The SKilL Language

Timm Felden

TR 2013/06

Abstract

This paper presents an approach to serializing objects, which is tailored for usability, performance and portability. Unlike other general serialization mechanisms, we provide explicit support for extension points in the serialized data, in order to provide a maximum of upward compatibility and extensibility.

Acknowledgements

Main critics: Erhard Plödereder and Martin Wittiger.
Additional critics: Dominik Bruhn and Dennis Przytarski.

Contents

1 Motivation

Many industrial and scientific projects suffer from platform or language dependent representation of their core data structures. These problems often cause software engineers to stick with outdated tools or even programming languages, thus causing a lot of frustration. This does not only increase the burden of hiring new project members, but can ultimately cause a project to die unnecessarily.

The approach presented in this paper provides means of platform and language independent specification of serializable data structures and therefore a safe way to let old tools of a tool suite talk to the new ones, without even the need of recompiling the old ones. We set out to design a new and easy to use way of making core data structures of a tool platform language independent, because we believe that the best language a programmer can use to write a new tool, is the language that he likes the most. We also had the strict requirement to provide a solution that can describe an intermediate representation with stable parts that can be used for decades and unstable parts that may change on a daily basis.

In order to achieve this goal, we introduce two new concepts:

The first one is an easy to use specification language for data structures providing simple data types like integers and strings, container types like sets and maps, type safe pointers, extension points and single inheritance. The specification language is modular in order to make large specifications more readable.

The second one is a formalized mapping of specified types to a bitwise representation of stored objects. The mapping is very compact and therefore scalable, easy to understand and therefore easy to bind to a new language. It does encode the type system and can therefore provide a maximum of upward and downward compatibility, while maintaining type safety at the same time. It allows for a maximum of safety when it comes to manipulating data unknown to the generated interface, while maintaining high decoding and encoding speeds[1].

An improvement over the Extensible Markup Language (XML), our main competitor, is that the reflective usage of stored data is expected to be quite rare, because the binding generator is able to generate an interface that ensures type safety of modifications and provides a nice integration into the target language. This leads to a situation, where it is possible to use files containing data of arbitrary types. If the data stored in the file is not used by a client, he does not have to pay for it with execution time or memory. Furthermore a client does not have to know the whole intermediate representation of a tool suite, but only the parts he is going to use in order to achieve his goals. The expected file sizes range from a megabyte to several gigabytes, while having virtually no relevant numerical limits in the file format[2]. Please note, that the SKilL file format is a lot more compact then equivalent XML files would be. It is expected, that files contain objects of hundreds of types with thousands of instances each. If a type in such a file would contain three pointers on average, the file size would still be around a mega byte, which is due to a very compact representation of stored data. This will also lead to high load and store performance, because the raw disk speed is expected to be the limiting factor.

[1] The serialization and deserialization operations are linear in the size of the input/output file.

[2] There are practical limits, such as Java having array lengths limited to 2^{31} or current file systems having a maximum file size limit that is roughly equivalent to the size of a file completely occupied by objects with a single field of a single byte. There will also be problems with raw I/O-Performance for very large files and an implementation of a binding generator, which can handle files not storable in the main memory is a tricky thing to do.

1.1 Scientific Contributions

This section is a very concise representation of contributions that in part have already been mentioned above and in parts will be mentioned much later.

The suggested serialization format and serialization language offer all of the following features in a single product:

- a small footprint and therefore high decoding speeds
- a fully reflective type encoding
- type safe storage of references both to known and unknown types[3]
- a rich type system providing amongst others references, containers, single inheritance and extension points
- the specification language is modular[4] and easy to use
- no tool using a common intermediate representation has to know the complete specification. It is even possible to strip away or add individual fields of commonly used types.
- the coding is platform and language independent
- the coding offers a maximum of downward **and** upward compatibility
- a programmer is communicating through a generated interface, which allows programmers, knowing nothing about the Serialization Killer Language (SKilL), to interact with it, and ensures type safety. It also allows programmers to write tools in the language they know best[5].
- stored data, that is never needed by a tool, will never be touched.
- new objects can be added to a file by appending data to the existing file.

Any of the arguments above have already been made in various contexts (e.g. [TDB+06] §13.13, [LA13], [xml06], [Lam87]), but there is, to the best of our knowledge, no solution bringing all these demands together into a single product that does the job automatically.

1.2 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.

XML

XML is a file format (defined in [xml06]). The main differences are:

+ XML can be manipulated with a text editor[6].
+ It is easier to write a libXML for a new language than to write a SKilL back-end[7].

[3] I.e. regular references and annotations.

[4] I.e. it can be distributed over many files.

[5] This is a problem especially in the scientific community, where many researchers work on similar problems but with completely different tools.

[6] Whereas SKilL files are binary and require a special editor, which will be provided by us eventually.

[7] This is only a relevant point if no bindings exist for the language you want to use.

- XML is not an efficient encoding in terms of (disk-)space usage. This can be overcome by the Efficient XML Interchange Format (EXI) (see [SK11]).
- XML is not type safe. This can be overcome partially by the XML Schema Definition Language (XSD).
- XML does not provide references to other objects out of the box.
- XML stores basically a tree, whereas a SKilL file contains an arbitrary amount of graphs of objects.
- XML is usually accessed through a libXML, whereas SKilL provides an API for each file format, thus a SKilL user does not require any SKilL skills, i.e. no knowledge about SKilL types or the representation of serialized objects is required in general. To be honest, there are some language bindings, mainly for Java, which offer this benefit for XML as well.

XML Schema definitions

SKilLs description language itself is more or less equivalent to XML schema definition languages such as XSD (as described in [GSMT+08, PGM+08]). The most significant difference is caused by the fact that XML operates on trees and SKilL operates on arbitrary graphs.

The type systems offered by SKilL and XSD are quite different, thus it might be worth a look which one better fits ones needs.

JAXP and xmlbeansxx

For Java and C++, there are code generators that turn an XML schema file into code, which is able to deal with an XML in a similar way as proposed by this work. In case of Java the mechanism is even part of the standard library. The downside is that, to the best of 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.

ASN.1

Is not powerful enough to fit our purpose.

IDL

The published format is stated to be ASCII ([Lam87] §2.4), which will cause similar efficiency problems as raw XML does, if large amounts of data are stored.

Apache Thrift & Protocol Buffers

Thrift states that there is no sub-typing (see [Apa13][8]). Protocol Buffers (see [Goo13]) seem not to support sub-typing either. Both seem to be a pragmatic approach to generalization of efficient network protocols. The type system of Protocol Buffers is also a rather pragmatic solution offering types such as unsigned 32 bit integers,

[8] In section "structs", first sentence: "Thrift structs define a common object – they are essentially equivalent to classes in OOP languages, but without inheritance.", as of 29.Aug.2013

which can not be represented in an efficient and safe way by e.g. Java. Both do not have storage pools, which are the foundation of our serialization approach and an absolute requirement for some of our optimizations, such as hints (see section 5.2).

Protocol Buffers provide a variable length integer type, namely `int64`, which seems to be binary compatible[9] with the variable length integer type used in SKilL (see section A).

Java Bytecode, LLVM/IR and others

Although Java Bytecode (see [LYBB13]) and the LLVM Intermediate Representation (see [LA13]) are hand crafted formats, they served as a guiding example in many ways.

Language Specific Serialization

Language specific serialization is language specific and can therefore not be used to interface between subsystems written in different programming languages, without a lot of effort. Our aim is clearly a language independent and easy to use serialization format.

[9] The Protocol Buffer implementation seems not to optimize away the ninth flag, thus it might use an additional byte for very large numbers.

2 Syntax

This section discusses the syntax of the description language in brevity. The semantics is discussed in section 3, the file format is discussed in section 6.

We use the tokens `<id>`, `<string>`, `<int>`, `<float>` and `<comment>`. They equal C-style identifiers, strings, integer literals, float literals and comments respectively. Identifiers, strings and comments are explicitly enriched by printable Unicode characters above `\u007f`, although this feature should be used with care. We use a comment token, because we want to emit the comments in the generated code, in order to integrate nicely into the target language's documentation system.

2.1 Reserved Words

The language itself has only the reserved words **annotation**, **auto**, **const**, **include**, **with**, **bool**, **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.

2.2 The Grammar

The grammar of a SKilL definition file is defined as:

```
UNIT :=
  INCLUDE*
  DECLARATION*

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

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

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

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

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

R_ARG := (<float>|<int>|<string>)

HINT := "!" <id>
```

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

DATA :=
  "auto"? TYPE <id>

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

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

SETTYPE :=
  "<" GROUNDTYPE ">"

LISTTYPE :=
  "<" GROUNDTYPE ">"

ARRAYTYPE :=
  GROUNDTYPE
  ("[" (<int>)? "]")?

GROUNDTYPE :=
  (<id>|"annotation")
```

Note: The Grammar is LL(1).[10]

2.3 Examples

Listing 1: Running Example

```
/** A location in a file pointing to a character in that
    file. Assumes ordinary text files. */
Location {
  /** the line of the character starting from 0 */
  i16 line;
  /** the column of the character starting from 0 */
  i16 column;
  /** the file containing the location */
  File path;
}

/** A range of characters in a file. */
Range {
```

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

```
  /** first character; inclusive */
  Location begin;
  /** last character; exclusive */
  Loctaion end;
}

/** A hierarchy of file/directory names. */
File {
  /** Name of this file/directory. */
  string name;
  /** NULL iff root directory. */
  @nullable File directory;
}
```

Includes, self references

Listing 2: Example 2a

```
with "example2b.skill"

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

Listing 3: Example 2b

```
with "example2a.skill"

B {
  A a;
}
```

which is equivalent to the file:

Listing 4: Example 2

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

B {
  A a;
}
```

Subtypes

Types can be extended using subtyping:

Listing 5: Subtyping Example

```
with "runningExample.skill"

/** a message is just a string */
Message {
  string message;
}

/** located messages contain a location as well */
LocatedMessage extends Message {
  Location location;
}
```

Containers

Container types can be used to store more elaborate data structures, then just plain values or references. In the current version, there is support for sets, maps, lists and arrays:

Listing 6: Containers

```
/** E.g. a user in a social network. */
User {
  string name;

  /** friends of this user */
  List<User> friends;

  /** default values of permissions can be overriden on
      a per-user basis. The value is stored explicitly to
      ensure that the override survives changes of the
      permissions default value. */
  Map<User, Permission, Bool> permissionOverrides;
}

Permission {
  string name;
  bool default;
}
```

Unicode

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

Listing 7: Unicode Support

```
Ä {
  Ä ∀;
  Ä €;
}
```

3 Semantics

This section will describe the meaning of specifications by explaining the effect of declarations.

3.1 A Specification File

```
UNIT :=
  INCLUDE*
  DECLARATION*
```

Specification files are fed into binding generators, which generate code that provides means to deal with instances of the declared types.

SKilL specifications consist of a set of declarations which themselves consist of fields. A declaration is roughly equivalent to a type declaration in an object oriented programming language. The main difference is, that declarations are pure data, because we do not offer a real execution model. The only operations from the perspective of SKilL are loading and storing of data.

A declaration will instruct the language binding generator to create a type which has the declarations name that consist of the fields specified in the body of the declaration. Fields behave just like fields in object oriented programming. Both form the structure of serialized data and are identified using human readable names.

Types are discussed in section 4. Restrictions and Hints are discussed in section 5.1 and 5.2.

3.2 Includes

```
INCLUDE :=
  ("include"|"with") <string>+
```

Includes are used to structure a specification into smaller models, e.g. by moving data that is only used by some tools to its own file.

The files referenced by the `include` statement are processed as well. The declarations of all files transitively reachable over `with` statements are collected, before any declaration in any file is evaluated. The order of inclusion is irrelevant. The same file may even be included multiple times by the same `include` statement.

Therefore evaluation of declarations happens as if all declarations were defined in a single file.

3.3 Type Declarations

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

The SKilL specification language is all about type declarations. A type decalartion consists at least of a name and a body, containing field decalartions.

Descriptions

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

Type (and field) declarations can be enriched with descriptions.

Comments provided in the SKilL specification will be emitted into the generated code[11] to serve as a natural language description of the respective entity. This approach enables users to get tool-tips in an IDE showing him this documentation. Future revisions are expected to expand the notion of a comment by introducing tags as used by most documentation generation systems, such as doxygen [vH13] or javadoc [jav13].

Restrictions and hints will be explained below.

Subtypes

A subtype of a user type can be declared by appending the keyword `with` (or `:` or `extends`) and the supertypes name to a declaration. A subtype behaves like a subtype in object oriented programming. Subtypes inherit all fields of their super types. Only user defined types can be sub-typed. In order to be well-formed, the subtype relation must remain acyclic and must not contain unknown types.

3.4 Field Declarations

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

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

DATA :=
  "auto"? TYPE <id>
```

Types are sets of named fields. Fields are either constants or real data.

A usual field declaration consists of a type and a field name. In this case, the field declaration behaves like a field declaration in any object oriented programming language, except that the field data will be serialized.

Constants

A `const` field can be used in order to create guards or version numbers. The deserialization mechanism has to report an error if a constant field has an unexpected value. This mechanism is intended to be used basically for preventing from reading arbitrary files and interpreting them as the expected input. The mechanism can be used defensively, because storing constant fields creates a constant overhead and is not influenced by the number of instances of a type.

Only integer types can be used as constants.

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

Transient Fields

Transient fields, i.e. fields which are used for computation only, can be declared by adding the keyword `auto` in front of the type name. The language binding will create a field with the given type, but the content is transparent to the serialization mechanism. This mechanism can be used to add fields to a data structure, which simplify algorithms computing the interesting data, if these helper fields are not of interest after computation. This is especially useful in combination with the possibility to add or drop some fields while generating the binding for a specific tool. The mechanism can also be used for a field with content that can likely be computed very fast.

The keyword `auto` is used, because the content of the field is computed automatically. Besides the name, it has nothing todo with the **auto** type declaration of C++.

3.5 Field Types

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

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

SETTYPE :=
  "<" GROUNDTYPE ">"

LISTTYPE :=
  "<" GROUNDTYPE ">"

ARRAYTYPE :=
  GROUNDTYPE
  ("[" (<int>)? "]")?

GROUNDTYPE :=
  (<id>|"annotation")
```

Basic types are just an identifier with the type name. For compatibility reasons[12], type names are case insensitive.

Types are explained in-depth in section 4.

Container 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 encoded arrays. Their main purpose is to increase the usability of the generated Application Programming Interface (API).

[12] Some programming languages, e.g. Ada, do distinguish types by casing of identifiers. Using types which differ only in case in such languages would be very nasty, because type name would have to be escaped in some way.

These containers showed to increase the usability and understandability of the resulting code and file format.

Annotations

The `annotation` type is basically a typed pointer to an arbitrary user type. Its main purpose is to provide extension points in the form of references to objects, who's type could not be known at the time of the specification of the annotation field.

3.6 Type Annotations

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

R_ARG := (<float>|<int>|<string>)

HINT := "!" <id>
```

SKilL offers two kinds of type annotations: restrictions and hints. Restrictions can be used to restrict a type, e.g. by reducing the range of possible values of an integer fields to those above 23. Hints can be used to optimize the generated language binding.

Both are explained in-depth in section 5.1 and 5.2.

3.7 Name Resolution

Because SKilL is designed to be downward and upward compatible and offers subtyping, it is possible that a future revision of a file format specification will add a field with a name that already exists in a subtype. In general, it is assumed that the maintainer of the super class does not know about all subclasses. Thus, it is desirable to have a mechanism which ensures client code to work correct after such a change.

Therefore, identical field names are legal in SKilL and refer to different fields, as long as they belong to different type declarations.

A generated language binding has to provide means of accessing a field shadowed by a field of the same name in a subtype. In most languages there are built-in mechanisms for this task. Language binding generators shall take care that they do not override field access methods in a way that will actually make the field of the super type inaccessible.

4 The Type System

The description language and the file format are both intended to be *type safe*. The notion of type safety is usually connected to a state transition system. In our context, the only observable state transitions are from the on-disk representation to the in-memory representation and vice versa. Thus, with *type safe* we want to state that deserialization and serialization of data will not change the type of the data. It is further guaranteed that deserialized references will point to objects of the static type of the reference. Further, if one were to deserialize an object of an incompatible type, an error will be raised.

These properties require some form of platform independent type system[13], which is described briefly in this section. The general layout of the type system is visualized in Fig. 1.

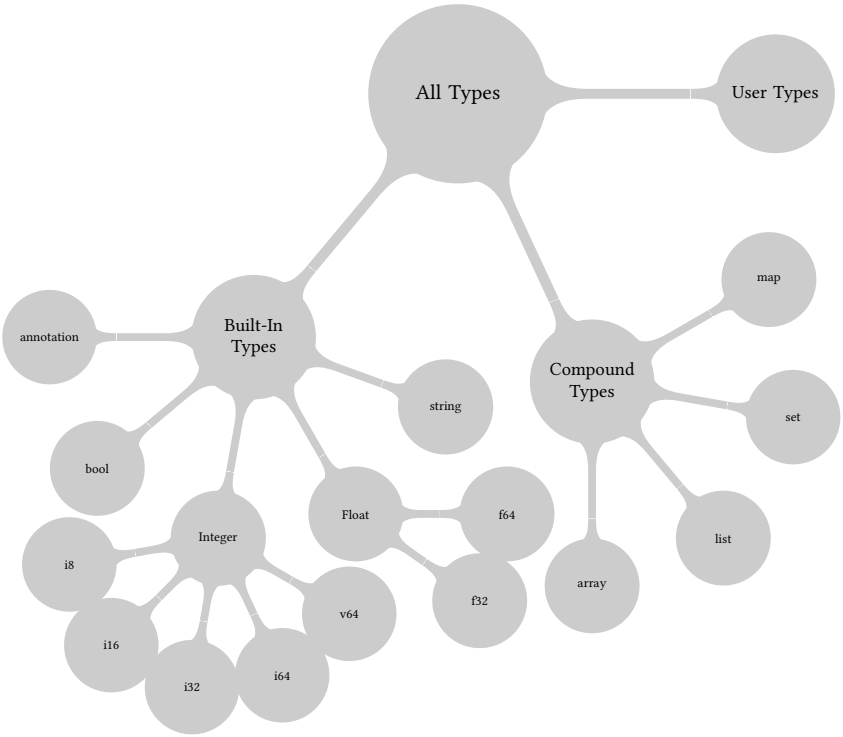

Figure 1: Layout of the Type System

Common Abbreviations

We will use some common abbreviations for sets of types in the rest of the manual:

Let ...

... $\mathcal{T}$ be the set of all types.

... $\mathcal{U}$ be the set of all user types.

... $\mathcal{I}$ be the set of all integer types, i.e. $\{\texttt{i8}, \texttt{i16}, \texttt{i32}, \texttt{i64}, \texttt{v64}\}$.

... $\mathcal{B}$ be the set of all built-in types.

4.1 Built-In Types

The type system provides built-in types, which are the building blocks of type declarations.

[13] In contrast to e.g. C, objects of certain type have a known length and endianness.

Integers

Integers come in two flavors, fixed length and variable length. There is currently only a single variable length integer type, namely Variable length 64-bit signed integer (v64). The variable length integer type can store small values in a single byte (see appendix A for details). Large values ($\geq 2^{55}$) and negative values require one additional byte, i.e. nine bytes.

Booleans

Booleans can store the values true($\top$) and false($\bot$). Unlike most C programmers, we do not perceive booleans as integers.

Annotations

Annotations are designed to be the main extension points in a file format. Annotations are basically typed pointers to arbitrary types. This is achieved by adding the type of the pointer to a regular reference. A language binding is expected to provide something like an annotation proxy, which is used to represent annotation objects. If an application tries to get the object behind the proxy for an object of an unknown type, this will usually result in an error or exception[14]. Therefore language bindings shall provide means of inspecting whether or not the type of the object behind an annotation is known.

Strings

Strings are conceptually a variable length 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. The serialization is described in section 6. If a language demands NULL-termination in strings, the language binding will ensure this property.

Strings should not contain NULL characters, because this may cause problems with languages such as C.

The API shall behave as if strings were defined with the hint `pure` and `unique` (see section 5.2).

NULL Pointer

Fields of type strings, annotations or a user type can store a NULL pointer. Annotations are by default nullable, other references are by default non-NULL. Ordinary references can be made nullable, using the nullable restriction as explained in section 5.1.

Floating Point Numbers

For convenience, it is possible to store 32 bit and 64 bit IEEE-754 floating point numbers. For a description, see [iee08] especially §3.4.

[14]The reflection mechanism allows for other solutions, but raising an exception is the most obvious reaction.

4.2 Compound Types

The language offers several compound types. Sets, Lists and variable length Arrays[15] are basically views onto the same kind of serialized data, i.e. they are a length encoded sequence of instances 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 object twice. All compound types 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 have two or more type arguments.

4.3 User Types

User types can be interpreted as sets of type-name-pairs. Built-in types can be wrapped in order to give them special semantics. For example an appointment can be represented as:

Listing 8: Example User Type

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

  /** A topic, such as "team meeting". */
  string topic;

  /** the name of the room. */
  string room;
}
```

4.3.1 Legal Types

The given grammar of SKilL already ensures that intuitive usage of the language will result in legal type declarations. The remaining aspects of illegal type declarations boil down to ill-formed usage of type and field names and can be summarized as:

- Field names inside a type declaration must be unique. Field names of super types are not relevant.
- The subtype relation is a partial order[16] and does not contain unknown types.
- Any base type has to be known, i.e. it is a user type defined in any document transitively reachable over include commands.
- The type names must be unique in the context of all[17] types.

[15] I.e. arrays, which do not have constant size. Constant length arrays exist as well.

[16] In fact it forms a forest.

[17] From the perspective of a client, i.e. all types that were declared at the time of the generation of its interfaces plus all types that are ever observed in the form of *unknown* types encoded in SKilL files.

4.3.2 Equivalence of Type and Field Names

Type and field names, i.e. any strings referenced from reflection data stored in a SKilL file, shall be treated as equivalent, if they were equal after converting all characters to lower case.

We recommend using CamelCase in SKilL definition files in order to provide a hint to the language binding generator on how to separate parts of identifier names. For example an Ada generator may decide to add underscores to the names in the generated interface leading to a more natural feeling for Ada programmers.

4.4 Examples

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

Listing 9: Legal Super Types

```
EncodedString extends string {
  string encoding;
}
```

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

Listing 10: Legal Type Names

```
/**
 The german word for "car".
*/
Auto {
  ...
}
```

Error: “Auto” is a reserved word and can not be used as type name.

Listing 11: Usage of Unknown Types

```
A {
  map<A,B> f
}
```

Error: The field “A.f” refers to a missing type B. Did you forget to include ”B.skill”?

5 Type Annotations

SKilL provides two concepts of extending the basic type system used in the serialization process.

The first concept is called restriction and is inspired by the concept of (type or class) invariants. This concept can be used to restrict the set of legal objects storable in a field or the set of legal instances of a class.

The second concept is called hint. Hints are used to improve the generated language binding and do not influence types per se.

5.1 Restrictions

Restrictions can be added to declarations and fields. They can occur in any number at the same places as comments. Restrictions start with an @ followed by a name and optional arguments. If multiple restrictions are used, the conjunction of them forms the invariant, i.e. all of them have to apply. The set of legal restrictions is explained below.

If Restrictions are used on compound types, they expand to the components of the respective compound type. Restrictions can not be combined with map-typed fields.

Restrictions are checked by the generated binding at least before serialization and after deserialization. If checking restrictions involves fields, which are not present in a deserialized file, the respective restriction is ruled to hold. This is important to guarantee compatibility with older or newer versions of a file format used in a tool chain. This behavior puts the burden of fulfilling restrictions to the creator of data.

Restrictions are serialized to ensure the asserted properties.

Range

Range restrictions are used to restrict ranges of integers and floats. They can restrict the minimum or maximum value or both. Restrictions can be inclusive or exclusive – the default is inclusive.

Note that this will change the default value of the argument field to *min* iff $0 \notin [min, max]$.

Applies to fields:	Integer, Float.
Signatures:	`range(min, max, boundaries)`: $\alpha \times \alpha \times string^?$
	`min(min, boundaries)`: $\alpha \times string^?$
	`max(max, boundaries)`: $\alpha \times string^?$

Listing 12: Examples

```
RangeRestricted {
  @min(0)
  v64 natural;

  @min(1)
  v64 positive;
  /* or */
  @min(0,"exculsive")
  v64 positiveAlt;

  @range(0.0, 360.0, "inclusive,exclusive")
  f32 angle;
}
```

Nullable

Declares that the argument field can be `NULL`.

Note that fields, which have not been initialized, always contain `NULL` values.

Applies to Field: **string** or any user types.

Listing 13: Examples

```
Node {
  @nullable 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. Because the combination of unique with sub-typing has counter-intuitive properties, we decided that using the unique restriction together with a type that has sub- or super-types is considered an error, which has to be detected at runtime.

Furthermore, be warned that adding or removing fields from unique types can cause serious compatibility issues.

Applies to Declarations.

Listing 14: Examples

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

Singleton

There is at most one instance of the declaration.

Applies to Declarations.

Listing 15: Examples

```
/** Stores properties of the target system. */
@singleton System {
  string name;
  ...
}
```

Monotone

Instances of this type can not be deleted. This restriction is basically a type system way of ensuring properties required by fast append operations. It should be used, where ever it is not necessary to delete instances of a type. The monotone restriction can only be added to base types and expands to all sub types of the base type[18].

Monotone types can be treated by a language binding in a very optimized way.

[18] This behavior is caused by the fact, that for A <: B, all B instances are As as well. If B would not be monotone, deleting a B would directly delete an A. If A would not be monotone, deleting an A might delete a B.

The monotone restrictions implies the monotone hint (see section 5.2).
Applies to Base Type Declarations.

Listing 16: Examples

```
/** just kidding */
@Monotone SocialNetworkPost {
  string message;
}

/** this is monotone as well */
PrivatePost extends SocialNetworkPost {
  string recipient;
}
```

Constant Length Pointer

The argument pointer is serialized using i64 instead of v64. This can be used on regular references and annotations. The restriction makes only sense if the generated binding supports lazy reading of partial storage pools and if the files that have to be dealt with, would not fit into the main memory of the target machine. Using this restriction will most certainly increase the file size and does not restrict any pointer targets.

This restriction is serializable and, thus, does not affect compatibility in any way.
Applies to fields.

Listing 17: Examples

```
/* stored points to information may exceed the available
    main memory, thus we have to access it directly from
    disk */
PointsToTargets {
  @constantLengthPointer
  Context context;
  @constantLengthPointer
  HeapObject object;
  @constantLengthPointer
  PointsToSet targets;
}
```

5.2 Hints

Hints are annotations that start with a single `!` and are followed by a hint name. Hints are used to optimize the behavior of the generated language binding. They do not impact the semantics of type declarations or stored data. Therefore they will not be serialized.

Language bindings shall provide the same public interface as if no hints were used.

Language binding generators shall provide an option that adds a hint to all applicable declarations.

Access

Can be used on: Container-typed field declarations.

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

Modification

Can be used on: Container-typed field declarations.

Try to use a data structure that provides fast (random) modification, e.g. a linked list[19].

Unique

Can be used on: Type declarations.

Serialization shall unify objects with exactly the same serialized form. In combination with the @unique restriction, no error shall be reported on serialization.

Note that this will increase the runtime complexity of the serialization phase from $O(n)$ to $O(nlog(n))$.

Pure

Can be used on: Type declarations.

Deserialized objects of the annotated type shall not be modifiable. The generated interface will provide a copy operation, which will create a modifiable copy of the object. An example of this behavior is the string pool. An equivalent, in terms of API observable behavior, would look as follows:

Listing 18: User Strings

```
!pure
!unique
UserString {
  i8[] utf8Chars;
}
```

Distributed

Can be used on: Field declarations.

A static map will be used instead of fields to represent fields of definitions in memory. 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 even lazy.

Note that this will increase both the memory footprint[20] and the access time for the given field and will only be a benefit for memory-cache locality reasons, because single objects can be significantly smaller[21]. The internal representation will change

[19] Which has faster insert/delete operations than an array list.

[20] Because additional data structures, such as trees, are required in order to provide acceptable access times.

[21] Note this does not apply to operations on distributed fields, but to operations on objects having distributed fields.

from `o.f`, i.e. a regular field, to `pool.f[o]`, i.e. a map in the storage pool which holds the field data for each instance.

Note that the presence of distributed, lazy or ignored fields will require objects to carry a pointer to their storage pool, which may eliminate the cache savings completely.

Lazy

Can be used on: Field declarations.

Deserialize the respective field only if it is actually used. Lazy implies distributed. This hint should be used, if fields are usually not accessed, e.g. in the context of error reporting.

Monotone

Can be used on: Base Type declarations.

Instances of the argument type will not be deleted. New instances can be added to the state. This allows an optimized treatment of data, because assumptions about object IDs can be made.

Accessing lazy fields of instances of monotone types is about as efficient as accessing ordinary fields. Note that usage of v64 and references requires deserialization of all fields in a type block. Thus, it might be worth considering usage of constant length integers or the constantLengthPointer restriction, if this case is encountered on a regular basis.

In contrast to the monotone restriction, the hinted monotone property applies only to the generated binding.

ReadOnly

Can be used on: Base Type declarations.

The generated code is unable to modify instances of the respective type. This hint can be used to provide a consistent API while preventing from logical errors, such as modifying data from a previous stage of computation. The ReadOnly hint expands to all subtypes of the base type, because this is the only safe way[22].

ReadOnly implies Monotone.

Ignore

Can be used on: Type and Field declarations.

The generated code is unable to access the respective field or any field of the type of the target declaration. A language binding shall raise an error (or exception), if the field is accessed nonetheless. This hint can be used to provide a consistent API for a combined file format, but restrict usage of certain fields, which should be transparent to the current stage of computation. This is actually more restrictive than deleting fields from declarations, because the generated reflective API will respect this hint.

[22]This behavior is caused by the fact, that for A <: B, all B instances are As as well. If A would not be ReadOnly, modifying an A would directly modify a B. If B would not be ReadOnly, modifying a B might modifying an A.

6 Serialization

This section is about representing objects as a sequence of bytes. We will call this sequence *stream*, its formal type will be called S, the current stream will be called s. We will assume that there is an implicit conversion between fixed sized integers[23] and streams. We also make use of a stream concatenation operator $\circ : S \times S \rightarrow S$.

We will use upper case letters for types (e.g. A) and lower case for instances of the respective type (e.g. a). The type $\mathcal{T}$ denotes the set of types and τ a type. We will use τ_i for arbitrary type names and f_i for arbitrary fields, i.e. $\tau_i.f_j$ is the j'th field of the i'th type.

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.

The serialization function $[\![\_]\!]_\tau : \tau \times \mathcal{T} \rightarrow S$ maps an object _ of a type τ to a stream. Usually the type of the object can be inferred from context, thus we can simply write $[\![\_]\!]$. During the process of (de-)serialization, the type of an object can always be inferred from context.

6.1 Steps of the Serialization Process

In general it is assumed that the serialization process is split into the following steps:

1. All objects to be serialized are collected. This is usually done using the transitive closure of an initial set.
2. Objects are organized into their storage pools, i.e. the index function is calculated.
   - If the state was created by deserialization and indices have changed[24], fields using these indices have to be updated.
   - All known restrictions have to be checked.
3. The output stream is created as described below.

6.2 General File Layout

The file layout is optimized for fast appending of new objects. It is further optimized for lazy and partial loading of existing objects. It does also support type-safe and consistent treatment of unknown data structures. In order to achieve these goals, we have to store the type system used by the file together with the stored data. The type system itself is using strings for its representation. We want to be able to diagnose file corruption as early as possible, therefore most information stored in a file relies only on information that has already been processed. The only exception to this rule are field types referring to user types which are declared later in the same block.

Therefore the file is structured as an altering sequence of string blocks and type blocks, starting with a string block. The layout of string blocks (S) and type blocks (T) is visualized in figure 2, details will be explained in the following sub-sections.

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

[24] Indices can change by inserting objects before an existing object or by deleting objects. This is caused by the base pool index concept explained in section 6.3

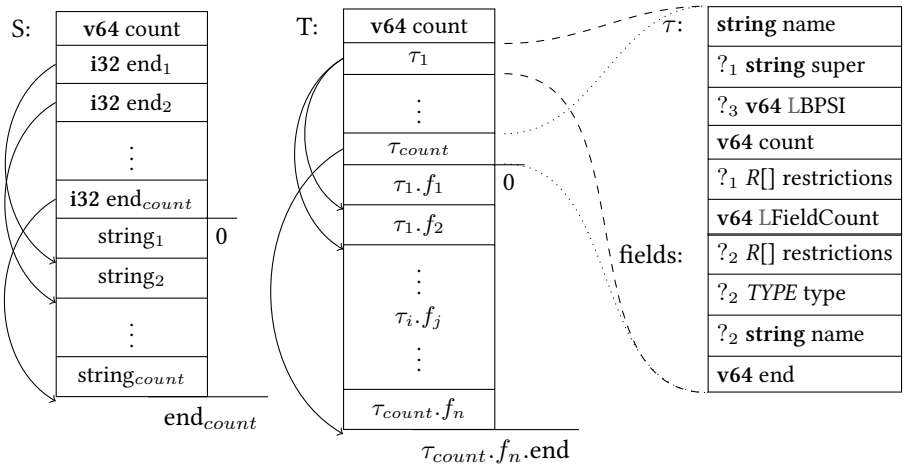

Figure 2: Visualization of the layout of string(S) and type(T) blocks. Type descriptors (τ) and their fields can drop fields in some contexts ($?_i$). Fields prefixed with a gray L contain information only relevant for the local block. The data chunks of blocks start at the respective **0** and reach until the respective **end**. Arrows indicate end-offsets – type blocks have one per field declaration.

6.2.1 Layout of a String Block

A string block starts with a v64 *count*, which stores the number of strings stored in the block. It is followed by *count* many i32 values, which store the offset of the end of the respective string. The stored offsets split the data following after the last offset into utf-8 encoded strings.

The individual strings can be decoded using their index and the previous index (or 0, if there is no previous index). The strings stored in the string block are used by the following type blocks. For efficiency[25] reasons, strings used as type or field names shall be stored in lower case only. If a string containing user data equals a type/field name with different casing, a copy has to be made.

6.2.2 Layout of a Type Block

Instances of types are organized into storage pools (see section 6.3), which are stored in type blocks.

A type block starts with a v64 *count*, which stores the number of *instantiated* types stored in the block. The *count* is followed by the respective amount of type declarations.

Type declarations themselves contain field declarations, which contain end-offsets into a data chunk located at the end of the type block, i.e. the field data is stored between the end-offset (or the start of the data chunk) of the previous field and the end-offset of this field. The Local Base Pool Start Index (LBPSI) field (marked with $?_3$ in Fig. 2) is only present if there is a super type. The LBPSI is used to indicate which fields in a supertype declaration belong to instances of the current type. The local version of the Base Pool Start Index (BPSI) works in the same way as the BPSI in storage pools (see arrows in Fig. 5). The LBPSI field is only present if there is a

[25]The lower case conversion during deserialization is optimized away.

supertype and the type has not yet been defined in a previous type block, because the data is not needed otherwise.

A type is *instantiated*, if the block adds new instances, fields or both. For example, the first block may add a type *node*, with an *ID* field. The second block, which is the result of a graph coloring tool, adds a *color* field to the *node*. We will come back to this example in section 6.2.3.

Effects of Lazy Evaluation

In this context, lazy evaluation means "the ability to skip data".

Inside of string blocks, most indices and string payload can be skipped. The offset type is `i32`, which allows for random access deserialization of individual strings, if the ID is known. This feature is very valuable, if there are many user data strings and few type names. The decision has also the consequence, that strings can only have 2^{32} bytes of data. We consider this limit irrelevant, because a user can still use a byte array to represents string-like objects exceeding this limit.

Inside of type blocks, all type information has to be processed in any case. The field data can be skipped completely.

Thus the laziest processing of a SKilL file will read the count fields of blocks, the last index of string blocks and all type information, including strings storing type and field names. Even in case of large files with many types, the amount of processed data is expected to be below a megabyte.

Effects of Appending

The desire to append new data of an arbitrary kind to an existing file, without having to rewrite existing data, affects mostly the hidden part of the generated language binding. From the perspective of the file format, appending adds altering sequence of blocks, instead of a single string block followed by a single type block. Means to add fields or instances to existing storage pools (see section 6.3) are made necessary by this feature as well. The omitted data is marked with $?_1$ and $?_2$ in Fig. 2: Fields marked with $?_1$ do only appear in the first block and are left away in all other blocks adding data of the respective type. Fields marked with $?_2$ are only present if the field is added to a type.

If appending is not used at all, the overhead, compared with a similar format that does not allow for appending, is about two bytes. If appending is used in a way, that only adds new fields or new storage pools, the overhead is still in the range of several bytes. Adding instances to existing storage pools is expected to create an overhead of about forty bytes, mostly caused by end-offsets of added field data.

If additional instances are added, the order of already instantiated fields is the order of their occurrence in previous blocks. If additional instances are added in combination with adding new fields, the new fields are located after the existing fields. The already existing fields contain data for the new instances, whereas new fields contain data for all existing instances.

6.2.3 Node Example

Let us assume two tools with two SKilL specification files:

Listing 19: Specification of the Node Producer Tool

```
Node {
  i8 ID;
}
```

Listing 20: Specification of the Node Color Tool

```
Node {
  i8 ID;
  /** for the sake of simplicity a string like "red" */
  string color;
}
```

Let us assume, the first tool produces two nodes (”23” and ”42”) and the second appends color fields to these nodes (”red” and ”black”). The layout of the resulting file, after the second tool is done, is given in figure 3. Actual field data is kept abstract, because serialization of field data will be explained below. The data produced by the first tool is S and T, the second tool produces S’ and T’.

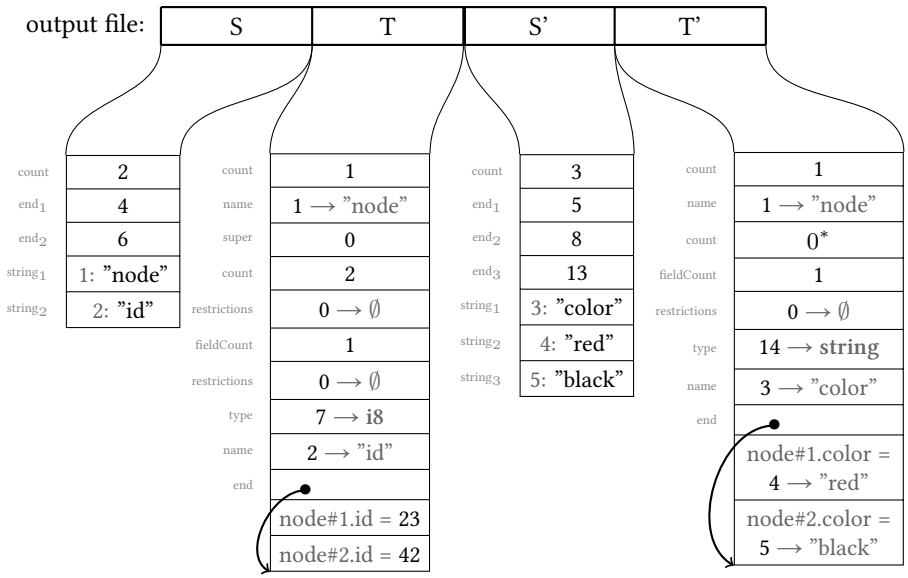

Figure 3: Illustration of the file obtained after running the example tool chain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations shall serve as a reminder of the meaning of the contents of the respective field. In section 6.6 there is a more complete example.

We will see in the next section, that fields referring to other objects are stored by reference. Strings are somewhat a special case of this. Strings are used both, as a first

class type and to represent type information itself. This has the consequence, that string pools contain user data and data, which might not be directly observed by a user.

Adding instances

Now let us assume, that the first tool is ran twice, adding (”23” and ”42”) in the first run and (”-1” and ”2”) in the second; we wont run the coloring tool, thus S/T are the result of the first run and S*/T* are the result of the second run.

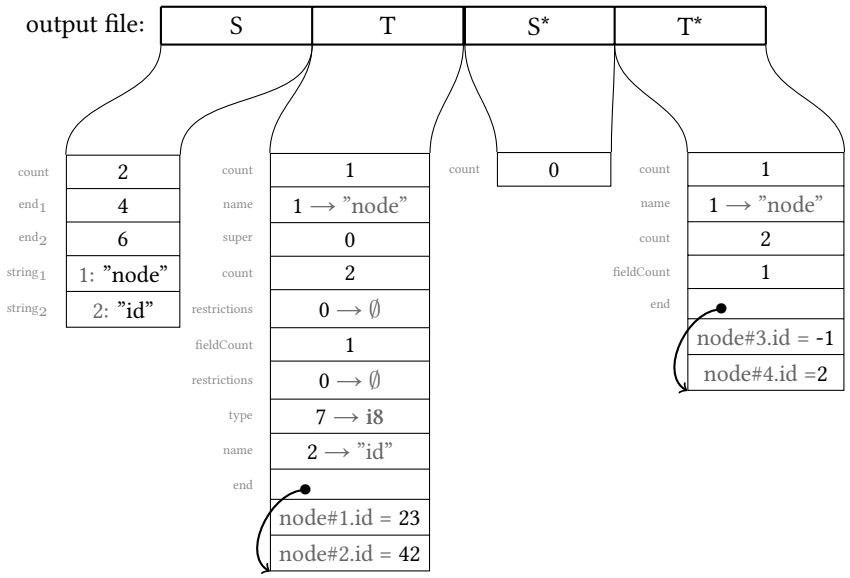

Figure 4: Illustration of the file obtained after running the example tool chain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations provide contents descriptions. Note that string blocks (and type blocks) can be empty, but must not be omitted.

6.3 Storage Pools

This section contains the serialization function for an individual storage pool. A storage pool stores the instances of a type, i.e. all field data of the type of the pool. We assume that storage pools are not empty. The serialization of an empty storage pool is the empty stream, i.e. it is simply skipped.[26]

Writing objects of a pool requires the following functions: $baseTypeName$: $\mathcal{U} \rightarrow S$, $typeName : \mathcal{U} \rightarrow S$ and $index : \mathcal{U} \cup \{\textbf{string}\} \rightarrow S$.

The baseTypeName is either the index of the string used as type name of the base type or `0x00`. The typeName is the index of the string used as type name of the argument type. The index is the unique index of the argument object. Indices are assigned ascending from 1 for instances of a base type, for each base type. For example the index 23 can be given to a *Node* object and a string ”hello”, because they do not

[26]This has the side effect, that only type information of instantiated types are present.

have a common base type. Indices do not have holes. Although indices are serialized using the v64 type, they are treated as if they were unsigned integers[27]

A basic concept of the serialization format is to store the data grouped by type into storage pools. This concept enables us to obtain type information of objects from their position in a file, instead of storing a type descriptor with each object.

If objects are referred to from other objects, those references are given as an integer, which is interpreted as index into the respective storage pool. The NULL pointer is represented by the index 0.

Each pool knows how many instances it holds. Storage pools with a supertype store the name of the supertype and a BPSI. Further, we assume that objects, which are written to a file, have indices such that for any type, all instances of that type have adjecent indices.

A short example (Fig. 5) illustrates the concept. It contains five types **A**, **B**, **C**, **D**, and **N**. For the sake of simplicity, each type has a single field of an arbitrary type τ_x (serialization of field data will be explained in the next section). **A** and **N** are base types. **A** has 6 instances. **B**, **C** are subtypes of **A**; **D** is a subtype of **B**. **B**/**C**/**D** have 4/1/1 instances and BPSIs of 2/6/5, respectively. The arrows represent the end-offsets stored in the field descriptions, which are used to separate the data part of the type block into field data. The *index*-row displays the index of the object that belongs to the serialized field.

For the sake of readability, the header of the type pool is omitted in the stream part of the picture.

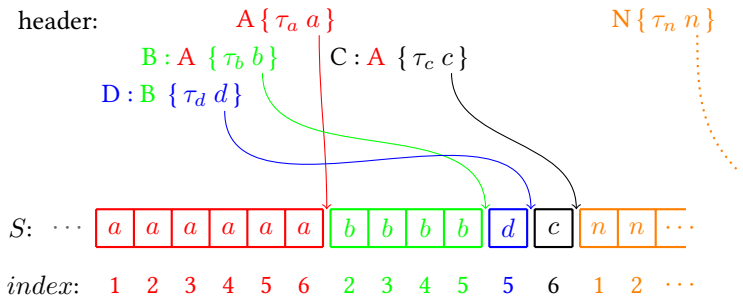

Figure 5: Field data of several pools stored in a type block. Arrows indicate the values of data fields of the respective field description in the header of the type block. S shows a part of the data chunk of a type block.

Although illustrations above might suggest an order of field data, this is in fact not the case. This allows for optimization during serialization, such as multi-threaded encoding of different fields at the same time.

6.3.1 Appending Fields to existing Pools

Appending a field to an existing pool is rather easy, because existing objects are not directly modified. The respective type block will have a type declaration with a field declaration of the added field. The added field data is stored in the data part of the type block.

[27] That is the index -1 refers to object number $2^{64} - 1$.

6.3.2 Appending Objects to existing Pools

If objects are added to existing pools, all type pools (i.e. the pools of the type and all its supertypes) have to be updated. If a supertype exists, the local base pool start index will give the start index of the added objects in the local pool. With the types as in figure 5 we can for example add several object in three blocks.

Let T1 contain 6 objects of types `aabbbc`, T2 contain 4 objects of types `bbdd` and T3 another 3 objects `acd`.

T1 contains the full type information of types A,B and C. There are 6 A instances, 3 B instances (with LBPSI=3) and 1 C instance (with LBPSI=6).

T2 contains only the type name of A and B (LBPSI=1) and field data for 4 instances each. Additinally, there is a new type D (with LBPSI=3) and D-field data for two instances.

T3 contains field data for A, B (LBPSI=2), C (LBPSI=2) and D (LBPSI=4) objects.

The in-memory representation after loading these objects is expected[28] to be either `aabbbbcbbddacd` or `aaabbbbbbdddcc`, depending on used hints and restrictions .

6.4 Serialization of Field Data

In this section, we want to describe the serialization of individual fields using the function $[\![\_]\!]_\tau$. The serialization of an object is done by serializing all its fields into the stream. In this section, we assume that the three functions defined in the last section are implicitly converted to streams using the `v64` encoding. We assume further, that compound types provide a function $size : \mathcal{T} \rightarrow \mathcal{I}$, which returns the number of elements stored in a given field. Let f be a non-constant[29] non-`auto` field of type t, then $[\![f]\!]$ is defined as[30]

- $\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ index(f) & else \end{cases}$
- $[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00}, & f = \texttt{NULL} \\ baseTypeName(f) \circ index(t) & else \end{cases}$ [31]
- $[\![\top]\!]_{\textbf{bool}} = \texttt{0xFF}$
- $[\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$
- $\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$
- $[\![f]\!]_{\textbf{v64}} = encode(f)$[32]
- $[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{f64}} = f$[33]
- $\forall g \in \mathcal{B}, n \in \mathbb{N}^+.t = g\texttt{[}n\texttt{]} \implies [\![f]\!] = [\![f_0]\!]_g \circ \cdots \circ [\![f_{n-1}]\!]_g$

[28] The in-memory representation is intentionally left unspecified. The expected behavior will be the result of a straightforward implementation.

[29] Constant fields are not serialized, because their value is already stored in the type declaration.

[30] We will use C-Style hexadecimal integer literals for integers in streams.

[31] We do not want to use type IDs here, because we do not want to touch all annotation fields if we modify the type Pool.

[32] With encode as defined in listing 22.

[33] Assuming the float to be IEEE-754 encoded (see [iee08] §3.4), which allows for an implicit bit-wise conversion to fixed sized integer.

- $\forall g \in \mathcal{B}, n = size(f), t \in \{g\texttt{[]}, \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!] = [\![n]\!]_{v64} \circ [\![f_0]\!]_g \circ \cdots \circ [\![f_{n-1}]\!]_g$
- Maps are serialized from left to right by serializing the keyset and amending each key with the map structure which it points to[34]. In case of Maps with two types, this is equal to a list of key value pairs. A field f of type `map<T,U,V>` is serialized using a schema:

$$[\![f]\!] = [\![size(f)]\!] \bigcirc_{i=1}^{size(f)} [\![f.t_i]\!] \circ [\![size(f[t_i])]\!] \bigcirc_{j=1}^{size(f[t_i])} [\![f[t_i].u_j]\!] \circ [\![f[t_i][u_j]]\!]$$

  Note that we treat maps like map<T, map<U,V> >.

- $[\![r]\!]_{restriction} = [\![r.id]\!]_{v64} [\![r.arg_1]\!]_{string} \circ \cdots \circ [\![r.arg_n]\!]_{string}$

  Note that restrictions store the image of their arguments as a string.

- $$[\![t]\!]_{type} = \begin{cases} [\![id]\!]_{i8} \circ [\![val]\!]_t & id \in [0,4] \\ [\![id]\!]_{i8} & id \in [5,14] \\ \texttt{0x0F} \circ [\![i]\!]_{v64} \circ [\![T]\!] & t = T[i] \\ \texttt{0x11} \circ [\![T]\!] & t = T[] \\ \texttt{0x12} \circ [\![T]\!] & t = list<T> \\ \texttt{0x13} \circ [\![T]\!] & t = set<T> \\ \texttt{0x14} \circ [\![n]\!]_{v64} \circ [\![T_1]\!] \circ \cdots \circ [\![T_n]\!] & t = map<T_i, \ldots, T_n> \\ [\![32 + storagePoolIndex(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$$

*id*s of restrictions and types are listed in appendix E. The holes are intentional and enable future built-in types without breaking the file format.

Note that the function $storagePoolIndex$ is implicitly given by the order of storage pools appearing in a file and has therefore to be computed both upon serialization and deserialization. The first $storagePoolIndex$ is 0.

6.5 Endianness

Files are stored in network byte order, as described in RFC1700, Page 2 [RP94].

If a client is running on a little endian machine, the endianness has to be corrected, both when reading and writing files. This can be done by changing the implementation of $[\![\_]\!]_{i*}$- and $[\![\_]\!]_{f*}$-translations. Note that some standard libraries provide functions to read and write binary data in network byte order.

6.6 Date Example

This section will provide a concise example of how serialization takes place. For the sake of simplicity and brevity, we will serialize two objects of a single (simple) type into a stream. We will use the following file format:

Listing 21: Date File Format

```
Date {
  v64 date;
}
```

[34]Maps are expected to be sparse, i.e. with less then half their fields being non default values.

Further, we want to store two objects with *date* values 1 and -1.

The first thing we have to do is to collect the objects to be stored. In a set-theory inspired notation with square brackets indicating records we would get something like:

$$Date = \{[date : 1], [date : -1]\}$$

Now we have to create storage pools. We start with the creation of type information. The date pool looks something like this:

$$[name : \text{"}date\text{"}, super : 0, count : 2, restr. : \emptyset, fields[$$

$$1 : [restr. : \emptyset, t : \texttt{v64}, n : \text{"}date\text{"}, [1, -1]]]$$

Now we can see, that we have a string, which has to be serialized, but is not yet in the string pool (which was empty up until now), so we create a string pool[35] as well:

$$[1 : \text{"}date\text{"}]$$

Now we can start to encode the pools:

1. write the string block:

   $[\![[1 : \text{"}date\text{"}]]\!] = [\![1]\!]_{v64} \circ offset(\text{"}date\text{"}) \circ [\![\text{"}date\text{"}]\!]$
   $= [\![1]\!]_{v64} \circ [\![4]\!]_{i32} \circ \texttt{64 61 74 65}$
   $= \texttt{01 00 00 00 04 64 61 74 65}$

2. write the type block:

   To keep things readable, we will first encode everything but the field data:

   $[\![[name : \text{"}date\text{"}, super : 0, count : 2, restr. : \emptyset, \cdots]\!]$
   $= [\![1]\!]_{v64} \circ [\![\text{"}date\text{"}]\!] \circ 0 \circ [\![2]\!]_{v64} \circ [\![\emptyset]\!]_{R[]}$
   $= 1 \circ index(\text{"}date\text{"}) \circ 0 \circ 2 \circ [\![size(\emptyset)]\!]_{v64}$
   $= 1 \circ 1 \circ 0 \circ 2 \circ 0 = \texttt{01 01 00 02 00}$

   Now we can continue with the field:

   $[\![[\cdots fields[1 : [restr. : \emptyset, t : \texttt{v64}, n : \text{"}date\text{"}, [1, -1]]]]\!]$
   $= [\![size(fields[\cdots])]\!] \circ [\![[restr. : \emptyset, t : \texttt{v64}, n : \text{"}date\text{"}, offset([1, -1])]]\!] \circ [\![[1, -1]]\!]$
   $= 1 \circ [\![\emptyset]\!]_{rest[]} \circ [\![\texttt{v64}]\!]_{type} \circ [\![\text{"}date\text{"}]\!] \circ offset(\text{"}date.date\text{"}) \circ [\![[1, -1]]\!]$
   $= 1 \circ 0 \circ \texttt{0B} \circ index(\text{"}date\text{"}) \circ offset(\text{"}date.date\text{"}) \circ [\![1]\!]_{v64} \circ [\![-1]\!]_{v64}$
   $= 1 \circ 0 \circ \texttt{0B} \circ 1 \circ offset(\text{"}date.date\text{"}) \circ \texttt{01} \circ \texttt{FF FF FF FF FF FF FF FF FF}$
   $= 1 \circ 0 \circ \texttt{0B} \circ 1 \circ [\![10]\!] \circ \texttt{01} \circ \texttt{FF FF FF FF FF FF FF FF FF}$
   $= \texttt{01 00 0B 01} \circ \texttt{0A} \circ \texttt{01 FF FF FF FF FF FF FF FF FF}$
   $= \texttt{01 00 0B 01 0A 01 FF FF FF FF FF FF FF FF}$

   The type block is now serialized to the stream:

   ```
   01 01 00 02  00 01 00 0B  01 0A 01 FF  FF FF FF FF  FF FF FF FF
   ```

3. writing the output:

   The remaining work is just to write the string block and the type block to a file, starting with the string block, so we get:

   ```
   01 00 00 00  04 64 61 74  65 01 01 00  02 00 01 00
   0B 01 0A 01  FF FF FF FF  FF FF FF FF  FF
   ```

Deserialization of this stream is explained in section 7.1.

[35]The string pool, unlike regular storage pools is just a length encoded array of strings.

7 Deserialization

Deserialization is mostly straightforward.

The only notable property is that a binding has to ensure that no objects can be allocated that have more fields in the deserialized file than in the specification. This is important because it prevents from violating invariants of other tools in a tool chain by allocating objects through a partial view onto a type.

The general strategy is:

1. the string block is processed
2. the header of the type block is processed
3. required fields are parsed using the type and position information obtained from the respective block
4. until the end of file has been reached, goto step 1

7.1 Date Example

Let d be the deserialization function – basically the inverse function of $[\![\_]\!]$. We want to read the sequence we created during the serialization example in section 6.6:

```
01 00 00 00  04 64 61 74  65 01 01 00  02 00 01 00
0B 01 0A 01  FF FF FF FF  FF FF FF FF  FF
```

1. deserialzation of a string block:

   $d(0100000004646174 65010100 \cdots)$
   $\longrightarrow$a string block starts with a v64 indicating the number of strings stored inside

   $d(01)d(00000004646174 65010100 \cdots)$
   $\longrightarrow$we got one string

   $d(01)d(00000004)d(64617465 010100 \cdots)$
   $\longrightarrow$the next string has 4 bytes, the block ends in 4 bytes

   $string[1 : d(64617465)]d(0101 \cdots)$
   $\longrightarrow$build the string pool with the first string block

   $string[1 : "date"]d(0101 \cdots)$
   $\longrightarrow$we processed the string directly, because lazy evaluation makes the example rather confusing

2. deserialzation of a type block (reading the header):

   $d(01)d(0100020001000B010A01 \cdots)$
   $\longrightarrow$there is one type definition in this block; read its name

   $d(01)d(00020001000B010A01 \cdots)$
   $= "date"d(00)020001000B010A01 \cdots)$
   $\longrightarrow$we do not know the type "date" yet (in terms of processing the file), so we expect super type information, count, restrictions and field declarations

   $[name : "date", super : 0]d(02)d(00)d(01)d(000B010A01FF \cdots)$
   $\longrightarrow$Date has no super type, thus the next field is not a local BPSI and we can read until the number of fields.

$[name : \text{"}date\text{"}, super : 0, count : 2, restr. : \emptyset, fields[1 : \_]]$
$d(00)d(0B)d(01)d(0A)01FF\cdots)$
→We did not know date yet, thus we do not know the type of the first field. Therefore the next information are the fields restrictions, type, name and offset.

$[name : \text{"}date\text{"}, super : 0, count : 2, restr. : \emptyset, fields[1 : [restr. : \emptyset, t : \texttt{v64}, n : \text{"}date\text{"}, offset : 10]]]d(01FF\cdots)$
→We read all data belonging to the header of this type block, thus we know that the next 10 bytes are field data. The file ends after the next 10 Bytes, so we can continue constructing objects.

3. construction of date objects:

   $Date = \{[date : \_], [date : \_]\}d(01FF\cdots)$
   →using the data from the header, we created two date objects; the only thing left is parsing field data for the date fields and setting the data accordingly

   $Date = \{[date : 1], [date : \_]\}d(FF\cdots)$
   →the first field is a 1

   $Date = \{[date : 1], [date : -1]\}$
   →the second field is a -1; we reached the end of the field data on the last object, so everything is fine – no additional or missing data

   Unsurprisingly, the restored objects are exactly the objects we serialized in section 6.6.

8 In Memory Representation

This section is about the generated API provided to the programmer and the representation of objects inside memory. It is meant as a concise guideline for implementers of language binding generators.

The behaviour in case of a mismatch between the expected type of a field and the type stored in a file is left unspecified. This will be changed in a future version. For now only matching types can be assumed to be compatible and portable.

8.1 API

The generated API has to be designed in a way that integrates nicely with the language's programming paradigms. For example in Java it would be most useful to create a state object, which holds state of a bunch of serializable data and provides iterators over existing objects, as well as factory methods and methods to remove objects from the state object. The serialized types can be represented by interfaces providing getters and setters, using hidden implementations only visible for the state object.

8.2 Representation of Objects

The combination of laziness and consistency has the effect that representation of objects inside memory is rather difficult. This section describes data structures and algorithms which can do the job in a sufficiently efficient way. In this section, we assume that all fields are present as arrays of bytes. We will describe the effects of parsing fields in an unmodified state, in a modified state, how to modify a state and finally how to write a state back to disk.

8.2.1 Proposed Data Structures

A state has to contain at least

- an array of strings
- type information obtained from a file
- storage pools

A storage pool has to hold the images of (or references to) fields, which are not yet parsed.

Objects are required to have an ID field, which corresponds to the ID of the deserialized(!) state. This field is required in order to map the lazy fields to the correct objects. It can also be reused in the serialization phase to assign unique IDs, which will be used instead of pointers.

Objects of types with eager fields should have the respective fields. For example, the declaration `T {t a; !lazy t b;}` should be represented by an object

```
InternalTObject {
  long ID;
  t a;
  /* getA, setA... */

  StoragePool tPool;
  /* getB, setB... */
}
```

Note that the pointer to the enclosing storage pool is required for the correct treatment of lazy and distributed fields. This is the case, because the pool holds the field data.

Now that we have a representation of objects, we still have to organize storage of objects across storage pools. The possibility of inheritance requires a view onto stored objects, which looks as if they were stored in multiple pools at the same time. We propose to store supertype and subtype information inside pools as references to the respective pools. The objects should be stored as a "double linked array list"[36], which is basically a linked list containing array lists:

Each pool stores the objects, with the static type of the pool in an array list. The pools of subtypes are stored in a linked list. Now an iterator over all instances of a type uses an iterator over all instances of a pool in combination with iterators over the pool and all its subpools. Therefore creating and deleting objects is an amortized O(1) operation[37] and it is guaranteed to maintain the semantic structure of the file, if IDs are not updated in phases other than reading and writing a file.

Note that the mere presence of distributed fields will change the worst case runtime complexity of these operations from (amortized) $O(1)$ to $O(log(n))$, where n is the largest number of objects with a common distributed field in a state.

[36] In Java this would be a LinkedList<ArrayList< ? extends B > >, where B is the provided interface of the base type of a pool. Note however, that the LinkedList-part is expected to be realized using references between storage pools.

[37] This can be achieved by appending new objects to the end of the pool and deleting existing objects by creating holes inside the pool. The compaction of the storage pool is done while writing the output and can be amortized into the writing costs.

8.2.2 Reading Unmodified Data

Reading unmodified data is basically done by creating objects with ascending IDs while processing all eagerly processed fields. Pointer resolution in an unmodified state is an O(t) operation, where t is the number of subtypes of the static type of the pointer[38].

During the reconstruction of the initial dataset, an array in the base pool may be used to reduce the cost to O(1)[39]. However, this helper array has to be dropped, as soon as the base pool is modified.

8.2.3 Modifying Data

The only legal way of modifying data is to access it through the generated API, which provides iterators, a type safe facade, factories and means of removing objects from states for each known type. A modification is any operation that might invalidate any existing object ID, i.e. deleting objects or inserting objects into non-empty storage pools. Adding objects to empty storage pools does not count as a modification in the sense of a modified state, because it is not possible that a pointer to such an object lurks in an yet untreated field.

8.2.4 Reading Data in a Modified State

Reading Data from disk in a modified state, is very similar to reading data in the unmodified state. Except that resolution of stored pointers can no longer rely on the invariant that the ID of an object is also the index into the base type pool. There is a solution for this problem using $O(t + log(n))$, where n is the number of instances with the same static type. However, the straightforward implementation is $O(t + n)$.

With this difference in mind, we strongly recommend adding a dirty flag to each storage pool which traces modifications. This is expected to eliminate the additional cost, because transformation of stored state is often the subject of monotonic growth, because each step of a computation usually adds instances of a type, which had no instances in a previous step.

8.2.5 Writing Data

Data which is lazy and modified can not directly be written to disk, because any data which can potentially refer to modified data has to be evaluated. After evaluating the respective data, serialization is straightforward. The evaluation of lazy data referring to potentially modified data can be done in $O(out)$, where out is the size of the output file. Writing the output is also in $O(out)$, thus writing is not per se a performance issue.

8.2.6 Final Thoughts on Runtime Complexity

Although the last sections read a lot like accessing serializable data is unnecessary expensive, this is in fact not the case.

[38] This is usually a very small constant.

[39] The creation time for the array can be paid for during the creation of the base pool.

Reading data without modifying is in $O(in)$, even if only a part of the data is read[40]. This is mainly caused by the requirement of being able to process unknown data correctly. The actual cost should be limited by the cost of sequentially reading (or seeking through) the input file from disk.

Reading data, modifying it and writing it back is $O(in + m + out)$, which is not surprising at all, because one has to pay for reading the initial file, writing the complete output file and for performing the modifications.

Only the usage of a lot of non-monotonic lazy or distributed data is expensive. It is generally advised against using the lazy attribute if a field is read for sure during the lifetime of a serializable state. On the other hand, lazy fields can be very valuable, if their data is used for error reporting or debugging.

9 Future Work

Further research has to be done in the area of restrictions.

Treatment of incompatible types and the notion of incompatible itself is underspecified. It is currently unknown whether or not an automated *upcast* shall take place if possible.

A general purpose viewer and editor for SKilL files should be implemented.

A future version of the specification language will add interfaces. The resulting types will be binary compatible with existing types. Interfaces will allow grouping of common properties of subtypes. Interface declarations will mostly behave as if they were hints, although they will change the generated API in a straight forward way.

We will look into ways of implementing type-safe unions and enumeration types. It has to be evaluated, whether they should be first class citizens of the SKilL type system, or if they can be represented with hints or restrictions. We will also evaluate possible ways to support type casts, in order to allow for using language specific types, such as bit-fields.

We will introduce a general assertion restriction, which can be used to assert per instance invariants with sufficiently powerful expressions.

We will evaluate the need and impact of true comments inside of SKilL definition files. Forcing users of the definition language to write their comments in a way that is emitted to the generated source code, might be very beneficial after all.

We will evaluate a template import statement, which can be used to import files together with a substitution. That way, one can create more complex data structures such as B-Trees. This feature is not a priority and only useful for large files and projects and requires some kind of type casts to yield the desired effect automatically.

We might add *add* and *subtract* declarations to the file format, starting with `++` or `--`, which allow taking away or adding fields to types. This feature can be used to create a lean interface for individual tools.

We will further evaluate means of supporting common high level specification tasks, such as creating effective views for individual tools of a tool chain. This can either be in the form of a skill specification editor or an extended specification language.

[40]Note that a file can contain little payload compared to the type information stored in the file, therefore $O(in)$ is a sharp complexity estimate.

Part II
Appendix

A Variable Length Coding

Size and Length information is stored as v64, i.e. 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 22: Variable Length Encoding

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

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

Listing 23: Variable Length Decoding

```
uint64_t decode(uint8_t* v64) {
  int count = 0;
  uint64_t result = 0;
```

```
  register uint64_t bucket;
  for(; count < 8 && (*pV64)&0x80; count++, v64++) {
    bucket = v64[0];
    result |= (bucket&0x7f) << (7*count);
  }
  bucket = v64[0];
  result |= (8==count?bucket:(bucket&0x7f)) << (7*count);
  return result;
}
```

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.

Serialization

- If new instances of a type would be appended to a file, which contains more field definitions than the generator, an error has to be reported. The new instances lack fields, which is very likely to violate an invariant. If this problem is encountered in a project, it is likely that either the unknown data forms a subtype or has to be considered by the tool producing the error.

Deserialization

- If `EOF` is encountered unexpectedly, an error must be reported before producing any observable result.
- If an index into a pool is invalid[41], an error must be reported.
- If the deserialization of a storage pool does not consume exactly the `sizeBytes` specified in its header, an error must be reported. This is a strong indicator for a format change.
- If the serialized type information contains cycles, an error has to 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 pool contains instances which, based on their location[42] in the base pool, should be subtypes of some kind, but have no respective subtype 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. This is a strong indicator for either file corruption or a bug in the previously used back-end.
- All known constant fields of a type have to be checked before producing any observable objects of the respective type. If some constant value differs from

[41] because it is larger then the last index/size of the pool

[42] I.e. after a subtype but not inside another subtype

the expected value, an error must be reported, which contains at least the type, the field type and name, the type block index, the total number of type blocks, the expected value and the actual value.

- If a serialized value violates a restriction or the invariant of a type,[43] an error must be reported as soon as this fact can be observed.

C Core Language

The core language is a subset of the full language which must be supported by any generator, which is called a SKilL core language generator. Features included in the core language are:

- Integer types `i8` to `i64` and `v64`
- `string`, `bool` and `annotation`
- Compound types
- User Types with sub-typing
- `const` and `auto` fields
- Reflection

Thus the remaining parts required for full SKilL support are:

- Floats
- Restrictions
- Hints
- Language dependent treatment of comments, e.g. integration into doxygen or javadoc.[44]
- Name mangling to allow for usage of language keywords or illegal characters (unicode) in specification files, without making a language binding impossible.
- A public reflective interface allowing reading of instances and type information of unknown types.

D Numerical Limits

In order to keep serialized data platform independent, one has to respect the numerical limits of the various target platforms. For instance, the Java Virtual Machine can not deal with arrays with a size larger then about 2^{31}. Therefore we establish the following rule:

(De-)serialization of a file with an array of more then 2^{30} elements or a type with more then 2^{30} instances may fail because of numerical limits of the target platform. Although, strings can have at most 2^{32} bytes of data, strings are usually represented as array of characters, therefore their length is expected to not exceed 2^{30} characters as well.

[43] Including sets containing multiple similar objects.

[44] This may even require a language extension providing tags inside comments which are translated into tags of the respective documentation framework.

E Numerical Constants

This section will list the translation of type IDs (as required in section 6.4) and restriction IDs (see section 5.1 and 6.4).

Type Name	Value
const i8	0
const i16	1
const i32	2
const i64	3
const v64	4
annotation	5
bool	6
i8	7
i16	8
i32	9
i64	10
v64	11
f32	12
f64	13
string	14
T[i]	15
T[]	17
list<T>	18
set<T>	19
map<T_1, ..., T_n>	20
T	32 + $index_T$

(a) Type IDs

Restriction Name	Value
range	0
nullable	1
unique	2
singleton	3
constantLengthPointer	4
monotone	5

(b) Restriction IDs

Glossary

base type The root of a type tree, i.e. the farthest type reach able over the super type relation. 28

built-in type Any predefined type, that is not a compound type, i.e. annotations, booleans, integers, floats and strings. 16, 18

subtype If a user type A extends a type B, A is called the sub type (of B). 13, 35, 36, 40

supertype If a user type A extends a type B, B is called the super type (of A). 13, 29, 35

unknown type We will call a type *unknown*, if there is no visible declaration of the type. Such types must not occur in a declaration file, but they can be encountered in the serialization or deserialization process. 13

user type Any type, that is defined by the user using a type declaration. 13, 16, 18

visible declaration We will call a type declaration *visible*, if it is defined in the local file, or in any file transitively reachable over include directives. 42

Acronyms

API Application Programming Interface. 15, 24, 34, 37

BPSI Base Pool Start Index. 29, 33

EXI Efficient XML Interchange Format. 6

LBPSI Local Base Pool Start Index. 26, 30

SKilL Serialization Killer Language. 5–8, 12, 13, 15, 18, 19, 27, 37, 40

v64 Variable length 64-bit signed integer. 17, 24–26, 28, 38

XML Extensible Markup Language. 4–6

XSD XML Schema Definition Language. 6

References

[Apa13] Apache Software Foundation. *Thrift Types*. `http://thrift.apache.org/docs/types/`, 2013.

[Goo13] Google. *Protocol Buffers Language Guide*. `https://developers.google.com/protocol-buffers/docs/proto`, 2013.

[GSMT+08] Shudi Gao, C. Michael Sperberg-McQueen, Henry S. Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. W3c xml schema definition language (xsd) 1.1 part 1: Structures. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620, June 2008.

[iee08] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[jav13] *Javadoc Technology*. `http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html`, 2013.

[LA13] Chris Lattner and Vikram Adve. *LLVM Language Reference Manual*. `http://llvm.cs.uiuc.edu/docs/LangRef.html`, 2013.

[Lam87] David Alex Lamb. Idl: Sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, 1987.

[LYBB13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java Se, 7 Ed.* Always learning. Prentice Hall PTR, 2013.

[PGM+08] David Peterson, Shudi Gao, Ashok Malhotra, C. Michael Sperberg-McQueen, and Henry S. Thompson. W3c xml schema definition language (xsd) 1.1 part 2: Datatypes. World Wide Web Consortium, Working Draft WD-xmlschema11-2-20080620, June 2008.

[RP94] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700 (Historic), October 1994. Obsoleted by RFC 3232.

[SK11] John Schneider and Takuki Kamiya. Efficient xml interchange (exi) format 1.0. Technical report, W3C - World Wide Web Consortium, `http://www.w3.org/TR/2011/REC-exi-20110310/`, March 2011.

[TDB+06] S. Tucker Taft, Robert A. Duff, Randall Brukardt, Erhard Plödereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[vH13] Dimitri van Heesch. *Doxygen User Manual*. `http://www.stack.nl/~dimitri/doxygen/manual/`, 2013.

[xml06] Extensible markup language (xml) 1.1 (second edition). W3c recommendation, W3C - World Wide Web Consortium, `http://www.w3.org/TR/2006/REC-xml11-20060816/`, September 2006.