The Serialization Killer Language

Timm Felden

March 27, 2013

Abstract

This work presents an alternative to various serialization approaches. The proposed serialization mechanism is fast, robust, extensible and easy to use. These goals are achieved by not using a human readable serialized form.

To do (1)

Acknowledgements

For productive criticism: Erhard Plödereder and Martin Wittiger

To do (2)

To do (3)

1 Motivation

This paper presents an approach to serializing objects, which is tailored for usability, performance and portability. In order to achieve these goals, in contrast to XML, we will sacrifice generality and human readability of the serialized format. Unlike other general serialization mechanisms, we provide explicit support for extension points in the serialized data, in order to provide a minimum of upward compatibility and extensibility.

1.1 Related Work

There are many approaches similar to ours, but most of them have a different focus. This section shall provide a concise list of related approaches. For potential users of skill, this might also present alternatives superior for individual use cases.

The very nature of the problem requires different solutions, blablabla, skill is basically related to serialization like XML on one side and language interfaces like IDL or even JNI on the other side.

XML

XML is a file format and might in fact be used as a backend. If a human readable storage on disk is not required, a binary encoding can be used to improve load/store performance significantly.

To do (4)

XML Schema definitions

The description language itself is more or less equivalent to most schema definition languages such as XML Schema . The downside is that schema definitions have to operate on XML and can not directly be used with a binary format. There is also no way to generate code for a client language, such as Ada, from schema definitions.

To do (5)

JAXP and xmlbeansxx

For Java and C++, there are codegenerators, which can turn a XML schema file into code, which is able to deal with an xml in a similar way, as it is proposed by this work. In case of Java this is even in the standard library. The downside is, that, to our knowledge, this is only possible for Java and C++, thus it leaves us with portability issues. A minor problem of this approach is the lack of support for comment generation and the inefficient storage of serialized data. An interesting observation is, that this approach deprives xml of its flexibility advantage over our solution. To do (6)

ASN.1

Is not powerful enough to fit our purpose.

IDL

To do (7)

Is not powerful enough and seems to be outdated.

Apatche Thrift & Protobuf

Lacks subtypeing. Protobuf has a overly complex notation language. Both seem to be optimized for network protocols, thus they do not have storage pools, which are the foundation of our serialization approach and an absolute requirement for some of our features, such as hints (see section 6).

Language Specific

Language specific is language specific and can therefore not be used to interface between subsystems written different programming languages such as Ada, Java, C or Haskell. Plus not every language offers such a mechanism. E.g. C.

Language Interfaces

Language Interfaces do not permit serialization capabilities. Most language only provide interfaces for C, with varying quality and varying degree of automation. A significant problem are interfaces between languages with different memory models. Interfaces between languages with different type systems are simply unproductive:D

2 Syntax

We use the tokens `<id>`, `<string>`, `<int>` and `<comment>`. They equal C-style identifiers, strings, integer literals and comments respectively. Note that we use a comment token, which is need, because we want to emit the comments in the generated code, in order to integrate nicely into the target languages documentation system.

```
UNIT :=
  INCLUDE*
  DECLARATION*

INCLUDE :=
```

```
  ("include"|"with") <string> ";"?

DECLARATION :=
  DESCRIPTION
  <id>
  ((":"|"with"|"extends") <id>)?
  "{" FIELD* "}"

FIELD :=
  DESCRIPTION
  (CONSTANT|DATA) ";"?

DESCRIPTION :=
  (RESTRICTION|HINT)*
  <comment>?
  (RESTRICTION|HINT)*

RESTRICTION :=
  "@" <id> ("(" (R_ARG ("," R_ARG)*)? ")")? ";"?

R_ARG := ("%"|<int>)

HINT := "!" <id> ";"?

CONSTANT :=
  "const" TYPE <id> "=" <int>

DATA :=
  "auto"? TYPE <id>

TYPE :=
  ("map" MAPTYPE
  |"set" SETTYPE
  |"list" LISTTYPE
  |ARRAYTYPE)

MAPTYPE :=
  "<" BASETYPE ("," BASETYPE)+ ">"

SETTYPE :=
  "<" BASETYPE ">"

LISTTYPE :=
  "<" BASETYPE ">"

ARRAYTYPE :=
  BASETYPE
  ("[" (<id>|<int>)? "]")?

BASETYPE :=
```

```
(<id>|"annotation")
```

Note: The Grammar is LL(1).[1]

Comment: The optional `;` at the end of includes or definitions are for convenience only.

2.1 Reserved Words

The language itself has only the reserved words **annotation**, **auto**, **const**, **with**, **map**, **list** and **set**. To do (8)

However, it is strongly advised against using any identifiers which form reserved words in a potential target language, such as Ada, C++, C#, Java, JavaScript or Python. To do (9)

2.2 Examples

Listing 1: Running Example

```
/** A source code location. */
SLoc {
  i16 line;
  i16 column;
  string path;
}

Block {
  SLoc begin;
  SLoc end;
  string image;
}

IfBlock : Block {
  Block thenBlock;
}

ITEBlock : IfBlock {
  Block elseBlock;
}
```

Includes, self references

Listing 2: Example 2a

```
with "example2b.skill"

A {
  A a;
```

[1] In fact it can be expressed as a single regular expression.

```
  B b;
}
```

Listing 3: Example 2b

```
with "example2a.skill"

B {
  A a;
}
```

Unicode

The usage of non ASCII characters is completely legal, but discouraged.

Listing 4: Unicode Support

```
/** some arguably legal unicode characters. */
ö {
  ö ∀;
  ö €;
}
```

2.3 On ADTs

ADTs are represented using arrays and pairs.

ADTs showed to be useful and to increase the usability and understandability of the resulting code and file format.

3 Semantik

Bedeutung der einzelnen Schlüsselwörter

3.1 Includes

The file referenced by the with statement is processed as well. The declarations of all files reachable over `with` statements are collected, before any declaration is evaluated.

3.2 Subtypes

To do (10)

3.3 `annotation`

The type has a tag and a size, which allows it to be inserted at any annotation locations. This is useful in order to provide extension points in the file format. The file will still be readable by older implementations, which are not able to map any meaningful type into the annotation.

As we will see later, annotation can be seen as equivalent to the type definition

```
annotation {
  v64 baseTypeName;
  v64 basePoolIndex;
}
```

Of course, this is made transparent to the user.

3.4 Subtypeing

A subtype of a userdefined type can be declared by appending the keyword `with` and the supertypes name to a declaration. In order to be wellformed, the subtype must have exactly the type modifiers(i.e. indexed, tagged/annotation) of the supertype. This is because all subtypes of an indexed type will share a single storage pool.

3.5 `const`

A const field can be used in order to create guards or version numbers, as well as overwriting deprecated fields with e.g. zeroes. The deserialization mechanism has to report an error if a constant field has an unexpected value.

3.6 `auto`

The language binding will create a field with the given type, but the content is transparent to the serialization mechanism. This is useful if the inference of the content of a field is likely to be faster then storing it, e.g. if it can be inferred lazy.

3.7 Abstract Data Types

The type system has a built-in notion of arrays, maps, lists and sets. Note that all of them are, from the view of serialization, equivalent to length encodeded arrays. They are added just to make modeling easier. To do (11)

To do (12)

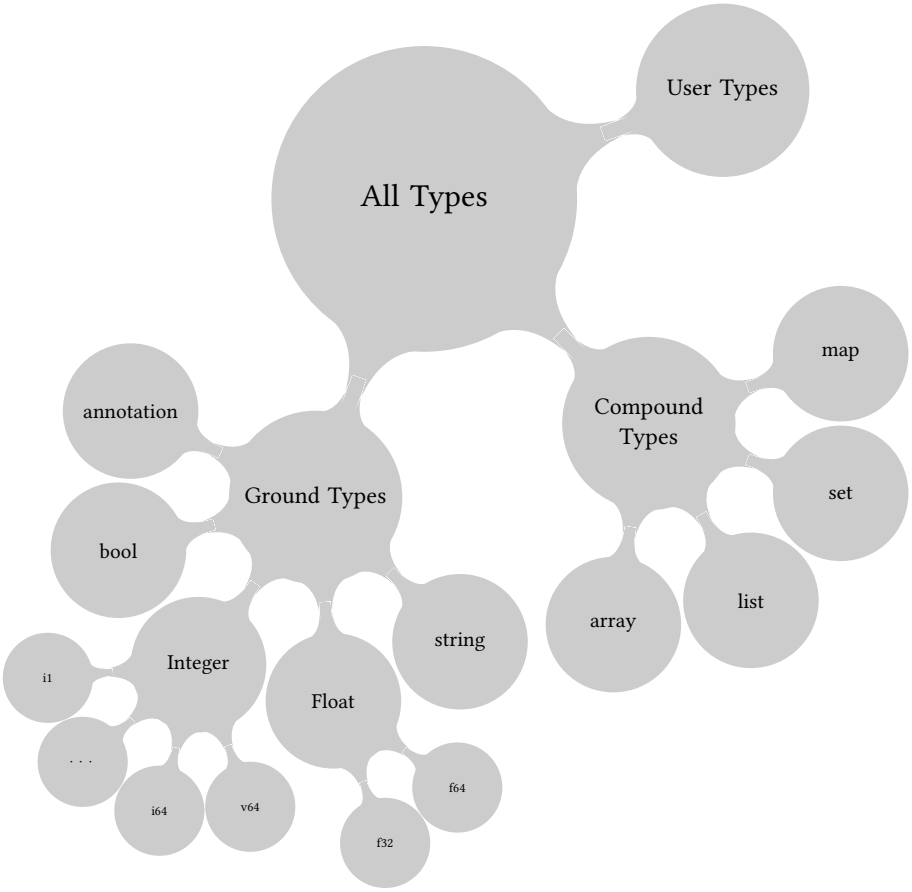
3.8 Comments

Comments provided in the skill file will be emitted into the generated code[2], thus allowing a user to profit from tooltips his IDE is likely to show him, containing this documentation. To do (13)

4 The Type System

The basic layout of the type system is:

[2]If the target language does not allow for C-Style comments, the comments will be transformed in an appropriate way.



User types can be seen as nonempty tuples over all types. It is legal to *rename* a ground type, in order to give it a special semantics. E.g. to create a time stamp by:

Listing 5: Time

```
time {
  /** seconds since 1.1.1970 0:00 UTC. */
  i64 date;
}
```

Legal Types

This section is to define the set of types, which can be used to declare legal fields inside a user type definition. Let $\mathcal{T}$ be the set of all types, $\mathcal{U} \subseteq \mathcal{T}$ be the set of user types, $\mathcal{G} \subseteq \mathcal{T}$ be the set of ground types, $\mathcal{C} \subseteq \mathcal{T}$ be the set of compound types and $\mathcal{I} \subseteq \mathcal{G}$ be the set of integer types.

In this context, we talk about an unknown, but fixed set $\mathcal{T}$, which corresponds to the contents of a set of input files. All types have unique names.

Let $\mathtt{t} \in \mathcal{U}$, then wellformedness is the smallest predicate satisfying

- If $\mathtt{s} \in \mathcal{U}$, then `t with s{...}` is a well formed type declaration and `t` is called a subtype of `s`, written $t <: s$

- All field definitions of t are *legal*.
- The subtype relation forms a forest.

Let f be a field. The field is *legal* in one of the following cases:

- If $\texttt{t} \in \mathcal{I}$ then `const t f` is a legal constant field.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `t f` is a legal field.
- For any $n \geq 2$ with $\texttt{t}_i \in \mathcal{T} \setminus \mathcal{C}$, `map<`$\texttt{t}_1$`, ` $\cdots$ `, `$\texttt{t}_n$`> f` is a legal field.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `set<t> f` and `list<t> f` are legal fields.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `t[] f` is a legal field.
- If $i \in \mathbb{N}^+$ and $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `t[`i`] f` is a legal field.
- If $\texttt{s} \in \mathcal{I}$, $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ and `s size` is a field, then, inside the same type declaration, `t[size] f` is a legal field.

Then informal short definitions is that compound types can not be nested[3] and array lengths have to be integers.

Type Order

Let $<_l$ be the lexical order. We define a partial order $\leq_t$ on $\mathcal{T}$ as follows:

- $\forall t \in \mathcal{G}, s \in \mathcal{T} \setminus \mathcal{G}.t \leq_t s$
- $\forall t \in \mathcal{C}, s \in \mathcal{U}.t \leq_t s$
- $\forall s, t \in \mathcal{U}.t \leq_t s \leftarrow s <: t$[4]
- $\forall s, t \in \mathcal{U}.t \leq_t s = t \leq_l s \leftarrow \exists S \in \mathcal{U} \cup \{\bot\}.t <: S \wedge s <: S$[5]

The informal short description is, first ground types, then compound types and user types at the end, where the forest of user types maintains its structure but is order using the lexical order of type names.

Notice, that this order corresponds to an left to right order in the types overview picture.

The missing order of compound types is left away intentionally, because it allows for the exchange of some type definition after publishing a format, e.g. `t[] f` can be exchanged with `list<t> f`.

Strings

Strings are conceptually a zero terminated sequence of utf8 encoded unicode characters. The in memory representation will try to make use of language features such as java.lang.String or std::u16string.

Strings must not contain 0 characters except for the terminating 0. If the users concept of a *string* allows such data, he has to declare his own data type.

[3] This is actually ensured by the grammar.
[4] This is *super types first.*
[5] Types with the same or no supertype are order lexically.

Compound Types

The language offers several compound types. Sets, Lists and auto sized Arrays, i.e. arrays without an explicit size, are basically views onto the same kind of serialized data, i.e. they are a length encoded list of elements of the supplied base type. Arrays are expected to have a constant size, i.e. they are not guaranteed to be resizable. Sets are not allowed to contain the same element twice. All ADTs will be mapped to their closest representation in the target language, while preserving these properties. Maps are viewed as a representation of serializable partial functions. Therefore they can contain other map types as their second type argument, which is basically an instance of currying.

NULL Pointer

The null pointer is serialized using the index 0. Conceptually, null pointers of different types are different. In fact if an annotation is a null pointer, it still has a type. However, this detail should not be observable in most languages.

4.1 Examples

This section will present some examples of ill-formed type declarations and brief explanations.

Listing 6: Legal Super Types

```
EncodedString : string {
  string encoding;
}
```

Error: The built-in type “string” can not be sub classed.

5 Invariants

Some invariants can be added to declarations and fields. These invariants can occur at the same place as comments, but can occur in any number. Invariants start with an @ followed by a predicate. Each predicate has to supply a default argument %, such that using only default arguments would not imply a restriction. If multiple predicates are annotated, the conjunction of them forms the invariant. The set of legal predicates is explained below.

If predicates, which are not directly applicable for compound types are used on compound types, they expand to the contents of the compound types, if applicable. Otherwise the usage of the predicate is illegal.

Range

Range restrictions are used to restrict integers and floats.

Applies to fields: Integer, Float.

Signature: `range(min, max)`: $\alpha \times \alpha \rightarrow bool$

Defaults: obvious.

Listing 7: Examples

```
natural {
  @range(0,%)
  v64 data;
}
positive {
  @range(1,%)
  v64 data;
}
nonNegativeDouble {
  @range(0,%)
  f64 data;
}
```

NonNull

Declares that an indexed field may not be null.

Applies to Field: Any indexed Type.
Signature: `nonnull()`
Defaults: none.

Listing 8: Examples

```
Node {
  @nonnull Node[] edges;
}
```

Unique

Objects stored in a storage pool have to be distinct in their serialized form, i.e. for each pair of objects, there has to be at least one field, with a different value.

NOTE: This can cause difficulties in combination with sub-classing, because the uniqueness property must hold even on the part restricted to the topmost class declared to be unique.

Applies to Declarations of indexed types.
Signature: `unique()`
Defaults: none.

Listing 9: Examples

```
@unique Operator {
  string name;
}
@unique Term {
  Operator operator;
  Term[] arguments;
}
```

Singleton

There is at most one instance of the declaration.

Applies to Declarations.
Signature: `singleton()`
Defaults: none.

Listing 10: Examples

```
@singleton System { ... }
@singleton Data{
  /** Note: if data would not be a singleton itself, it
      is likely to violate the singleton property */
  System foo;
}
```

Tree

The reference graph below created by objects of this type forms a tree. The type of the objects is irrelevant. Strings and fields with notree annotation, are not taken into account.

Applies to Declarations or Field.
Signature: `tree()`
Defaults: none.

notree

Applies to field.
Signature: `notree()`
Defaults: none.

Listing 11: Examples

```
Sloc{...}
@tree
SyntaktikEntity{
  /** not a tree, because several entities, might share
      them */
  @notree Sloc sloc;

  SyntaktikEntity[] children;
}
Routine {
  @notree
  Routine[] callers;
  @tree
  Routine[] dominators;
}

@tree
File {
  File[] children;
```

```
  /** several files could have the same name,
      but strings are implicitly @notree */
  string name;
  string content;
}
```

Note: In case of the File example, there is no way to violate the tree property. Note: It is legal for trees to form forests.

6 Hints

Hints are annotations that start with a single `!` and are followed by a hint name.

Access

Try to use a data structure that provides fast (random) access. E.g. an array list.

Modification

Try to use a data structure that provides fast (random) modification. E.g. an linked list.

Unique

Serialization shall unify objects with exactly the same serialized form. In combination with the @unique restriction, there shall at most be an error reported on deserialization.

Distributed

Use a static map instead of fields to represent fields of definitions. This is usually an optimization if a definition has a lot of fields, but most use cases require only a small subset of them. Because hints do not modify the binary compatibility, some clients are likely to define the fields to be distributed or lazy.

Lazy

Deserialize the fields data only if it is actually used. Lazy implies distributed.

Ignore

The generated code is unable to access the respective field or type of declaration. This will lead to errors, if it is trying nontheless. This option is provided to allow clients to reduce the memory footprint, if needed.

7 On the Choice of Built-In Types

Hier irgendwie erklären, dass es nur typen gibt, die entweder notwendig sind (string) oder überall verfügbar, d.h. man redet im prinzip über den kleinsten gemeinsamen nenner aller prorgrammiersprachen. Der typ bool ist vorhanden, weil die meisten sprachen explizit zwischen integer und bool unterscheiden.

Floats sind vorhanden, weil die Sprache sonst keine akzeptanz finden würde.

Dass es typen wie unsigned oder positive nicht gibt ist schade, kann aber im zuge einer beschreibungssprache die restriktionen bietet einfach nachgerüstet werden.

Es gibt keinen binary typ, weil dieser trivial definiert werden kann:

Listing 12: Binary

```
/** binary as found in Apache Thrift */
binary{ i8[] data }
```

8 On Extensibility and Canonical Field Order

Extensibility is an important property. In this section, we develop a normal form of skill definitions, which will allow a robustness of a file format against modification. We will describe the effect of some changes, which can break decoding or encoding capabilities or break the API but not the file format (further ABI).

8.1 Equality of Field Names

Field names are equal, if their lexical representation is equal after converting all characters to lower case. Type declarations must not contain fields with equal names.

8.2 Canonical Field Order

A declaration is in canonical field order, if all fields are in type order and fields with the same or uncomparable type order are sorted in lexical order.

The type order relation is motivated by properties of compound types. The lexical order is motivated by the observation, that this order does not come with a cost, but provides some additional robustness against changes in definitions.

8.3 A Rule of Thumb

A change of the organization of input files or the order of their definition has no effect.

The addition of new declarations has no effect.

A change regarding comments has no significant effect.

A change in restrictions of any kind may break the API, potentially depending on the target programming language. It will most certainly change the set of legal files.

A change of hints shall have no significant effect, although some applications can stop working after a change of hints, e.g. if they access fields which are annotated with `!ignore`.

Inserting or removing the keyword `const` may break compatibility. To do (14)

Changing the value of a constant will break the ABI.[6]

[6]Which is btw. the very purpose of constants.

Inserting or removing the keyword `auto` will break ABI, but not the API.

Any change of the structure of existing declarations, i.e. changing the modifier, adding or removing fields, etc., will break compatibility as a whole.

9 Serialization

This section is about representing objects as a sequence of bytes. We will call this sequence *stream*, its formal Type will be named S, the current stream will be named s. We will assume that there is an implicit conversion between fixed sized integers[7] and streams. We also make use of a stream concatenation operator $\circ : S \times S \longrightarrow S$.

This section assumes, that all objects about to be serialized are already known. It further assumes, that their types and thus the values of the functions (i.e. baseTypeName, typeName, index, $[\![\_]\!]$) explained below can be easily computed.

9.1 Storage Pools

This section contains the description of the high level file layout and gives meaning to the functions $baseTypeName : \mathcal{U} \longrightarrow S$, $typeName : \mathcal{U} \longrightarrow S$ and $index : \mathcal{U} \longrightarrow S$.

The basic idea behind the serialization format is to store the data grouped by type into storage pools. If objects are referred to from other objects, those references are usually given as integer, which is interpreted as index into the respective storage pool.

Because we will identify types by their name in human readable form, the first pool has to be the string pool. Because it is the first pool and the only pool, which stores objects of a built in type directly, it has a special layout. It starts with its size and is followed by exactly as many zero encoded utf8 strings. In skill inspired represenation, this would look like: [8]

```
v64 size;
utf8[size] stringPool;
```

Now, as we have the first pool, we can explain the *index* function. All objects are stored concise pools, which basically can be seen as arrays of these objects. The index function returns the index of the objects, starting with the index 1[9] for the first object. The NULL pointer is represented by the index 0.

Because we want to allow for full reflection capabilities, but we do not want to force users to pay for the mechanism, if they do not require it, we spend a byte to indicate whether the required data is present or not. The data is stored as the last object in the string pool. In skill this part would be:

```
bool lastStringIsDefinition;
```

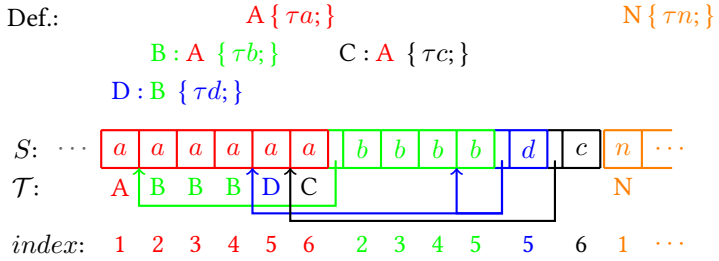
The remaining part is the serialization of the remaining nonempty storage pools. First, these pools are sorted in ascending type order. Then the fields defined of in the type stored in the pool are stored. Each pool keeps a start index, which allows for

[7] As well as between fixed sized floating point numbers, because we define them to be IEEE-754 encoded 32-/64-bit sequences.

[8] Note that the utf8 type is not predefined in skill; it is used to distinguish between the serialization of a string typed field and the very string object.

[9] This is a bit unfortunate, because it may cause some bugs in language binding generators, but using the v64 encoding, it is important to save the 0 for the NULL pointer, as it is encoded with 1 byte instead of 9 bytes as it is the case for -1, which would be the most obvious alternative.

the reconstruction of the complete object. A short example shall illustrate the basic concept. It contains five types A,B,C,D and N. Each has a single field of type τ which is used to simplify the representation. The type Information for the objects in the base type pool can be inferred from the data stored in the pools using the links between the base type pool and the subtype pools (shown as arrows).



Note, that the pool of D comes in front of the pool of C, because it is smaller according to the type order.

This leads to the serialization of a pool as:

```
v64 poolName
v64 superIndex
option(v64 basePoolStartIndex; iff superIndex!=0)
v64 sizeCount
v64 sizeBytes
T[sizeCount] elements
```

A concise description of the file layout could look like this:

```
v64 size
utf8[size] stringPool

bool lastStringIsDefinition

while(next!=EOF){
  v64 poolName
  v64 superIndex
  option(v64 basePoolStartIndex; iff superIndex!=0)
  v64 sizeCount
  v64 sizeBytes
  [[ T[sizeCount] elements ]]
}
```

An interesting observation about this encoding is, that the shortest legal file consists of two bytes, which are both zero. Although the encoding makes use of various invariants it provides some means of error detection, as explained in the appendix.

9.2 Pool Elements

und hier erklären, wie man die felder von elementen serialisiert, wenn man den typ und den index des elements innerhalbe des serialisierten storage pools kennt.

9.3 old garbage! delete

Before being able to understand the serialization of a whole graph of objects into a file, we have to fix a serialization function, which turns an arbitrary object into a stream. Let $[\![\_]\!] : \mathcal{T} \rightarrow S$ be the translation function, which serializes an object of any type into a stream.

This section uses the notion of base and super types. A super type denotes the direct super type of a type. The base type, is the topmost type in a type hierarchy, i.e. any type, which has itself no super type. The super type relation forms a forest.

In the following definitions, let o be the object to be serialized and T be its type.

- The NULL-Pointer is serialized as $[\![\texttt{NULL}]\!] = 0x00$
- For any o of annotation type, the serialization is defined as $[\![o]\!] = [\![baseTypeName(o)]\!] \circ [\![baseTypeIndex(o)]\!]$, i.e. a pointer to the name of a base type pool (=`v64`) and an index into that pool (=`v64`).
- For any o of a user type not treated above, the serialization is defined as $[\![o]\!] = indexOf(o)$, i.e. the index of the object inside the respective storage pool.
- The serialization of a string is its index in the string pool. The serialization of a string inside the string pool is its representing utf8 sequence followed by exactly one terminating 0 character. Interfaces requiring strings without terminating 0 character will add/remove them automatically.[10]
- Booleans are serialized using $[\![\top]\!] = 0xFF$ and $[\![\bot]\!] = 0x00$, i.e. a byte is used, which is all 1s in case of true and all 0s in case of false.
- For any o of fixed size integer type $[\![o]\!] = o$.
- v64, sizes and lengths are serialized using the variable length coding explained in appendix A: $[\![o]\!] = encode(o)$ with encode as in listing 17. This coding favors small natural numbers.
- For any o of floating point type, the serialization is $[\![o]\!] = o$, i.e. the in memory represenation is written into the stream. This assumes floats to be encoded according to IEEE-754.
- For any variably sized array o, $[\![o]\!] = [\![size(o)]\!] \circ [\![o[0]]\!] \circ \cdots \circ [\![o[size(o)-1]]\!]$. Note, that size is treated as if it were a field of type `v64`.
- For any fix sized array o or any array with a size that is the value of another field, $[\![o]\!] = [\![o[0]]\!] \circ \cdots \circ [\![o[size(o)-1]]\!]$.
- Lists and Sets are serialized as if they were variable sized arrays.
- Maps are serialized, as if they were pairs of elements of their argument types. Note that this will cause a nested map type such as `map<T,map<U,V>>` to be serialized using a schema $[\![size(o)]\!] \circ [\![o.t_1]\!] \circ [\![size(o[t_1])]\!] \circ [\![o[t_1].u_1]\!] \circ [\![o[t_1].v_1]\!] \circ [\![o[t_1].u_2]\!] \circ \cdots \circ [\![size(o[t_2])]\!] \circ \cdots \circ [\![o[t_n].v_m]\!]$

blablabla pools, ...

[10] If the stored string happens to be a type name, it is guaranteed that a lowercase conversion of the very string is idempotent.

- Pools are stored in ascending type order.
- Strings which are used as pool names, are stored in the same order, thus the type order of user defined types is equal to the total order defined on the natural numbers during the serialization and deserialization process.

File Layout:

```
v64 size
string[size] stringPool

bool lastStringIsDefinition

while(next!=EOF){
  v64 poolName
  v64 superIndex
  v64 sizeCount
  opt(v64 basePoolStartIndex; iff superIndex!=0)
  v64 sizeBytes
  [[ T[sizeCount] elements ]]
}
```

Iff lastStringIsDefinition is true, the last string of the string pool contains a text that describes the all types usable in the file. This is provided mainly for checking and reflection purpose.

The poolName is an index into the string pool and points to the type name stored in the pool.

The superIndex is an index into the string pool and points to the name of the super type.

The sizeCount contains the number of elements in the pool. This is required in order to move objects of an unknown subtype. It does also simplify the deserialization process.

The basePoolStartIndex is the index of the first element in the base types pool. To do (15)

Note: Pools which do not have entries have to be omitted.

10 Deserialization

Deserialization is mostly straight forward.

The general strategy is:

- the string pool is deserialized and a map from index to strings is created.
- all other objects are deserialized. Any fields referring to indexed types are stored in an adequate data structure.
- fields referring to indexed types are filled with pointers to the actual instances.
- the root object is deserialized

Of course, there can be optimizations in some languages. E.g. in C++, one can simply store create all Objects in one pass and store indices to the respective pools in the pointer typed fields by using an unsafe pointer/integer conversion, which can be corrected in a second pass, where those indices are replaced by the correct pointer values.

10.1 Examples

Nice example in C++:

Listing 13: C++ Examples

```
#include <stdint.h>
#include <string>
[...some other bouilerplate includes...]
struct SLoc {
  uint16_t line;
  uint16_t column;
  std::string* path;
};
struct Block {
  std::string* tag;
  SLoc* begin;
  SLoc* end;
  std::string* image;
};
struct IfBlock : public Block {
  Block thenBlock;
};
struct ITEBlock : public IfBlock {
  Block elseBlock;
};
[...plus some boilerplate code for visitors, iostreams etc. ...]
```

Listing 14: Java Examples

```
class SLoc {
  public short line;
  public short column;
  public String path;
}
class Block {
  final public String tag() { return this.getClass().getName(); }
  public SLoc begin;
  public SLoc end;
  public String image:
}
class IfBlock extends Block {
  public Block thenBlock;
}
class ITEBlock extends IfBlock {
  public Block elseBlock;
}
[...some read and write code, plus some visitors...]
```

Listing 15: LaTeX Examples

```
$(line, column, path) \in SLoc
  \subseteq \mathbb{Z} \times \mathbb{Z} \times string$

$(begin, end, image) \in Block
  \subseteq SLoc \times SLoc \times string$

$(super, thenBlock) \in IfBlock
  \subseteq Block \times Block$

$(super, elseBlock) \in ITEBlock
  \subseteq IfBlock \times Block$
```

Which looks like:

$(line, column, path) \in SLoc \subseteq \mathbb{Z} \times \mathbb{Z} \times string$
$(begin, end, image) \in Block \subseteq SLoc \times SLoc \times string$
$(super, thenBlock) \in IfBlock \subseteq Block \times Block$
$(super, elseBlock) \in ITEBlock \subseteq IfBlock \times Block$

Note: The incentive of the LaTeX-output is to provide a mechanism for users to formalize their file format using mechanisms, that are or can not be available as a specification language. E.g. the sentence "The path of a SLoc points to a valid file on the file system and the line and column form a valid location inside that file." can not be verified in a static manner. This is because the correctness of the property depends not only on the content to be verified, but on the verifying environment as well.

11 Case Study: Skill Encoded XML

Although it is not very clever to use skill for encoding xml files, because one basically looses all benefits from both worlds, we will do so as demonstration for the compression yielded by the skill serialization scheme. Honestly most effects will be obtained from strings being stored in the string pool. Because most of the validation mechanisms directly built into xml are not required in skill and for the sake of simplicity, we will strip xml to its bare payload:

Listing 16: Skill Encoded XML

```
XML {
  string xmlDecl;
  Element element;
}
Element {
  string name;
  map<string, string> attributes;
  /** @note we will supply the empty string, if no
            content is present */
  string content;
  Element[] children;
}
```

To do (16)

To do (17)

To do (18)

12 Future Work

XML output mit XML Schema.

Das neue Serialisierungsschema erlaubt es einen Viewer zu bauen, der Definition+Datei anzeigen kann. (Die future work ist hier der viewer)

Integration der Definition in die Serialisierte Form, damit man die Daten generisch prüfen und anzeigen kann. Hier braucht man noch ein gutes encoding, weil man sonst zu viel platz verbraucht.

Abuse annotations for type-safe unions. The type system does not allow for unrestricted unions or intersection types. The former violate serialization invariants, the latter would either have no instances or be equal to an already existing (super) type.

Fun fact: Garbage Collection for serializable objects comes for free, if objects are always held in storage pools.

To do...

- ☐ 1 (p. 1): blablabla
- ☐ 2 (p. 1): muss noch schöner werden
- ☐ 3 (p. 1): leider wird man nicht um einen glossar rumkommen. ABI, API, super type, base type, ...
- ☐ 4 (p. 1): proof!
- ☐ 5 (p. 1): cite w3c
- ☐ 6 (p. 2): brr
- ☐ 7 (p. 2): ref David Lamb
- ☐ 8 (p. 4): check for updates
- ☐ 9 (p. 4): Appendix with a list of all identifiers which form reserved words in one of the languages above, including our keywords
- ☐ 10 (p. 5): write new section
- ☐ 11 (p. 6): brrr sprache!
- ☐ 12 (p. 6): Ehrlich gesagt sollte man hier irgendwie das LaTeX-backend beschreiben, das würde meistens erklären, was die einzelnen wörter bedeuten.
- ☐ 13 (p. 6): sprache!
- ☐ 14 (p. 13): really?
- ☐ 15 (p. 17): all pools related to the same base type have to be contiguous and adhere to the index order inside the base types storage pool; this should be a side effect of a straight forward implementation
- ☐ 16 (p. 19): compare size of some svg files
- ☐ 17 (p. 19): compare speed of load/store of those svg files
- ☐ 18 (p. 19): some final comments to say, that the comparison is of course not completely fair, and that it is advised against mixing xml and skill in most cases

Part I
Appendix

A Variable Length Coding

Size and Length information is stored as variable length coded 64 bit unsigned integers (aka C's `uint64_t`). The basic idea is to use up to 9 bytes, where any byte starts with a 1 iff there is a consecutive byte. This leaves a payload of 7 bit for the first 8 bytes and 8 bits of payload for the ninth byte. This is very similar to the famous utf8 encoding and is motivated, as it is the case with utf8, by the assumption, that smaller numbers are a lot more likely. It has the nice property, that there are virtually no numerical size limitations. The following small C++ functions will illustrate the algorithm:

Listing 17: Variable Length Encoding

```
uint8_t* encode(uint64_t v){
  // calculate effective size
  int size = 0;
  {
    auto q = v;
    while(q){
      q >>= 7;
      size++;
    }
  }
  if(!size){
    auto rval = new uint8_t[1];
    rval[0]=0;
    return rval;
  }else if(10==size)
    size = 9;

  // split
  auto rval = new uint8_t[size];
  int count=0;
  for(;count<8&&count<size-1;count++){
    rval[count] = v >> (7*count);
    rval[count] |= 0x80;
  }
  rval[count] = v >> (7*count);
  return rval;
}
```

Listing 18: Variable Length Decoding

```
uint64_t decode(uint8_t* p){
  int count = 0;
  uint64_t rval = 0;
  register uint64_t r;
```

```
  for(;count<8 && (*p)&0x80; count++, p++){
    r = p[0];
    rval |= (r&0x7f)<<(7*count);
  }
  r = p[0];
  rval |= (8==count?r:(r&0x7f))<<(7*count);
  return rval;
}
```

B Error Reporting

This section describes some errors regarding ill-formatted files, which must be detected and reported. The order is based on the expected order of checking for the described error. The described errors are expected to be the result of file corruption, format change or bugs in a language binding.

Deserialization

- If `EOF` is encountered unexpectedly, an error must be reported before producing any observable result.
- If an index into a pool is invalid[11], an error must be reported.
- If the deserialization of a storage pool does not consume exactly the `sizeBytes` in its header, an error must be reported. Note: This is a strong indicator for a format change.
- If the serialized type order of storage pools does not match the expected type order, an error must be reported.
- If the serialized type information contains cycles, an error must be reported, which contains at least all type names in the detected cycle and the base type, if one can be determined.
- If a storage pools contains elements which, based on their location in the base pool, should be subtypes of some kind, but have no respective sub type storage pool, an error must be reported with at least, the base type name, the most exact known type name and the adjacent base type names.
- All known constant fields have to be checked before producing any observable result. If some constant value differs from the expected value, an error must be reported, which contains at least the type, the field type and name, the basePoolIndex, the index inside the types pool, the expected value and the actual value.
- If a serialized value violates a restriction or the invariant of a type,[12] an error must be reported as soon as this fact can be observed. It is explicitly not required to check all serialized data for this property.

[11] because it is larger then the last string in the pool

[12] Including sets containing multiple similar elements.

C Reserved Words

This section contains a table of words which must not be used as field names, because they are keywords in some languages. The usage of skill keywords will result in a direct error, whereas the usage of a word listed below will result in a warning, because the identifier will be escaped in the target language binding.

if	**then**	**else**	**begin**	**end**
struct	**class**	**public**	**protected**	**private**
⇒	…			