    uint32    numTiles;
    Tile      tiles[numTiles];
};

Tile
{
    TileHeader    header;
    uint32        stringOffset;
    uint16        numFeatures;
```

```
Database::stringOffset:
    StringTable   stringTable;
};
```

8. Expressions

The semantics of expression and the precedence rules for operators is the same as in Java, except where stated otherwise. DataScript has a number of special operators `sizeof, lengthof, is` and `forall` that will be explained in detail below.

The following Java operators have no counterpart in DataScript: `++, --, >>>, instanceof`.

8.1. Binary Operators

8.1.1. Arithmetic Operators

The integer arithmetic operations include `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo). In addition, there are the shift operators `<<` and `>>`.

8.1.2. Relational Operators

There are the following relational operators for integer expressions: `==` (equal to), `!=` (not equal to), `<` (less than), `<` (less than or equal), `>` (greater than), `>=` (greater than or equal).

The equality operators `==` and `!=` may be applied to any type

8.1.3. Boolean operators

The boolean operators `&&` (and) and `||` (or) may be applied to boolean expressions.

8.1.4. Bit operators

The bit operators `&` (bitwise and), `|` (bitwise or), `^` (bitwise exclusive or) may be applied to integer types.

8.1.5. Assignment operators

The assignment operator `=` and the combined assignment operators `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=` have the usual semantics.

8.1.6. Comma operator

The comma operator `,` evaluates to the expression on the right hand side.

8.1.7. Postfix operators

The postfix operators include `[]` (array index), `()` (instantiation with argument list), and `.` (member access).

8.2. Unary Operators

8.2.1. Boolean Negation

The negation operator `!` is defined for boolean expressions.

8.2.2. Integer operators

For integer expressions, there are `+` (unary plus), `-` (unary minus) and `~` (bitwise complement).

8.2.3. sizeof Operator

The `sizeof` operator returns the size of a type or an expression in bytes. `sizeof` may not be used, when the size in bits is not divisible by 8. When `sizeof` is applied to a type name, the size of the type must be fixed and known at compile time. When `sizeof` is a applied to a member, it refers to the actual size of the member after decoding.

8.2.4. lengthof Operator

The `lengthof` operator may be applied to an array member and returns the actual length (i.e. number of elements of an array.Thus, given `int32 a[5]`, the expression `lengthof a` evaluates to 5. This is not particularly useful for fixed or variable length arrays, but it is the only way to refer to the length of an implicit length array.

8.2.5. is Operator

The `is` operator can be applied to two field names, e.g. `x is y`. `x` must be a member of union type, and `y` must be one of the branch names of that union. The expression is true if and only if the decoder has selected branch y for the union.

8.3. Ternary Operators

8.3.1. Conditional Expression

A conditional expression *booleanExpr* ? *expr1* : *expr2* has the value of *expr1* when *booleanExpr* is true. Otherwise, it has the value of *expr2*.

8.3.2. Quantified Expression

A quantified expression has the form `forall` *indexIdentifier* `in` *arrayExpr* `:` *booleanExpr*. The quantified expression is true if and only if the *booleanExpr* is true for all indices of the array. This is only useful when the boolean expression after the colon involves the array expression and the index identifier from the left hand side.

Example: The constraint

```
forall i in a : (i == 0) || (a[i] == a[i-1]+1)
```

means the elements of `a` are a sequence of consecutive integers.

8.4. Operator Precedence

In the following list, operators are grouped by precedence in ascending order. Operators on the bottom line have the highest precedence and are evaluated first. All operators on the same line have the same precedence and are evaluated left to right, except assignment operators which are evaluated right to left.

- comma
- assignment
- `forall`
- `? :`
- `||`
- `&&`
- `|`

- ^
- &
- == !=
- < > <= >=
- << >>
- + -
- * / %
- cast
- unary + - ~ !
- sizeof lengthof
- [] () . is

9. Nested Types

DataScript syntax permits the definition of nested types, however, it is not easy to define the semantics of such types in a consistent way. For the time being, the only supported use is a sequence type definition within a sequence or union field definition, or a union type definition within a sequence field definition, and even this should be avoided in favour of a reference to a type defined at global scope. Example:

```
VarCoord
{
    uint8        width;
    union
    {
        {
            int16    x;
            int16    y;
        } coord16 : width == 16;
        {
            int32    x;
            int32    y;
        } coord32 : width == 32;
    } coord;
};
```

The sequence type `VarCoord` contains the member `coord` which has a nested union type definition. This union type has two members each of which is a nested sequence type. All nested types in this example are anonymous, but this it not necessary.

The nested type definitions can be avoided as follows:

```
VarCoord
{
    uint8    width;
    Coords   coords;
};

union Coords
{
    Coord16    coord16 : VarCoord.width == 16;
    Coord32    coord32 : VarCoord.width == 32;
};
```

```
Coord16
{
    int16    x;
    int16    y;
};

Coord32
{
    int32    x;
    int32    y;
};
```

Note that the constraints for the members of the `Coords` union refer to the containing type `VarCoord`. This is explained in more detail in the following section.

10. Member Access and Contained Types

The dot operator can be used to access a member of a compound type: the expression *f.m* is valid if

- *f* is a field of a compound type *C*
- The type *T* of *f* is a compound type.
- *T* has a member named *m*.

The value of the expression *f.m* can be evaluated at run-time only if the member *f* has been evaluated before.

There is a second use of the dot operator involving a type name:

At run-time, each compound type *C* (except the root type) is contained in a type *P* which has a member of type *C* which is currently being decoded. Within the scope of *C*, members of the parent type *P* may be referenced using the dot operator *P.m*.

The containment relation is extended recursively: If *C* is contained in *P* and *P* is contained in *Q*, then *Q.m* is a valid expression in the scope of *C*, denoting the member *m* of the containing type *Q*.

Example:

```
Header
{
    uint32    version;
    uint16    numItems;
};


Message
{
    Header    h;
    Item      items[h.numItems];
};

Item
{
    uint16    p;
    uint32    q if Message.h.version >= 10;
};
```

Within the scope of the `Message` type, `header` refers to the field of type `Header`, and `header.numItems` is a member of that type. Within the scope of the `Item` type, the names `h` or `Header` are not defined. But `Item` is contained in the Message type, and `h` is a member of `Message`, so `Message.h` is a valid expression of type `Header`, and `Message.h.version` references the `version` member of the `Header` type.

11. Parameterized Types

The definition of a compound type may be augmented with a parameter list, similar to a parameter list in a Java method declaration. Each item of the parameter list has a type and a name. Within the body of the compound type definition, parameter names may be used as expressions of the corresponding type.

To use a parameterized type as a field type in another compound type, the parameterized type must be instantiated with an argument list matching the types of the parameter list.

For instance, the previous example can be rewritten as

```
Header
{
    uint32    version;
    uint16    numItems;
};


Message
{
    Header    h;
    Item(h)   items[h.numItems];
};

Item(Header header)
{
    uint16    p;
    uint32    q if header.version >= 10;
};
```

When the element type of an array is parameterized, a special notation can be used to pass different arguments to each element of the array:

```
Database
{
    uint16                          numBlocks;
    BlockHeader                     headers[numBlocks];
    Block(headers[blocks$index])    blocks[numBlocks];
};

BlockHeader
{
    uint16 numItems;
    uint32 offset;
};

Block(BlockHeader header)
{
Database::header.offset:
    Item    items[header.numItems];
};
```

`blocks$index` denotes the current index of the `blocks` array. The use of this expression in the argument list for the `Block` reference indicates that the i-th element of the `blocks` array is of type `Block` instantiated with the i-th header `headers[i]`.

12. Subtypes

A subtype definition defines a new name for a given type, optionally in combination with a constraint. When the constraint is omitted, this is rather like a `typedef` in C:

```
subtype uint16  BlockIndex;
```

```
Block
{
    BlockIndex    index;
    BlockData     data;
};
```

A constraint in the subtype definition is, as usual, a boolean expression introduced by a colon which may contain the keyword `this` to refer to the current type:

```
subtype uint16 BlockIndex : 1 <= this && this < 1024;
```

Subtype constraints will be checked by the decoder for every occurrence of the given subtype in a field definition.
Implementation note: Subtypes were introduced in version rds 0.7. Subtype constraints are not yet implemented.

13. Comments

13.1. Standard Comments

DataScript supports the standard comment syntax of Java or C++. Single line comments start with `//` and extend to the end of the line. A comments starting with `/*` is terminated by the next occurrence of `*/`, which may or may not be on the same line.

```
// This is a single-line comment.


/* This is an example
   of a multi-line comment
   spanning three lines. */
```

13.2. Documentation Comments

To support inline documentation within a DataScript module, multi-line comments starting with `/**` are treated as special documentation comments. The idea and syntax are borrowed from Java(doc). A documentation comment is associated to the following type or field definition. The documentation comment and the corresponding definition may only be separated by whitespace.

```
/**
 * Traffic flow on links.
 */
enum bit:2 Direction
{
    /** No traffic flow allowed */
    NONE,
    /** Traffic allowed from start to end node. */
    POSITIVE,
    /** Traffic allowed from end to start node. */
    NEGATIVE,
    /** Traffic allowed in both directions. */
    BOTH
};
```

The content of a documentation comment, excluding its delimiters, is parsed line by line. Each line is stripped of leading whitespace, a sequence of asterisks (`*`), and more whitespace, if present. After stripping, a comment is composed of one or more paragraphs, followed by zero or more tag blocks. Paragraphs are separated by lines.

A line starting with whitespace and a keyword preceded by an at-sign (`@`) is the beginning of a tag block and also indicates the end of the preceding tag block or comment paragraph.

The only tag currently supported is `@param`, which is used for documenting the arguments of a parameterized

type. The documentation comment should contain one `@param` block for each argument in the correct order. The `@param` tag is followed by the parameter name and the parameter description. The parameter name must be enclosed by whitespace.

```
/**
 * This type takes two arguments.
 * @param  arg1   The first argument.
 * @param  arg2   The second argument.
 */
ParamType(Foo arg1, Blah arg2)
{
    ...
};
```

14. Packages and Imports

Complex DataScript specifications should be split into multiple packages stored in separate source files. Every user-defined type belongs to a unique package. There is an unnamed default package used for files without an explicit package declaration. Types from other packages can be imported into the current package.

The package and import syntax and semantics follow the Java example.

```
package map;

import common.geometry.*;
import common.featuretypes.*;
```

Package and file names are closely related. The above example declares a package `map` stored in a source file `map.ds`. The import declarations direct the parser to locate and parse source files `common/geometry.ds` and `common/featuretypes.ds` and to import *all* types from these packages into the current scope. Imported files may again contain import declarations.

Unlike Java, DataScript cannot import individual types from another package (e.g. `import common.geometry.Line;`), nor is it possible to refer to an external type via its fully qualified name without an import declaration.

References

[Back] Godmar Back, *DataScript - a Specification and Scripting Language for Binary Data*. Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002), published as LNCS 2487. ACM. Pittsburgh, PA. October 2002. pp. 66-77. http://www.cs.vt.edu/~gback/papers/gback-datascript-gpce2002.pdf

[DataScript] DataScript Reference Implementation, http://datascript.sourceforge.net.