The Glasgow Packet Language: Type System and Execution Model

Stephen McQuistin
University of Glasgow

Vivian Band
University of Glasgow

Colin Perkins
University of Glasgow

7th November 2019

Abstract

This memo defines the intermediate representation used in the implementation of the Glasgow Packet Language. It also describes the execution model used to parse protocols described in that language.

1 Introduction

In order to generate a parser for a protocol, it's necessary to describe the format of protocol data units (PDUs). These can be described by a set of types that represent the objects to be parsed, along with constraints on the parsed values. If there are multiple ways of describing these types, it's necessary to have a common *intermediate representation* into which all those input formats can be converted, and from which parsers can be generated.

We describe such an intermediate representation in this memo. It specifies a set of internal types used by the parsers, and constructors for representable types that describe the PDUs to be parsed. The execution model of the parsers is also described.

2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [1, 2] when, and only when, they appear in all capitals, as shown here.

Runtime refers to software that supports the type system that this document describes. The runtime instantiates a set of primitive types, provides functions for the construction of other types, and synthesises the primitive and constructed types into a state that is suitable for the generation of parser or serialisation code.

Implementation refers to code that is generated, with support from the runtime, to parse or serialise the objects that the type system describes.

3 Type System

Types can be *internal* or *representable*. An internal type is an artefact of the runtime, and cannot be parsed or serialised. A representable type describes something that can be parsed or serialised.

The runtime provides a set of *type constructors* that can be used to create different kinds of type, and instantiates a set of primitive types.

Each type has a *kind*, a *name*, and a *value*. A type implements one or more *traits*. Traits define *methods* that can operate on instances of that type.

A type name is formed of upper- and lower-case ASCII letters, digits, and dollar signs (`A-Za-z0-9$`). It must begin with an upper case letter (`A-Z`). It is an error to define two types with the same name.

3.1 Traits

A *trait* defines a named collection of methods. A type may implement one or more traits. A type that implements a trait gains an implementation of each of the trait's methods.

A trait definition specifies the name of the trait and defines the methods it provides.

A trait name is comprised of upper and lower case ASCII letters, digits, or dollar signs (`A-Za-z0-9$`). It MUST begin with an upper case letter (`A-Z`). Trait names occupy a different namespace to type names.

A method definition specifies its name, parameters, and return type. Methods take one or more parameters. Each parameter has a name and a type. The first parameter of a method MUST have the name `self` and MUST have the type parameterised as `Self`. Type parameter `Self` refers to the type that implements the trait. Other parameters, if any, MUST have names that are unique within the method definition.

Types used in method definitions MAY be concrete types (i.e., types specified when the trait is defined) or type parameters. Concrete types for any type parameters MUST be provided when a type that implements the trait is constructed.

Method names and parameter names are formed of upper and lower case ASCII letters, digits, dollar signs, or underscores (`A-Za-z0-9$_`). They MUST begin with a lower case letter (`a-z`). It is an error to define two methods that have the same name within a given trait.

When a method is called on an instance of a type that implements a trait, that instance will be passed as the first parameter of the method.

It is not possible for further traits to be defined beyond those defined by the runtime.

The runtime defines the following traits:

- The `Value` trait defines the following two methods that can be used to get and set the value of an instance of a type:
  - `get(self: Self) → Self`
  - `set(self: Self, value: Self) → Nothing`

  The syntax *name(parameters) → return_type* is used to describe a method. The parameters are a list of *name: type* pairs. `Nothing` is a primitive representable type defined in Section 3.3.1.
- The `Sized` trait defines a method that can be used to get the size, in bits, of an instance of a type:
  - `size(self: Self) → Number`

  `Number` is a primitive internal type defined in Section 3.2.1. All representable types MUST implement the `Sized` trait. `size` MAY return an expression (Section 3.4) that evaluates to a `Number`.
- The `IndexCollection` trait defines methods that can be used to get and set the values of elements of an array-like data structure, which has multiple, numerically indexed elements:
  - `get(self: Self, index: Number) → `*T*
  - `set(self: Self, index: Number, value: `*T*`) → Nothing`

  Type parameter *T* MUST NOT be `Self` and MUST NOT be `Nothing`. The type parameterised by *T* MUST be specified when the trait is implemented.
- The `Equality` trait defines methods that can be used to compare to instances of a type for equality:
  - `eq(self: Self, other: Self) → Boolean`
  - `ne(self: Self, other: Self) → Boolean`

  `Boolean` is a primitive internal type defined in Section 3.2.1.
- The `Ordinal` trait defines methods that can be used to compare the values of instances of a type:
  - `lt(self: Self, other: Self) → Boolean`
  - `le(self: Self, other: Self) → Boolean`
  - `gt(self: Self, other: Self) → Boolean`
  - `ge(self: Self, other: Self) → Boolean`
- The `BooleanOps` trait defines methods that can be used to perform boolean operations on, and between, instances of a type:
  - `and(self: Self, other: Self) → Boolean`
  - `or(self: Self, other: Self) → Boolean`
  - `not(self: Self) → Boolean`
- The `ArithmeticOps` trait defines methods that can be used to perform arithmetic operations on instances of a type:
  - `plus(self: Self, other: Self) → Self`
  - `minus(self: Self, other: Self) → Self`
  - `multiply(self: Self, other: Self) → Self`
  - `divide(self: Self, other: Self) → Self`
  - `modulo(self: Self, other: Self) → Self`
- The `NumberRepresentable` trait defines a method that can be used to represent a type as an `Number` type:
  - `to_number(self: Self) → Number`
- The `TypeEquality` trait defines a method that can be used to determine if two type instances are of the same type:
  - `is_type(self: Self, other: `*T*`) → Boolean`

  where `is_type` returns `true` if `Self` and *T* are the same type, and `false` otherwise.

3.2 Internal types

The runtime defines two primitive internal types (`Boolean` and `Number`) and type constructors for three kinds of internal type (`Function`, `Protocol`, and `Context`).

3.2.1 Primitive type

The runtime defines two primitive internal types:

- The `Boolean` type is a boolean value, either `true` or `false`. It implements the traits `Value`, `Equality` (two `Booleans` are equal if they have the same value), and `BooleanOps`.
- The `Number` type is a positive integral value. It implements the traits `Value`, `Equality`, `Ordinal`, and `ArithmeticOps`. The size of the `Number` type is implementation defined.

3.2.2 Functions

A `Function` type can be constructed to represent the signature of a function to be provided by the implementation of the protocol. The code comprising the body of the function is not captured by the intermediate representation.

The `Function` type constructor is parameterised by the *name* of the function, the list of *parameters* that the function takes (each with its own name and type), and the function's *return type*. The return type `Nothing` is used for functions that return no value.

Function names and parameter names are formed of upper and lower case ASCII letters, digits, dollar signs, or underscores (`A-Za-z0-9$_`), and MUST begin with a lower case letter (`a-z`). Function names are defined in the global type namespace. Each parameter within a function definition MUST have a unique name.

3.2.3 The Parsing Context

A `Context` type represents a sequence of fields, possibly of different types, that can be accessed in the parsing of other types, where needed.

The `Context` constructor is parameterised by the details of the *fields* it includes.

The `Context` holds internal state for the parser, and does not represent a PDU, or any part of a PDU, of the protocol being parsed. The `Context` is defined along with the other *type definitions* in a `Protocol` (defined in Section 3.2.4), however, since it is specific to a protocol.

Each field has a name that is formed of upper and lower case ASCII letters, digits, dollar signs, or underscores (`A-Za-z0-9$_`). The field name MUST begin with a lower case letter (`a-z`). Field names must be unique within the `Context`.

Each field has a type. That type MUST be a previously defined representable type (primitive representable types and type constructors for representable types are defined in Section 3.3).

3.2.4 Protocols

A `Protocol` type can be constructed to represent the types and protocol data units (PDUs) that form a protocol.

The `Protocol` type constructor is parameterised by the *name* of the protocol, a set of *type definitions* for types used by the protocol parser, and the list of type names of *PDUs* used in the protocol.

Any type constructor for a representable type (defined in Section 3.3) or a function type MAY be included in the set of type definitions. In addition, the set MUST include one context type constructor. This context MUST include a field named "`data_size`" with the type `Number`. As described in Section 4, the implementation will set this field to the size of the current data unit being parsed or serialised.

The list of PDU type names MAY be empty, although this represents a protocol that is useless. PDU type names can be type names of any representable type whose constructor is part of the set of type definitions of the protocol.

3.3 Representable types

The runtime specifies two primitive representable types (`Nothing` and `DataUnit`). In addition, the runtime provides type constructors for `BitString`, `Option`, `Array`, `Struct`, and `Enum` types. Finally, a mechanism by which new types can be derived from existing representable types is also defined.

The base representable type is a `DataUnit` (defined in Section 3.3.1). All data for parsing or serialisation is initially represented within the language as a `DataUnit`.

`BitString` and `Option` types cannot be initialised from another type. All other representable types MUST be associated with a single initialisation function, with the following signature:

$$\texttt{init(from: } T\texttt{)} \rightarrow \texttt{Option<Self>}$$

where T is any previously defined representable type and MUST NOT be `Nothing`, and `Self` refers to the type that is being instantiated. A type's `init` function constructs and initialises an instance of that type, from an instance of the type T. These functions exist within the namespace for all functions, as described in Section 3.2.2.

If T is **not** a BitString type, then a `init` function MUST be specified when the `Self` type is being constructed. Otherwise, and if an initialisation function is not specified, an initialisation function will be generated by the runtime, with T as a BitString type constructed by the runtime, whose length is appropriate for the type `Self`.

There are two properties that follow from these constraints on `init` functions. Firstly, given that runtime defines only one non-`Nothing` type (`DataUnit`), it is always possible to construct a chain from a `DataUnit` (the base type) to any other representable type via a series of `init` functions. Secondly, given that each type has only a single initialisation function, only one such chain of `init` functions exists for a given representable type. This allows for the unambiguous parsing of incoming data.

3.3.1 Primitive types

The runtime defines two primitive representable types:

- The `Nothing` type is the empty type. It implements the `Sized` trait and has size 0.
- The `DataUnit` type is the base type. All data for parsing or serialisation is initially represented within the language as this type. It is constructed as a `BitString` type that has size equal to the "*data_size*" `Context` field.

3.3.2 Bit Strings

Representable types of kind `BitString` can be constructed to represent multi-bit values that can be parsed or serialised.

The `BitString` type constructor takes as parameters the *name* of the new bit string type and a *size* in bits. The size may be set to `Nothing`.

Instances of `BitString` types implement the `Sized` trait. The `size()` method of `BitString` types returns *size*, if *size* is a `Number` or an expression that evaluates to a `Number`. If *size* is `Nothing`, then the return value of `size()` is inferred by the runtime (see Section 4.1).

Instances of `BitString` types also implement the `Value`, `Equality`, and `NumberRepresentable` traits.

3.3.3 Options

Representable types of kind `Option` can be constructed to represent optional types that are either the `Nothing` type *or* another representable type.

The `Option` type constructor takes as parameters the *name* of the new type and another *reference type*. The `Option` type is either `Nothing` or the reference type.

Instances of the `Option` type implement the `TypeEquality` trait. `is_type(T)` returns `true` if the Option is the type *T*, and returns `false` otherwise.

Instances of `Option` types implement the `Sized` trait. `size()` returns an `IfElse` expression, where the condition is `This.is_type(reference type)`, the *if true* expression is the size of the reference type, and the *if false* expression is the size of the `Nothing` type.

3.3.4 Arrays

Representable types of kind `Array` can be constructed to represent a sequence of elements that can be parsed or serialised.

The `Array` type constructor takes as parameters the *name* of the new array type, the *element type*, the *length* of the new array, and an *init* function, as described above. The element type must be a previously defined representable type. The length is the number of elements contained in the array. The length may be `Nothing`.

Type parameter `Self` in the `init` function of an `Array` refers to the new type with name *name*.

Instances of the `Array` type implement the `Sized` trait. The size of the array is equal to the *length* of the array multiplied by the `size()` of the element type. If *length* is `Nothing`, then the `size()` of the array will be inferred by the runtime (see Section 4.1).

Instances of `Array` types also implement the `Equality`, and `IndexCollection` traits. Two arrays are equal if they have the same element type and length, and their elements are equal.

3.3.5 Structure Types

Representable types of kind `Struct` can be constructed to represent a sequence of fields, with possibly different types, that can be parsed and serialised.

The `Struct` type constructor takes as parameters the *name* of the new structure type, a list of *fields*, a possibly empty list of *constraints* on those fields, a possibly empty list of *actions* that are to be carried out once the structure has been parsed, and an `init` function, as described above.

Type parameter `Self` in the `init` function of a `Struct` refers to the new type with name *name*.

Fields are constructed with a *name* for the field, the *type* of the field, and an expression indicating if the field *is present* in a particular instantiation of the structure type. Structure types MUST contain at least one field.

Field names are formed of upper and lower case ASCII letters, digits, dollar signs, or underscores (`A-Za-z0-9$_`). The field name MUST begin with a lower case letter (`a-z`).

Field types MUST have been previously defined. Fields can be of `BitString`, array, structure type, or enumerated type, or a type derived from such a type. The names of each field MUST be unique within a structure definition, but several fields can have the same type.

Each field has an *is present* expression that indicates whether that field is present in a particular instantiation of the structure type. This is used to model data formats that contain optional fields. Expressions are described in Section 3.4. The *is present* expression MUST evaluate to a `Boolean` value.

A structure type implements the `Sized` trait. The size of a structure is inferred by the runtime.

A structure type also implements the `Equality` trait. Two structures are equal if they are the same type, and the corresponding fields are equal. Two fields are equal if they have the same *name*, the same *type*, and the same *is present* expression that evaluates to the same value.

A structure type is parameterised by a set of constraints (i.e., boolean expressions that MUST evaluate to `True`) on the fields. The set of constraints MAY be empty. Expressions are described in Section 3.4.

Finally, on the successful parsing of the structure type, the list of *actions* is evaluated. The list of actions MAY be empty. The expressions in the list of actions MUST be tree expressions, and each expression MUST return `Nothing`. Actions are typically expected to update the parsing context (see Section 3.2.3).

3.3.6 Enumerated Types

Representable types of kind `Enum` can be constructed to represent data that can exist as one of several possible variants when parsed or serialised.

The `Enum` type constructor takes as parameters the *name* of the new `Enum` type, an ordered list of the types of the *variants* that the `Enum` might take, and an *init* function, as described above.

Type parameter `Self` in the `init` function of an `Enum` refers to the new type with name *name*.

Variant types MUST have been previously defined, and the variants list MUST NOT be empty. Variants of an enumerated type have their own types, but are otherwise unnamed. Variants can be of `BitString`, array, structure type, or enumerated type, or a subtype of such a type.

The `Enum` will take the type of the first variant type that the data can be successfully parsed or serialised as. Parsing or serialisation will be attempted for each variant type in turn until successful, or until the list of variant types has been exhausted.

An enumerated type implements the `TypeEquality` trait. `is_type(`*T*`)` returns `true` if the enumerated type's variant is of type *T*, and returns `false` otherwise.

An enumerated type implements the `Sized` trait. `size()` returns an `IfElse` expression, where the condition is `This.is_type(`*variant*`)`, the *if true* expression is the size of the *variant* type, and the *if false* expression is either a further `IfElse` expression conditional upon the next variant in the list of variants (if there are two or more remaining variants), or the `size()` of the final variant in the list.

3.3.7 Derived Types

The type system includes shorthand for the creation of new types from other types. A derived type has the same representation and traits as the type it is derived from, but has a new name and is distinct from the original type. It MAY also implement additional traits.

The type constructor of a derived type is parameterised by the *name* of the new type, the *type* that it is derived from, a (possibly empty) list of additional *traits* the new type implements, and an *init* function, as described above.

Type parameter `Self` in the `init` function of a derived type refers to the new type with name *name*.

A new type can be derived from any previously defined representable type. It is an error to define a new type that has the same name as an existing type. A new type cannot be derived from itself.

Once constructed, derived types do not have any relationship with the type they are derived from.

3.4 Expressions

Expressions are split into two classes: tree (`MethodInvocation`, `FunctionInvocation`, `FieldAccess`, `ContextAccess`, and `IfElse`) and leaf (`This`, and `Constant`).

Expressions MUST evaluate to a result that is of a previously defined type.

3.4.1 Tree Expressions

A `MethodInvocation` expression is parameterised by a *target* expression, a *method name*, and list of *arguments* for that method. The expression evaluates to the result of calling the named method, with `self` set to the *target* expression and the other arguments set as specified. Arguments are specified as expressions.

A `FunctionInvocation` expression is parameterised by a *function name* and a list of *arguments*, specified as expressions. The expression evaluates to the result of calling the named function with the arguments specified. The *name* MUST refer to a previously defined `Function`. The set of arguments MUST contain values for all of the parameters specified in the `Function` definition, and the `value` expression for each MUST evaluate to a value matching the type of the parameter as specified in the `Function` definition.

A `FieldAccess` expression is parameterised by a *target* expression, and a *field name*. The *target* expression MUST resolve to a structure type. A `FieldAccess` expression resolves to the type of the *field name* within the *target* structure.

A `ContextAccess` expression is parameterised by a *field name*. The expression resolves to the type of the *field name* within the protocol's context. It is an error to evaluate a `ContextAccess` expression on a protocol that protocol does not contain a `Context` instance.

A `IfElse` expression is parameterised by three expressions: a *condition*, an *if true* expression and an *if false* expression. The expression resolves to the `Boolean` type. It resolves to the true expression if the condition expression evaluates to True, otherwise, it evaluates to the false expression.

3.4.2 Leaf Expressions

Expressions are always evaluated in the context of a containing type. A `This` expression resolves to that containing type.

A `Constant` expression is parameterised by a *type name* and a *value*. The expression evaluates to the specified value, whose type is specified.

4 Parser Model

The goal of the intermediate representation described in this document is to allow for the unambiguous description of the parsing and serialisation of binary protocols, from which code that supports these processes can be generated. However, the intermediate representation limits the constraints it places on the design and architecture of generated code. This allows code generation backends to produce code that is idiomatic within the target language or that fulfils other non-functional properties (such as readability or performance).

Figure 1 gives pseudocode for the parsing process, as captured by the intermediary language. Parsing begins with an incoming

```
new_type = Nothing()

# try to parse as each listed PDU type in turn
for pdu in protocol.PDUs:
  from_type = type(pdu.init.from)

  # build chain of constructors that begins with
      DataUnit
  init_chain = []
  while from_type != DataUnit:
    init_chain.append(from_type)
    from_type = type(from_type.init.from)
  input_chain.append(pdu)

  # call constructors in sequence
  new_type = input.copy()
  for init_type in init_chain:
    new_type = init_type.init(new_type)
    if type(new_type) is Nothing:
      continue # try next PDU type if an initialiser
           returns Nothing
  return new_type
```

Figure 1: Pseudocode for parsing process

PDU (input). Within the model described by the intermediate representation, this PDU originates as a DataUnit. From this DataUnit, the parser should then attempt to construct the first PDU type as listed within the Protocol. This begins by building the chain of constructors from the DataUnit to the PDU type. This chain of constructors is then called in sequence, until either parsing fails (because a constructor returns Nothing) or the PDU type is constructed successfully. If parsing fails, then the construction of the next PDU type is attempted. This continues until the list of PDU types is exhausted (in which case the incoming data cannot be parsed) or a PDU type is successfully constructed.

The intermediary representation does not specify how the generated code should represent the types described, or their traits and methods. It is only required that the logical consistency of the intermediary representation's type system is maintained. For example, the particular representation of our language's Number type is not specified; It remains that this type must be represented in a way that supports the traits and methods that apply to the Number type.

4.1 Type size inference

The constructors for representable types allow for the size of the constructed types to be unspecified. However, the runtime MUST be able to infer an expression for the size of all representable types once all types have been constructed.

The runtime performs a size inference process, starting with each type specified as a PDU in the Protocol's *PDUs* list. Size inference is a recursive process. We begin by defining the *container size*: that is, the size of the outer type. This is initially set to data_size.

The runtime infers a size for each type as follows:

BitString The size of a BitString type is set to the *container size*.

Option The size of an Option type is as defined in Section 3.3.3.

The size inference process continues by inferring the size of the Option's *reference type*, where the *container size* is unchanged.

Array The size of the array's *element type* is inferred first. Then, the array's length can be inferred as *container size* divided by the size of each element. Finally, the size of the array is set to *container size*.

Structure A structure can contain at most one field whose type has a size that is unspecified, and that must be infered by the runtime. The size of that field is infered by the runtime, with *container size* set to its existing size less the sum of the sizes of all of the other fields.

Enumerated The size of an Enum type is as defined in Section 3.3.6.

The size inference process continues by inferring the size of each of the Enum's *variants*, where the *container size* is unchanged.

5 Outstanding issues

The language specification does not yet consider:

- The role of the init function for Enum types.
- The ways in which the size inference process can fail.
- How the size inference process interacts with constraints.

6 Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/R04144X/1).

References

[1] S. Bradner. Key words for use in RFCs to indicate requirement levels. Internet Engineering Task Force, March 1997. RFC 2119.

[2] B. Leiba. Ambiguity of uppercase vs lowercase in RFC 2119 key words. Internet Engineering Task Force, May 2017. RFC 8174.