The Serialization Killer Language

Timm Felden

March 13, 2013

Abstract

fooblaar

1 Einleitung

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 Abgrenzung

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 (1)

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 (2)

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 (3)

ASN.1

Is not powerful enough to fit our purpose.

IDL

To do (4)

Is not powerful enough and seems to be outdated.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 indexed types, which will be a huge benefit.

To do (5)

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>`. The represent 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 documenting system.

```
UNIT :=
  INCLUDE*
  DECLARATION*

INCLUDE :=
  ("include"|"with") <string> ";"?

DECLARATION :=
  DESCRIPTION
  ("tagged"|"indexed"|"annotation")*
  <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
  |GROUNDTYPE)

MAPTYPE :=
  "<" TYPE "," TYPE ">"

SETTYPE :=
  "<" TYPE ">"

LISTTYPE :=
  "<" TYPE ">"

GROUNDTYPE :=
  (<id>|"annotation")
  ("[" (<id>|<int>)? "]")?
```

Note: The Grammar is LL(1).

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

Comment: There will eventually be a mechanism to add functional specifications, such as range restrictions, to fields, which is expected to use `@` and `;`.

2.1 Reserved Words

The language itself has only the reserved words **annotation**, **auto**, **const**, **indexed**, **tagged**, **with**, **map**, **list** and **set**.

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 (6)

2.2 Examples

Nice example

To do (7)

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

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

tagged IfBlock with Block {
  Block thenBlock;
}

tagged ITEBlock with IfBlock {
  Block elseBlock;
}
```

Includes, self references

`example2a`:

```
with "example2b.skill"

indexed A {
  A a;
  B b;
}
```

`example2b`:

```
with "example2a.skill"

B {
  A a;
}
```

Unicode

The usage of non ASCII characters is completely legal, but might cause severe portability issues.

```
/** some arguably legal unicode characters. */
indexed ö {
  ö ß;
  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 indexed

Instances of this type are serialized using a storage pool and indices into that pool. The in-memory representation uses pointers and shared instances.

3.3 tagged

The type has a tag which allows it to be subtyped using single inheritence.

3.4 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. Annotations may be subclassed by other annotations using single inheritence.

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

```
tagged annotation {
  v64 size;
}
```

With the constraint, that size is the size of the serialized form of the annotation in bytes.

3.5 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.

3.6 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.

3.7 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.

3.8 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 (8)

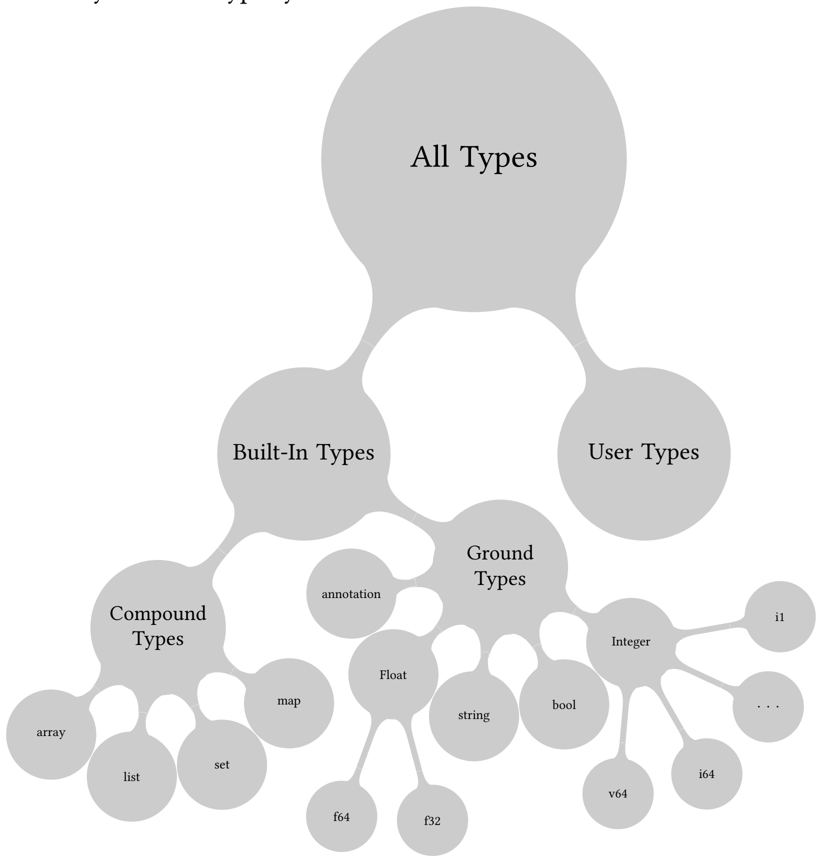
To do (9)

3.9 Comments

Comments provided in the skill file will be emitted into the generated code[1], thus allowing a user to profit from tooltips his IDE is likely to show him, containing this documentation. To do (10)

4 Typsystem

The basic layout of the type system is:



[1]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:

```
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, $\mathcal{B} \in \mathcal{T}$ be the set of built-in types, $\mathcal{G} \in \mathcal{B}$ be the set of ground types, $\mathcal{C} \in \mathcal{B}$ be the set of compound types and $\mathcal{I} \in \mathcal{G}$ be the set of integer types. Let $mod : \mathcal{T} \setminus \mathcal{G} \to 2^{\{indexed, tagged, annotation\}}$ be the set of modifiers for a given type t.

In this context, we talk about an unknown, but fixed set $\mathcal{T}$, which corresponds to the contents of a set of input files. We silently assume, that all types have unique names. Declaring a type with the name of another types is therefore considered an error.

Wellformed type declaretions:

- If $\texttt{s} \in \mathcal{T} \setminus \mathcal{G}$, $mod(\texttt{t}) = mod(\texttt{s})$ and $annotation \in mod(\texttt{t}) \to tagged \notin mod(\texttt{t}) \wedge indexed \notin mod(\texttt{t})$ then $mod(\texttt{t})$`t with s{...}` is a wellformed type declaration and `t` is called a subtype of `s`.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{G}$ and $annotation \notin mod(\texttt{t}) \vee tagged \notin mod(\texttt{t})$ then $mod(\texttt{t})$`t {...}` is a wellformed type declaration.
- The subtype relation is acyclic.

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<`$\texttt{t}_1$`, map<`$\cdots$ `,` $\texttt{t}_n$`>`$\cdots$`> f` is a legal field.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `set<t> f` is a legal field.
- If $\texttt{t} \in \mathcal{T} \setminus \mathcal{C}$ then `list<t> f` is a legal field.
- 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 `s size` is a field in the same type declaration as `f` with $\texttt{s} \in \mathcal{I}$ and `size` is declared before `f` [2] then `t[size] f` is a legal field.

[2]this is required in order to make serialization work; another approach would be to choose a normal form, which would also make declarations robust against reordering

Strings

Strings are conceptually a length encoded 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. If a language, such as C, requires the usage of a terminating 0, all strings have to be terminated with a 0 in their deserialized form. The terminating 0 character is never part of the serialized version of a string.

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 Pointers

NULL Pointers can only be serialized if the respective type is indexed. Otherwise a NULL Pointer is considered an error and will cause the serialization to fail. This is because the resulting file can not be deserialized in all languages, because some languages might choose to represent the respective field e.g. as object. A short example shall illustrate the problem:

```
time{ i64 seconds }
timeStamp{
  ...
  time lastModified;
}
```

In C this will likely look like:

```
struct time { uint64_t seconds; };
struct timeStamp {
  ...
  time lastModified;
}
```

Note that the time object is allocated inside the timeStamp object.

In Java this will likely look like:

```
class time { public long seconds; }
class timeStamp { public time lastModified; }
```

Note that it is legal to store `null` in the lastModified field.

In case of indexed types, the special index 0 is used to serialize NULL. Indexed Types will always be represented by the client languages pointer implementation, i.e. pointer in C, references in Java or access in Ada.

4.1 Examples

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

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

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

```
annotation ...
```

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)`: $\alpha \times \alpha \to \mathit{bool}$

Defaults: obvious.

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.

Examples

```
indexed 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.

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

Examples

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

Singleton

There is at most one instance of the declaration.

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

Examples

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

Examples

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

  SyntaktikEntity[] children;
}
indexed 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.

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.

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:

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

8 Serialization

This section is about representing objects of an arbitary legal type $\mathcal{T}$ 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 and streams. Thus we only need to define a stream concatenation operator $\circ : S \times S \longrightarrow S$

Folgenden Phasen:

- Menge der erreichbaren Objekte bestimmen
- Indices vergeben
- Stringpool schreiben.
- Andere Pools schreiben[3].

The section is to describe the serialization function $[\![\_]\!] : \mathcal{T} \longrightarrow S$, which serializes an object of any type into a stream.

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

[3]Das serialisierte root objekt ist das erste objekt im entsprechenden pool.

- For any o of an unmodified user type, i.e. $mod(T) = \emptyset$, with field declarations $t_i\ f_i$, the serialization is defined as $[\![o]\!] = [\![f_0]\!] \circ \cdots \circ [\![f_n]\!]$. The serialization order equals the declaration order.

- For any o of an indexed user type, the serialization in the respactive pool is done as if the type were not indexed. Outside of the pool, the serializaton of o is defined as $[\![o]\!] = indexOf(o)$, i.e. the index of the object inside the pool.

- The NULL-Pointer, which is legal in case of indexed types is serialized as $[\![\texttt{NULL}]\!] = 0$

- For any o of a tagged type, the serialization is defined as $[\![o]\!] = [\![name(T)]\!] \circ [\![o']\!]$, where o' is o treated as if its type would not be tagged anymore, i.e. serialized tagged types are preceeded by their typenames. If T is indexed as well, the indexed rule preceeds the tagged rule, i.e. the type name is stored in the storage pool, but not in a field.

- For any o of annotation type, the serialization is defined as $[\![o]\!] = sizeOf([\![o']\!]) \circ [\![o']\!]$, i.e. the serialization of the actual payload is proceeded by the size of the payload. Therefore annotatio fields can contain arbitrary content.

- The serialization of a string is its index in the string pool. The serialization of a string inside the string pool is its length, followed by its representing utf8 sequence **without** terminating 0 character. Interfaces requiring strings with terminating 0 character will add/remove them automatically.

- 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 2. 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 $[\![size(o)]\!] \circ [\![o.t_1]\!] \circ [\![size(o[t_1])]\!] \circ [\![o[t_1].u_1]\!] \circ [\![o[t_1].v_1]\!] \circ [\![o[t_1].u_2]\!] \circ \cdots \circ [\![size(o[t_2])]\!] \circ \cdots \circ [\![o[t_n].v_m]\!]$

Let T_i be the set of indexed types with nonzero elements reachable from the serialization root r. Then the serialization of r can be described as $[\![strings]\!]_{Pool} \circ encode(n) \circ [\![T_1]\!]_{Pool} \circ \cdots \circ [\![T_n]\!]_{Pool} \circ [\![r]\!]$

9 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.

9.1 Examples

Nice example in C++:

```
#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. ...]
```

Nice example in Java:

```
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...]
```

Nice example in LaTeX-formulas:

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

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

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

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

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.

10 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 1: 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 (11)

To do (12)

To do (13)

11 Future Work

unique keyword? (alles, was in einem pool landet und die selbe serializierte darstellung hat wird durch das selbe objekt repräsentiert. Das ist derzeit nicht der Fall; eventuell muss man das machen um sicherzustellen, dass deserialisieren direkt gefolgt von serializisieren zu einer bitweise identischen datei führt, andernfalls möchte man das eventuell garnicht, weil man dann auch eine reihenfolge für die erzeugung von pools und das vergeben von indices in der serialisierten form vorgeben müsste)

Serialisierungsnormalform? (Robustheit gegenüber Änderungen in der Spezifikation)

Rangechecks, Invarianten und anderes Gedöns. -> @-statements

Implementation hints für bestimmte typen wie listen, die fast access oder fast insert bieten. -> !-statements

XML output mit XML Schema.

To do...

- ☐ 1 (p. 1): proof!
- ☐ 2 (p. 1): cite w3c
- ☐ 3 (p. 1): brr
- ☐ 4 (p. 2): ref David Lamb
- ☐ 5 (p. 2): proof!
- ☐ 6 (p. 3): Appendix with a list of all identifiers which form reserved words in one of the languages above, including our keywords
- ☐ 7 (p. 4): more comments, nicer, we will use this as running example
- ☐ 8 (p. 6): brrr sprache!
- ☐ 9 (p. 6): Ehrlich gesagt sollte man hier irgendwie das LaTeX-backend beschreiben, das würde meistens erklären, was die einzelnen wörter bedeuten.
- ☐ 10 (p. 6): sprache!
- ☐ 11 (p. 16): compare size of some svg files
- ☐ 12 (p. 16): compare speed of load/store of those svg files
- ☐ 13 (p. 16): 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 justifed, 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 2: 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 3: 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;
}
```