Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3752

SKilL Graph Visualiziation and Manipulation

Moritz Rathgeber

Course of Study:	Informatik
Examiner:	Prof. Dr. Erhard Plödereder
Supervisor:	Timm Felden
Commenced:	2016-09-07
Completed:	2017-03-09
CR-Classification:	D.2.2, H.5.2

Abstract

The serialisation system SKilL stores mutually referenced objects in a binary file format. Thus, a special programme is required to view the serialised data or perform ad hoc modification. This thesis discusses the implementation of such an editor in the programming language Scala.

After giving a sketch of the SKilL system, the main part of this thesis is concerned with the visualisation of the data. Due to references, the contents of an object can usually not be understood in isolation but only in combination with the objects which it references. The objects and their mutual references are, therefore, visualised as a node-link diagram of the neighbourhood of a given object. For the layout of the diagram, existing force directed placement algorithms are reviewed and solutions from different sources are combined. In order to produce comprehensible layouts, an additional constraint, which aligns all links that represent the same field in the underlying data, is added to the placement algorithm. The implementation works reasonably well with small graphs, but the quality of the layout deteriorates for larger or highly connected graphs.

The remainder covers the modification, creation, and deletion of objects. No satisfying solution was found for nested maps. Finally, a simple search function is sketched.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

SKilL (serialisation killer language, Felden [Fel17]) is a language and platform independent serialisation format which targets applications involving large data sets. For efficiency reasons the data are stored in a binary format, thus a special editor program is required for performing ad hoc lookups and modifications in the serialised files. The development of such an editor is the scope of this thesis.

The cases that the editor has to cover depend on the file format. We therefore begin with a description of the structure of SKilL files in chapter 2. We do so from a high level perspective, because we use an existing program library to access SKilL data. As the library deals with the byte level representation of the data, the description starts at the level of abstraction provided by the SKilL library for the implementation language Scala [Oa04].

SKilL files store user data records, which can, and usually do, refer to each other. It is quite possible, that a record stores nothing but references, and so the meaning of a record can not usually be understood in isolation. They must therefore be viewed together with the records to which they refer, but care must be taken not to show too much. In chapter 3 we develop a way to show the records and their references in a node-link diagram, which the user can interactively explore.

The records in a SKilL file conform to a user defined type system which is stored in the binary files along the user data. How the type system is displayed is quickly described in chapter 4.

In chapter 3, we will argue, that it is easier to grasp the contents of a SKilL file when irrelevant parts of the data are hidden. When it comes to modifying the file, we will, however, need the complete contents available. To this end, we define a second editing view of the objects in chapter 5.

No editor is complete without a search function; in chapter 6 a simple query language is outlines, which is inspired by the semantic web query language SPARQL [PS08]. The queries permit searching records not only based on the values of their fields but also on the values of the fields of referenced objects.

Chapter 7 concludes with a summary of the results and recommendations for further developments.

2 The SKilL Serialisation System

The core of SKilL is a platform and language independent binary file format which stores user data records, called *SKilL objects*, together with a description of their format or data type. This binary format is accompanied with a specification language that describes the data types of the records one wants to store in the binary files. The system is completed by program generators, which translate specifications into language specific bindings. A binding is a program library which reads and writes SKilL binaries that conform to the specification it was built for. The application programming interface (API) of the binding represents the contents of a SKilL file with the native data types of the host language. The SKilL implementation currently used at the Institute of Software Technology at the University of Stuttgart (ISTE) can generate bindings for the programming languages Ada, C, C++, Haskell, Java, and Scala.

SKilL stores the description of the data types in the binary files in order to guarantee type safety. The bindings check that the type system in the binary file matches their specification. However, specifications are subject to change, and one does not want to lose existing of binary data when a specification is amended. Therefore, bindings accept some deviations between the specification and the binary file. For example, when record type in the binary lacks a field that is present in the specification, the binding can add the field and supply a default value. So, when a program opens a SKilL file, the effective type system is a combination of the one already present in the file and the one from the specification built into the program. This combined type system is written back to the file, and so SKilL files can collect several unrelated extensions from mutually compatible specifications.

The bindings, therefore, provide a reflective API, through which the type system as stored in a file can be inspected, and fields and types which are not part of the specification can be accessed. As an extreme case, one can build a binding for an empty specification and use the reflective API throughout. Such a binding can open all SKilL files since all types in the specification are trivially compatible with the binary file. This is how the editor is accessing SKilL files.

This chapter contains a description of the SKilL type system, for this determines the data that the editor has to be able to show and manipulate. We also establish the terminology used in the remaining chapters. The binary format is not covered here, because the existing binding takes care of it; interested readers are referred to the work of Felden [Fel17]. At the end of the chapter we look how the user data can be retrieved from the file and inspect some properties of real world skill data which we cannot derive from the specification of the file format but which will guide our later decisions.

2.1 The SKilL Type System

SKilL uses a type system which is similar to those of object oriented programming languages. The data types of objects that exist independently resemble class types—without methods, of course—and are called *user types*. Fields of user type objects have a *field type*, which is either a *built-in type* like integer, boolean, and string, a *compound type* like array and set, or a reference type. Like in the Java programming language, user types are only nested through references. Also, there are no references to built-in or compound types.

User types and fields can be refined by *restrictions* like ‘singleton’ for user types or ranges for numeric fields.

2.1.1 User Types

A user type consists of a sequence of typed fields. An object having a user type stores a tuple of values which have the type of its fields. Optionally, a user type can extend another user type, that is, it inherits all fields of the extended type. The former is then called a *sub-type* of the latter, which, in turn, is called the *per-type* of the former. A type which has no super type is called a *base type*.

User types and their fields are identified by names. Upper case and lower case letters are not distinguished, because that would cause implementation problems in languages like Ada which do not do so either. For efficiency reasons, the names are converted to lower case before they are stored to SKilL binaries. Therefore, the identifiers displayed in the editor do not always read nicely and may require some guesswork as to where the words end.

SKilL permits the re-use of a field name in a sub-type. This allows users to extend ‘foreign’ specifications by sub-typing without causing binary incompatibilities when fields are added to the super type. Consequently, field names may have to be disambiguated by stating the name of the type that introduced the field.

Every user type can be used as a field type. A user type field stores a reference to an object of the indicated type or one of its transitive sub-types. Objects are uniquely identified by their base type and a consecutive number, which is called their *SKilL ID*. User type references are usually nullable, i.e. they can be set to refer to nothing.

2.1.2 Built-in Types

Aiming for language independence means that the built-in types have to be representable by the types of the host language, therefore most of them are well-known to any programmer and listed without much comment:

- A field of type `bool` stores one of the truth values `false` and `true`.
- Fields of type `i8`, `i16`, `i32`, `i64`, and `v64` store signed[1] integer numbers occupying the number of bits indicated in the type name.

  `v64` is serialised using a variable length encoding, but for an application this is abstracted away by the binding.
- Fields of type `f32` and `f64` store floating-point numbers in the IEEE 754 single and double format.
- A field of type `string` can store an unbounded sequence of Unicode characters.
- An `annotation` field stores a reference to an arbitrary other object.

The one outlier in this list is annotation, which stores a reference to an object having an arbitrary user type. A program must enquire the type of the referenced object before it can access its fields. In the following, annotations will usually be discussed together with references to user type object under the heading *reference types*. The remaining built-in types are called *primitive types*.

Strings are implemented as reference types, which improves space efficiency by removing duplicate strings from the binary file. However, the language bindings expose a value semantics in that they will allocate a new string or re-use an existing one when a string field is written instead of changing the value of the potentially shared string object. We can largely ignore this implementation detail, but we will have to be aware of the fact that string fields can store null-references.

2.1.3 Compound Types

When $\tau_1, \tau_2, \ldots$ are ground types, i.e. built-in types or references to user types, the following types are compound types:

- List types `list<`τ_1`>` and variable length array types τ_1`[]`. Fields having one of those types can store unbounded sequences of values of type τ_1.

[1]Though not mentioned explicitly in the specification [Fel17], negative numbers use two's complement representation and the extra negative number—$-128, -32768, \ldots$—is a valid value.

- Fixed length array types τ_1[n]. Fields having this type store a sequence of exactly n values of type τ_1.
- Set types set<τ_1>, fields of which store a sequence of distinct values of type τ_1.
- Map types map<τ_1, τ_2, ..., τ_n>, fields of which store a partial function with maps a key of type τ_1 to a value of type τ_n if $n = 2$ or map<τ_2,~...,~τ_n> otherwise.

The difference between lists and variable length arrays is how the language binding maps them to the data types of the host language. Lists have to support resizing while arrays need not to.

Note that, while the grammar of specification language appears to make it possible to define containers of containers, sets of sets, say, through typedefs, this is explicitly forbidden in section 4.4.3 of the specification [Fel17].

2.1.4 Constant Fields

Constant fields are different from the other fields encountered so far in that their values are not stored in objects but in type itself; after all they are constant. Their values are integers which are used as magic numbers or version numbers to prevent a binding from reading a file with an unrelated or incompatible type system. For every type that is present in the binary file as well as in the specification, the values of the constant fields must be equal in both places or the specification must explicitly accept the value found in the file. As the editor does not use a specification, it just displays the values of the constants found in the binary.

2.1.5 Restrictions

User type and field definitions can be annotated with *restrictions*, which serve different purposes. Most of them restrict the instances which a user type or the values which a field can have—hence the name. Some control the way the data are serialised and are semantically transparent. A last one defines initial default values. In SKilL specification files, restrictions start with an @ sign, a practice which we will follow.

User types and fields each have a separate set of restrictions that can be applied to them. User type restriction are not handled by the current Scala binding. We list them merely for completeness:

- When a type is annotated with `@unique`, any two instances of this type must differ in the value of at least one field.
- A type restricted with `@singleton` can have at most one instance. Singleton types must not have sub-types.

- Instances of a `@monotone`-restricted type must, once written to a file, not be modified or deleted, but new instances may be appended. Sub-types of monotone types are monotone as well.
- A type with restriction `@abstract` must not have any instances itself. Its sub-types usually have.
- The restriction `@default(`τ_2`)`, where τ_2 is a singleton-restricted sub-type of the default-restricted type, specifies that the default value of fields referencing this type should by the single object of type τ_2; fields can override this type-wide default by specific ones. Sub-types inherit the default but may change it—and may have to if the default type is not one of their sub-types.

The fields in a user type can be subject to the following restrictions, which, if the field has a compound type, apply to its members. Fields with a map type can not be restricted.

- A `@nonnull`-restricted string or user-object-reference must not be set to null, i.e. it must refer to some string or object.

  Can not be used with numeric, boolean, and annotation fields.
- When a field has a `@default(`x`)` restriction, it is initialised with x when a new object is created or when an object is read from a binary file which does not contain the field. The second case applies only to bindings generated for a specification that contains the default restriction.

  For fields that have numeric, boolean, or string type, x is a constant value of the corresponding type. For reference and annotation fields, x is a singleton restricted user type and the field is initialised with a reference to the single instance of that type, provided it exists.
- A numeric field with restriction `@range(`min, max`)`, where min and max are constant values of the numeric type in question, will only store values in the range from min to max inclusive.

  The one-sided forms `@min()` and `@max()` and the forms with exclusive bounds are mainly[2] notational variants and translated to inclusive ranges internally.
- The restriction `@coding(`x`)` can add a further encoding layer between the in-memory and on-disk representations of the value of the field; for example, one might add LZMA compression for large fields.

  The coding restriction does not change the semantics of a field from the user's point of view, provided the language binding knows the encoding in question. If it does not,

[2]Min and max add to the expressiveness of the specification language because the user can not specify the floating-point values $-\infty$ or $+\infty$ as lower or upper bound in a range restriction.

which is well possible because SKilL leaves the set of codings undefined, the field can not be accessed.

- A reference or annotation field with `@constantLengthPointer` restriction is serialised with the object ID stored as a fixed width integer instead of the usual variable length one. This does not change the behaviour from an abstract point of view, but helps implementations to modify contents of large files on disk.

  Only applies to reference fields.

- A reference field with restriction `@oneOf(`τ_1`, ..., `τ_n`)` can only refer to instances of the types $\tau_1, \ldots, \tau_n$; this does not preclude null values.

Regarding the restrictions that really restrict the set of possible values of a field, there is the question whether they have to be obeyed throughout, and, if not, when they are to be enforced. In particular, enforcing the `@nonNull` restriction without exception can make the creation of new objects tricky to impossible. Therefore, one usually wants to accept that restrictions are temporarily violated. As the SKilL type system can be seen as a contract between the programs that successively process a file, SKilL bindings check restrictions when a file is written or read.[3] Thereby, every program gets an input that fulfills the contract and has to produce an output that does as well, but is not constrained in its operation.

In an interactive editor, more frequent checks are desirable. The user should be immediately notified when a restriction is violated. This allows them to fix the problem before they turn their attention to other matters. The Scala binding supports such early checks by exporting a method that checks whether a value fulfills a restriction in its reflective API.

Restrictions that modify the serialisation format are dealt with by the binding.

2.2 Non-Serialised Features

The SKilL specification includes a lot of features not addressed above, viz. transient (auto) fields, hints, typedefs, views, enumerations, interfaces, and customisations. Those appear in specification files and affect the generated bindings, but they are not part of the serialised data, therefore we will never encounter them.

Enumerated types are a special case, here, because they are translated to the serialised type system systematically in a way that would allow their detection and recovery: an enumerated type with name T with values with names $A, B, C, \ldots$ is realised as an abstract user type with name T with singleton sub-types named $TA, TB, TC, \ldots$. This can be easily detected,

[3]The check after reading is necessary because the specification of the reading program can introduce new restrictions, which the previous program did, of course, not validate.

but it is dubious whether we would gain anything from doing this. The intended semantics, that the value of an enumeration field is one of the enumerated values, is already covered by the translated representation. As far as the editor is concerned, changing the value of an enumeration field amounts to the selection of an object, which is covered anyway.

2.3 SKilL Data

We said initially, that a SKilL file stores user data objects and their types. After having described the type system, we now turn our attention to the objects. From the user's point of view, objects are often parts of complex data structures, like trees. This is explicitly expressed by linking objects through reference fields. It is, however, entirely up to the user which structures are built from the links and how they are traversed. This is necessarily so, because the conditions that describe the intended data structure—like 'must not form cycles' for trees—can not be expressed in the type system. From the perspective of SKilL the organisation of objects is, therefore, very simple: every user type has a flat list of instances. A program can iterate these lists and start following references from a suitable start object.

The editor is, of course, also unaware of the intended organisation of the data. Like SKilL itself, it can only show lists of instances and follow references.

2.3.1 Characteristics of the Example Data

An editor for SKilL files should be able to cope with all valid files. But it's the common cases that especially should be handled well. We therefore inspect the properties of the SKilL files that were generated in other projects at ISTE. Many subsequent decisions are based on the observations made here; if reference fields were rare compared to primitive type fields, or if the references would mostly form trees, we would have taken different turns at some points.

To get an idea of actual data, we collected statistical data from 32 comparatively small files containing a total of $4{,}5 \cdot 10^6$ objects. The analysed files are just those of the provided samples that have a file size between 2 MB and 3 MB. The reason is this: the example data was generated from different source data sets by different tools. Therefore, we want to include a fair number of cases in our analysis. The program we wrote to gather the statistics is kept very simple and does not cope well with large data sets. Consequently, the larger the files, the smaller the number of files it can handle. The chosen set of file is just a range delimited by round numbers that can be analysed with reasonable space and time requirements. We select files that have roughly the same size, because larger files also have a larger impact on the numbers, and would thereby hide the data gathered from smaller files.

Our first question is, how many fields a user type typically has and how they are distributed among the field types. In order to summarise the counts of all objects in our sample, we

	Average	Quartiles				
		Q_0	Q_1	Q_2	Q_3	Q_4
Reference fields per object	4.4	0	3	4	6	12
Primitive fields per object	4.6	0	2	6	7	15
Compound fields per object	0.43	0	0	0	0	12
Total fields per object	9.5	1	5	11	13	30

Table 2.1: Number of fields per object for different classes of field types (4,5 $\cdot$ 10^6 objects).

	Average	Quartiles				
		Q_0	Q_1	Q_2	Q_3	Q_4
Size of compound field	1.2	0	0	0	0	4249
Size of non-empty compound field	5.8	1	1	2	3	4249

Table 2.2: Number of elements per compound field (1,9 $\cdot$ 10^6 fields, 0,41 $\cdot$ 10^6 of them non-empty).

calculate the average numbers and, because the average tells nothing about the distribution, quartiles. We use Q_0 and Q_4 to denote the minimum and maximum counts.[4] The result, is given in table 2.1. We find, that, although the number of fields per object is not limited, 75 % of the objects have a relatively modest size of 13 fields or less. But we must be prepared to handle objects with 30 fields. Reference types and primitive types are roughly equally frequent and deserve equal attention. Compound fields, however, are comparatively rare.

There may be only few compound fields, but each of them can store an arbitrary number of elements. A statistic of their actual sizes is shown in table 2.2: more than three quarters of the containers are empty, and more than three quarters of the non-empty ones are small and have three or fewer elements. At the upper end, however, compound types can contain thousands of elements. Thus, we must, on the one hand, be prepared to handle very large containers. But, on the other hand, the majority of them is very small and may merit a special treatment.

An empty container is just the default value of a compound field. Given their frequency, one wonders, how common default values are for other field types. The result of counting fields with default values, i.e. null, zero, the empty string, an empty container, or false, and non-default fields separately is shown in table 2.3. We find, that about half of the fields have default values. A field with a default value has probably never been written. Their high frequency, and the inspection of some specific cases, indicates that this, indeed, is what happens. The reason

[4]Note that the quartiles are calculated for each count independently. Thus, for example, in table 2.1, the object with the least number of reference type fields has zero reference type fields but may have other fields. In fact, we can tell, that it has: there is no object with zero fields total.

	Average	Quartiles				
		Q_0	Q_1	Q_2	Q_3	Q_4
Null reference fields per object	1.3	0	0	1	2	10
Non-null reference fields per object	3.1	0	2	3	4	10
Default primitive fields per object	3.0	0	1	5	7	24
Non-default primitive fields per object	1.6	0	1	2	2	8
Empty compound fields per object	0.34	0	0	0	0	11
Non-empty compound fields per object	0.091	0	0	0	0	7
Total default fields per object	4.7	0	1	5	7	24
Total non-default fields per object	4.8	0	3	5	6	21

Table 2.3: Number of default and non-default fields per object for different classes of field type (4,5 $\cdot$ 10^6 objects).

	Average	Quartiles				
		Q_0	Q_1	Q_2	Q_3	Q_4
Fields total	41	2	15	31	67	168
Non-default fields	20	0	10	15	29	105

Table 2.4: Number distinct fields reachable from an object by following one reference (4,5 $\cdot$ 10^6 objects).

for this is, that the inspected data was generated by programs which in parts use different fields, but still share a specification. Fields that were never written to bear little information, so displaying them will be of little use to the user. Displaying many of them will even do harm, for makes the fields that are actually in use harder to find. It, thus, should be possible to filter out default fields.

We said in the introduction, that objects can not be comprehended in isolation. Thus, we conclude this section by examining the fields of the next objects. To this end, we count the fields of each object together with those of the objects that it references. We remove duplicates that arise when the same object is referenced by two fields or when the object stores a reference to itself. The results are given in table 2.4. The average counts are in line with what one would expect from the number of fields per object and the number of non-null reference fields. We can take this as a reassurance that nothing weird is happening in the usage of references that would need investigation. This means, however, that we encounter a huge number of fields and can not show all of them.

3 Showing objects and drawing graphs

The first task of an editor is to display the contents of a file. For SKilL files, this means displaying the type system and displaying objects. We consider objects first.

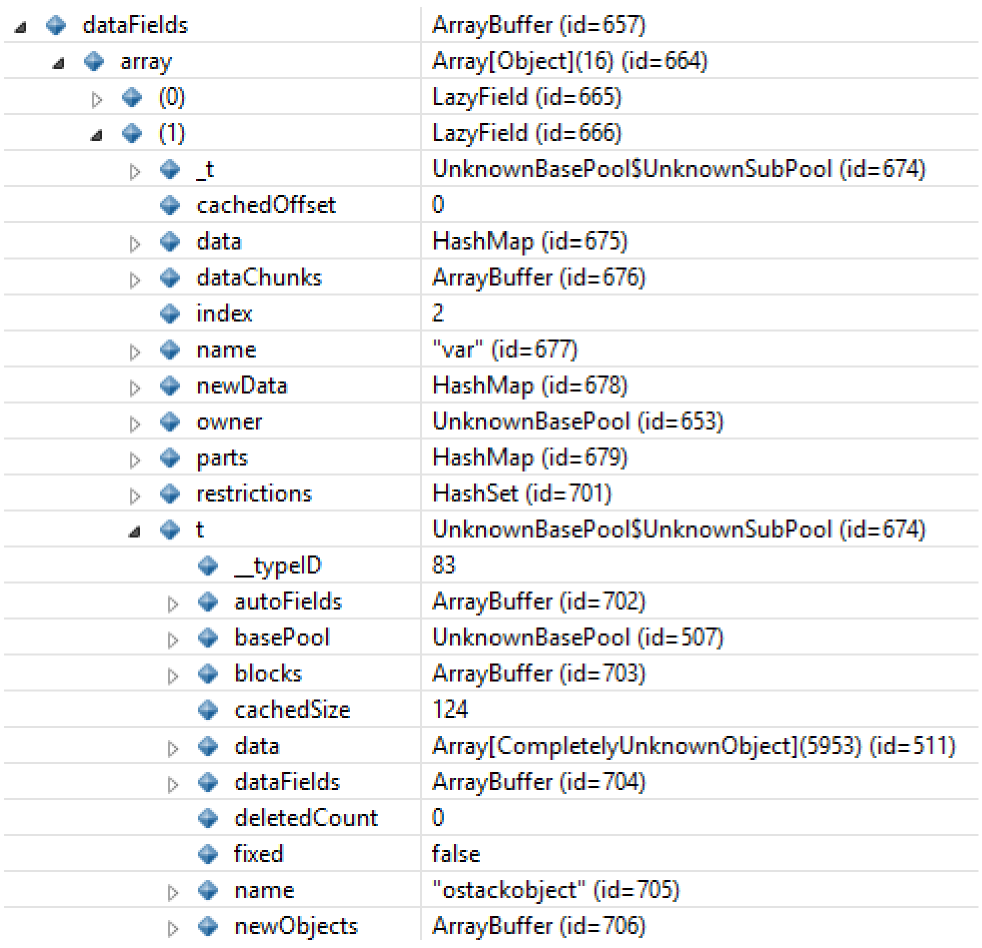
Objects have a data type, a SKilL ID[1], and fields. Data types have a meaningful name and the ID is an integer number, so those are straightforward to display. Displaying fields means displaying their names and values. It is reasonably clear how to do that for the values of primitive types and collections thereof. The main concern of this chapter is how to handle references. On an implementation level, user type references just store the SKilL ID of the referenced object; annotations additionally store the type. It is not sufficient to display the ID, because it is meaningless to the user, and tells them little except that the object exists. Thus, we are back to showing the whole referenced object, including its fields. We can not do so indiscriminately, however, for the number of fields that are reachable when we transitively follow references is usually far too high: as seen in section 2.3.1, for a quarter of the nodes, we find more than 60 distinct fields when following references once.

The object inspector functions of debugging tools, which face the same problem, usually solve this by allowing users to interactively show or hide the fields of any object according to their needs. A screen-shot of a typical specimen, the debugger of the Eclipse IDE, is shown in figure 3.1. This approach gets the job done and has the advantage that it is well-known to users. We will, therefore, adopt it in chapter 5 when we come to modifying objects. There are, however, some issues with this representation, which we want to improve upon.

First, the binary choice—to show all fields or none of them—can lead to a high number of visible fields that the user is not actually interested in. Especially after following a chain of references, all fields of the intermediate objects are visible. This makes it hard to locate the relevant information on the screen and quickly exceeds the available space. We rather want to show a short summary, which contains only relevant fields. Therefore, we add an option to hide selected fields completely and support keeping a remote object visible without showing all fields of the intermediate objects. As it would be rather pointless to hide the fields for every object separately, the choice is applied to all objects that have the field.

Which brings us to the second issue, namely that one always has to start following references ab initio. For every object, one has to 'click through' to the relevant fields again. This is especially

[1]But see section 5.2.2 for newly created objects



dataFields	ArrayBuffer (id=657)
array	Array[Object](16) (id=664)
(0)	LazyField (id=665)
(1)	LazyField (id=666)
_t	UnknownBasePool$UnknownSubPool (id=674)
cachedOffset	0
data	HashMap (id=675)
dataChunks	ArrayBuffer (id=676)
index	2
name	"var" (id=677)
newData	HashMap (id=678)
owner	UnknownBasePool (id=653)
parts	HashMap (id=679)
restrictions	HashSet (id=701)
t	UnknownBasePool$UnknownSubPool (id=674)
__typeID	83
autoFields	ArrayBuffer (id=702)
basePool	UnknownBasePool (id=507)
blocks	ArrayBuffer (id=703)
cachedSize	124
data	Array[CompletelyUnknownObject](5953) (id=511)
dataFields	ArrayBuffer (id=704)
deletedCount	0
fixed	false
name	"ostackobject" (id=705)
newObjects	ArrayBuffer (id=706)

Figure 3.1: Object representation in the Eclipse debugger (the example is taken from the reflective SKilL API for Scala). The objects dataFields, dataFields.array, dataFields.array(1), and dataFields.array(1).t are expanded. The user might be more interested in the name fields of array entry (0) than in most of the visible values.

a problem when looking through a list of objects; which is just what we have to begin with: the list of all objects of a selected type. It is usually a reasonable assumption, that the user wants to see the same fields in all comparable objects. In other words, the short summary should already contain all relevant fields. We, therefore, store, for each type, the sequences of fields that lead to the objects that are fully shown. Unless barred by null references, the corresponding referenced objects can then be shown for every start object.

The next point to consider is the frequent occurrence of cycles in the reference graph of SKilL files. A cycle means that more than one of the visible fields hold a reference to the same object. This should be easily detectable, so that the user does not have to inspect the same object twice. It would also be nice, if the linked data structures that exist in the file could be made explicitly visible. Thus, we deviate from the common hierarchical tree view and switch to a largely unconstrained node-link diagram. We say largely unconstrained because we need some constraints to prevent the diagram from changing its layout erratically.

What we aim for is a representation that summarises the relevant information about an object, including its relation to other objects and the values of selected fields, without cluttering the output with irrelevant information. While the decision what is relevant has to lie with the user, it should be carried over from one object to the next, when possible, so that objects can be easily compared. For comparing objects, we also need stable layouts, where the fields shown for two objects of the same type occupy the similar positions in the output.

In this chapter, we first describe in section 3.1 how a graph, in the abstract sense, is constructed from SKilL data. Then, we cover in section 3.2 how the user can select which part of that graph is visible, and how this selection is generalised to other objects. Finally, in section 3.3, we give the force directed placement algorithm that is used to lay out the node-link diagram of the visible part. The chapter then concludes in section 3.4with the results obtained with this procedure.

There are some topics regarding the layout and style of the link-node diagram that we can quickly dismiss here.

First is the idea of creating a layout for the whole file and using a zoom function to show the surroundings of the currently inspected object. This layout would not only take prohibitively long to create: Hu [Hu05] gives minutes for rendering graphs with 10'000 nodes, big skill files have millions. It also would run into problems with nodes that are referenced a lot. Obviously no layout can place a highly connected node next to all its neighbours.

Second one may wish to see which objects hold references to an inspected current object. But this not only would cause trouble with the layout for nodes that are referenced frequently but also require a full scan of all objects that could, qua type, store a reference. We rather aim to keep the application responsive and perform exhaustive searches only when the user explicitly requests them. Incoming edges are therefore not usually displayed.

Third, one may think of flow diagrams, entity relationship diagrams, or the like, and think about using different styles or decorations—like rectangular nodes, round nodes, diamond nodes, and solid line, dotted lines, crow's feet if you like—to keep edges and nodes of different kinds apart. However, we easily have 100 distinct user types, the number of fields being considerably higher. There are just not enough different shapes that are easily distinguished. Further, the fields and types already have names, which the user knows, unlike the putative shapes and styles which they would first have to remember. Shapes are, therefore, only used sparingly for few dedicated purposes, and not to distinguish user types.

3.1 Graph Representation of SKilL Data

As the first step towards displaying a node-link diagram, we define the underlying graph. The basic idea is straightforward:

- Objects are represented by nodes and reference fields are represented by directed edges.
- Nodes are labeled with the type and ID of the object they symbolise.
- Edges point from the object that contains the field to the object the field refers to and are labeled with the field name.

We also have handle primitive and compound types, which involves many special cases. Therefore, we give a detailed definition of the graph we construct from a SKilL file. To keep memory usage low, this graph is never built as a whole. Instead, the parts of it that are needed for displaying an object and its neighbourhood are instantiated on demand.

A labeled digraph $G = (V, E, s, t, \Sigma_V, \Sigma_E, l_V, l_E)$ is a tuple consisting of a set V of nodes, a set E of directed edges, two total functions $s, t \in E \to V$ that map an edge $e \in E$ to the ordered pair $(s(e), t(e))$ of the nodes which it connects, two sets Σ_V and Σ_E of labels, and two total functions $l_V \in V \to \Sigma_V, l_E \in E \to \Sigma_E$ which assign a label to every node and edge. We often have to take care that nodes and edges are distinct. The labels are elements of the user interface which can be rendered on screen and react to user interactions. They are described informally.

The simple case involving SKilL objects and reference types is displayed as shown in figure 3.2. This is handled by the following two definitions, where $o.f$ denotes the value of field f in object o:

Definition 1 (Objects)**.** If o is a SKilL object, then:

- $o \in V$ and
- $l_V(o)$ displays the type and ID of o.

Definition 2 (References)**.** If o is a SKilL object and f is a reference field in o with $o.f \neq null$, then:

- $(o, f) \in E$,
- $s((o, f)) = o$,
- $t((o, f)) = o.f$, and
- $l_E((o, f))$ contains the name of f.

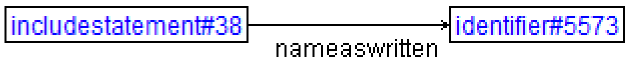


Figure 3.2: Representation of objects and reference fields. Field `nameaswritten` in object `includestatement#38` refers to object `identifier#5573`.

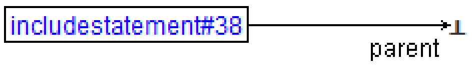


Figure 3.3: Representation of null references. Field `parent` in object `includestatement#38` is `null`.

The next question is how to deal with null references. We found in section 2.3.1, that half of the objects have null references. Therefore, a single null node, which all null references points to, would be highly connected and hard to place nicely in the layout. We prefer introducing more nodes over tangled edges and represent all null values by distinct nodes. The final representation on screen is shown in figure 3.3. In order to keep null nodes distinct, we use a constructor function $null(\cdot, \cdot)$ which is meant to yield a new entity which is not a SKilL object, nor a field, nor a value of another constructor.

Definition 3 (Nulls)**.** If o is a SKilL object and f a reference field in o with $o.f = null$, then:

- $(o, f) \in E$,
- $null(o, f) \in V$,
- $s((o, f)) = o$,
- $t((o, f)) = null(o, f)$,
- $l_E((o, f))$ contains the name of f, and
- $l_V(null(o, f))$ is a small bottom sign.

Concerning the primitive type fields (string, boolean, and numbers), we pursue two different lanes. Either we display their value in the label of the node that represents the object that contains them, or we render them as separate nodes. We didn't find a reason to prefer one way over the other, so the user can select the preferred representation for every field. Figure 3.4 shows an example of both representations. When the value is shown in a node, like for null references, two fields with the same value point to separate nodes.

Definition 4 (Values)**.** If o is a SKilL object and f is a boolean, integer, float, or string field in o, then:

- $value(o, f) \in V$,
- $(o, f) \in E$,

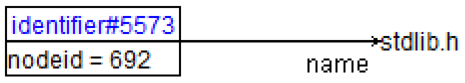


Figure 3.4: Representation of primitive type values. Field `nodeid` in `identifier#5573` is shown inside the node of the identifier object, field `name` as separate node.

- $s((o, f)) = o$,
- $t((o, f)) = value(o, f)$,
- $l_E((o, f))$ contains the name of f, and
- $l_V(value(o, f))$ shows the value $o.f$ as text.

As it is impossible to find a good position for huge nodes when the graph is laid out, the labels of string nodes leave out the middle part of long strings. The full value is included as mouse over text.

Next consider sets. Sets of references can be viewed as many-to-many-relations. Their natural representation would be a set of edges from the object containing the set to the members of the set. This would work for small sets, but would leave no trace of empty ones and overflow the screen for huge sets. We, therefore, first add a node that represents the collection itself. For small sets, we can then add links from the collection to the elements, and for large sets we can at least add a button that opens the list of all members. The node representing the collection is defined for all kinds of compound types—sets, lists, arrays, and maps—alike.

Definition 5 (Collections)**.** If o is a SKilL object and f a compound field in o, then:

- $compound(o, f) \in V$,
- $(o, f) \in E$,
- $s((o, f)) = o$,
- $t((o, f)) = compound(o, f)$,
- $l_E((o, f))$ contains the name of f, and
- $l_V(compound(o, f))$ shows the kind and size of $o.f$.

For small collections, we add edges and nodes that represent elements to the graph. Small means, that the number of elements is not above an adjustable threshold which defaults to five.

How exactly the elements are represented depends on the kind of collection as well as on the type of the elements. As above, we have to handle object references, null references, and

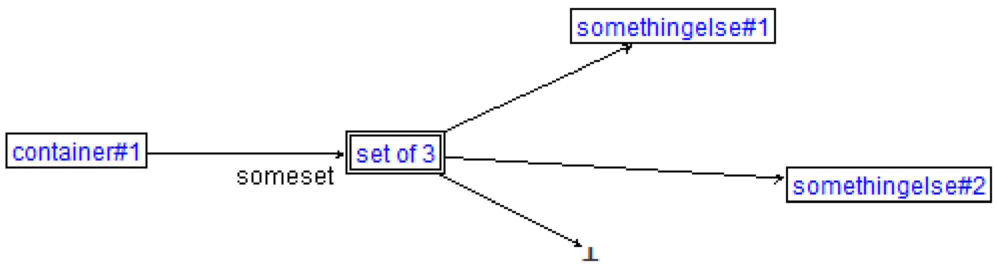


Figure 3.5: Representation of sets. Field `someset` in object `container#1` contains the three members, `somethingelse#1`, `somethingelse#2`, and `null`.

primitive values separately. At least, lists and arrays can be treated in the same way and nine cases suffice.

The members of sets are linked from the collection node by unlabeled edges. References, null references, and values of primitive type are treated analogously to simple fields. Figure 3.5 gives an example how a set of references is shown. The definition of the graph is tedious but simple:

Definition 6 (Members of sets, objects)**.** If o is a SKilL object, f is a field in o, $o.f$ is a set of references, $e \in o.f$ is a member of the set, $e \neq null$, and the size $|o.f|$ of the set is below a threshold, then:

- $member(o, f, e) \in E$,
- $s(member(o, f, e)) = compound(o, f)$,
- $t(member(o, f, e)) = e$, and
- $l_E(member(o, f, e))$ is blank.

Definition 7 (Members of sets, nulls)**.** If o is a SKilL object, f is a field in o, $o.f$ is a set of references, $e \in o.f$ is a member of the set, $e = null$, and the size $|o.f|$ of the set is below a threshold, then:

- $memberNull(o, f) \in V$,
- $member(o, f, e) \in E$,
- $s(member(o, f, e)) = compound(o, f)$,
- $t(member(o, f, e)) = memberNull(o, f)$,
- $l_E(member(o, f, e))$ is blank, and
- $l_V(memberNull(o, f))$ shows the bottom sign.

Definition 8 (Members of sets, values)**.** If o is a SKilL object, f is a field in o, $o.f$ is a set of primitive values, $e \in o.f$ is a member of the set, and the size $|o.f|$ of the set is below a threshold, then:

- $\mathit{memberValue}(o, f, e) \in V$,
- $\mathit{member}(o, f, e) \in E$,
- $s(\mathit{member}(o, f, e)) = \mathit{compound}(o, f)$,
- $t(\mathit{member}(o, f, e)) = \mathit{memberValue}(o, f, e)$,
- $l_E(\mathit{member}(o, f, e))$ is blank, and
- $l_V(\mathit{memberValue}(o, f, e))$ displays the value e.

For lists and arrays, we display the index of the element in the label of the edge. Besides this, they are very similar to sets, but we must take care in the definition of the graph that we keep members that have the same value distinct—unless they are objects.

Definition 9 (Members of lists and arrays, objects)**.** If o is a SKilL object, f is a field in o, $o.f$ is a list or array of references, $0 \leq i < |o.f|$ an index into the list, $o.f_i \neq \mathit{null}$ is the i-th element of the list, and the size $|o.f|$ of the set is below a threshold, then:

- $\mathit{member}(o, f, i) \in E$,
- $s(\mathit{member}(o, f, i)) = \mathit{compound}(o, f)$,
- $t(\mathit{member}(o, f, i)) = o.f_i$, and
- $l_E(\mathit{member}(o, f, i))$ shows the value of the index i.

Definition 10 (Members of lists and arrays, nulls)**.** If o is a SKilL object, f is a field in o, $o.f$ is a list or array of references, $0 \leq i < |o.f|$ an index into the list, $o.f_i = \mathit{null}$, and the size $|o.f|$ of the set is below a threshold, then:

- $\mathit{memberNull}(o, f, i) \in V$,
- $\mathit{member}(o, f, i) \in E$,
- $s(\mathit{member}(o, f, i)) = \mathit{compound}(o, f)$,
- $t(\mathit{member}(o, f, i)) = \mathit{memberNull}(o, f, i)$,
- $l_E(\mathit{member}(o, f, i))$ shows the value i, and
- $l_V(\mathit{memberNull}(o, f, i))$ shows the bottom sign.

Definition 11 (Members of lists and arrays, values)**.** If o is a SKilL object, f is a field in o, $o.f$ is a list or array of primitive values, $0 \leq i < |o.f|$ an index into the list, and the size $|o.f|$ of the set is below a threshold, then:

- $memberValue(o, f, i) \in V$,
- $member(o, f, i) \in E$,
- $s(member(o, f, i)) = compound(o, f)$,
- $t(member(o, f, i)) = memberValue(o, f, i)$,
- $l_E(member(o, f, i))$ shows the value i, and
- $l_V(memberValue(o, f, i))$ displays the value $o.f_i$.

Maps are a difficult topic, because they involve an unbounded number of ground types, and each of them can be a reference or a value type. Maps store partial functions with a finite domain. They can be considered to be sets of ordered pairs with the constraint that the first member of the pair, the key, is unique. The second member of the pair is the result of looking up the value in the map. If the map has more than two type parameters, the value is another map that takes care of the rest of the keys. Ordered pairs and sets are already dealt with in the previous definitions. Representing maps as sets of pairs would then lead to a graph as shown in figure 3.6.

We think, however, that this representation would consume too much screen space. Almost half of the nodes in figure 3.6 represent not actual data but auxiliary tuples and sub-maps. It would not make sense to build a representation which is never actually used because it overflows the screen. We therefore use a simplified representation, and only add edges from the collection node to the values of the last ground type. The sequences of keys that are mapped to these values are shown in the labels of the edges. We, thus, see nested maps not as function to functions, but as a function from a tuple to a value. For example $\{a \mapsto \{1 \mapsto 2, 3 \mapsto 4\}, b \mapsto \{5 \mapsto 6\}\}$ is read as $\{(a, 1) \mapsto 2, (a, 3) \mapsto 4, (b, 5) \mapsto 6\}$. This is displayed as shown in figure 3.6. Note that this transformation also allows us to speak of the size of the whole map.

The chosen representation has two serious shortcomings. First, keys which have reference types are not represented with an object node but only mentioned in the label of an edge by their type name and ID. Therefore, one can not reach the field of a key object in the graph. This is remedied by the fact that the editing function described in chapter 5 can dereference keys. The second problem is, that nested maps may contain inner empty functions, which we can not represent. For example, we can not show the c in the map $\{a \mapsto \{1 \mapsto 2, 3 \mapsto 4\}, b \mapsto \{5 \mapsto 6\}, c \mapsto \emptyset\}$. This can not be solved with null values, like $\{\ldots, (c, \bot) \mapsto \bot\}$, because null already is a legal value of reference types. We will come back to this issue in chapter 5.

We found no satisfying solution to the problem of representing maps generally. However, we did not encounter non-empty maps in the example data, thus it is unclear whether the maps that occur in practice are small enough to admit being shown as graph in the first place and whether empty functions are an issue. Therefore, we implement the compact but incomplete

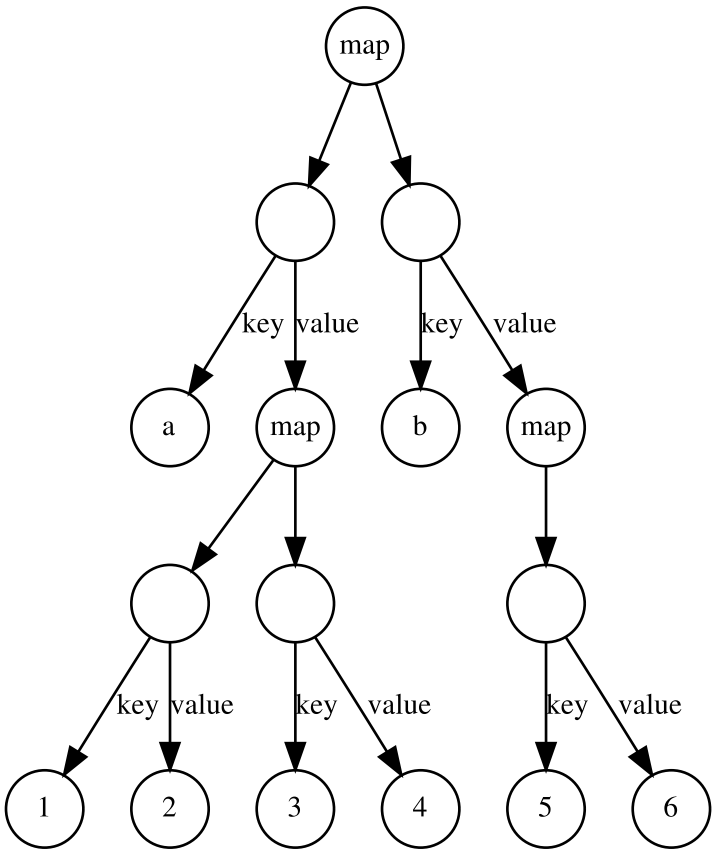


Figure 3.6: Example of a complete representation of the map $\{a \mapsto \{1 \mapsto 2, 3 \mapsto 4\}, b \mapsto \{5 \mapsto 6\}\}$.

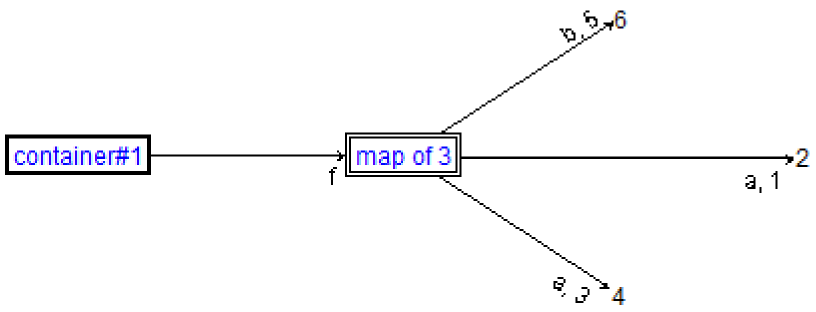


Figure 3.7: Example of the implemented representation of the map $\{a \mapsto \{1 \mapsto 2, 3 \mapsto 4\}, b \mapsto \{5 \mapsto 6\}\}$.

representation. The definitions, then, are very similar to the definition of lists, with indices replaced by sequences of keys:

Definition 12 (Members of maps, objects)**.** If o is a SKilL object, f is a field in o, $o.f$ is a map with type $\tau_1 \to \tau_2 \to \cdots \to \tau_n$ where τ_n is a reference type, $x_i : \tau_i$ for $1 \leq i < n$ are keys of the map, $o.f(x_1)\dots(x_{n-1})$ is defined and not $null$, and the size of the map is below a threshold, then:

- $member(o, f, x_1, \dots, x_{n-1}) \in E$,
- $s(member(o, f, x_1, \dots, x_{n-1})) = compound(o, f)$,
- $t(member(o, f, x_1, \dots, x_{n-1})) = o.f(x_1)\dots(x_{n-1})$, and
- $l_E(member(o, f, x_1, \dots, x_{n-1}))$ shows the values of the keys x_1 to x_{n-1}.

Definition 13 (Members of maps, nulls)**.** If o is a SKilL object, f is a field in o, $o.f$ is a map with type $\tau_1 \to \tau_2 \to \cdots \to \tau_n$ where τ_n is a reference type, $x_i : \tau_i$ for $1 \leq i < n$ are keys of the map, $o.f(x_1)\dots(x_{n-1})$ is defined and equal to $null$, and the size of the map is below a threshold, then:

- $memberNull(o, f, x_1, \dots, x_{n-1}) \in V$,
- $member(o, f, x_1, \dots, x_{n-1}) \in E$,
- $s(member(o, f, x_1, \dots, x_{n-1})) = compound(o, f)$,
- $t(member(o, f, x_1, \dots, x_{n-1})) = memberNull(o, f, x_1, \dots, x_{n-1})$,
- $l_E(member(o, f, i))$ shows the values of the keys x_1 to x_{n-1}, and
- $l_V(memberNull(o, f, x_1, \dots, x_{n-1}))$ shows the bottom sign.

Definition 14 (Members of maps, values)**.** If o is a SKilL object, f is a field in o, $o.f$ is a map with type $\tau_1 \to \tau_2 \to \cdots \to \tau_n$ where τ_n is a primitive type, $x_i : \tau_i$ for $1 \leq i < n$ are keys of the map, $o.f(x_1)\dots(x_{n-1})$ is defined, and the size of the map is below a threshold, then:

- $memberValue(o, f, x_1, \dots, x_{n-1}) \in V$,
- $member(o, f, x_1, \dots, x_{n-1}) \in E$,
- $s(member(o, f, x_1, \dots, x_{n-1})) = compound(o, f)$,
- $t(member(o, f, x_1, \dots, x_{n-1})) = memberValue(o, f, x_1, \dots, x_{n-1})$,
- $l_E(member(o, f, x_1, \dots, x_{n-1}))$ shows the values of the keys x_1 to x_{n-1}, and
- $l_V(memberValue(o, f, x_1, \dots, x_{n-1}))$ displays the value $o.f(x_1)\dots(x_{n-1})$.

Constant fields describe the type and not any individual object. They are, therefore, not included in the graph. We, thus, finish with:

Definition 15 (Closure)**.** V and E have no members except those from definitions 1 to 14

3.2 Exploring the Graph

The graph defined in the previous section is far too large to be shown at once. We now define, how the user can select which part of it is displayed at a given instance. Here we have to ensure, that the selection is done in a way that allows its generalisation to all objects of a type.

The first and simpler part of the selection procedure concerns what is not shown. We have seen, that objects can have a lot of fields. When all fields are shown, there is little space left for the fields of adjacent objects, which the user might be more interested in. We thus add, for every field, the option to remove it from the graph completely. This option applies to the field in general, that is, it is hidden in all objects of the type that contains the field. In combination with sub-typing, one might wish to treat a field differently depending on the sub-type of the object. We reject this idea because it would become hard to predict the behaviour of the program when the preference can be overridden repeatedly in the sub-type hierarchy. Thus, the user can only hide the field completely, but will face no surprises.

We also noticed in section 2.3.1, that half of the fields have default values, and are thus obvious candidates for hiding. The default behaviour of the editor is thus to hide all default values. This is often valid treatment of defaults, even if the field has non-default values in other objects. For example, boolean fields, are often used to indicate that an object is unusual in some respect, like `isSpecial`. When the field is false, the object is just a normal specimen of its class, which the user would already assume by default. Therefor it is not very informative to display `isSpecial = false`. This does, however, not work for all fields—the Royal Observatory in Greenwich *has* a longitude, it just happens to be 0.0—, and can, therefore, be disabled on a per field basis.

Those preferences—whether a field is hidden completely, hidden if default, or always visible—are available in the type viewer described in chapter 4. They can also be accessed directly through a pop-up menu from the object nodes in the graph view.

Having clarified what is hidden, we now turn to the second part: deciding what to show. Our approach is based on the assumption that the user is, at a given instant, mainly interested on one object. The adjacent objects are only interesting because they are referenced by the main one. This makes it possible to associate every visible object with the sequence of fields through which it was found. Those sequences enable us to transfer user choices from one main object to the next.

From the user's point of view, the selection of visible objects is very simple. Like in the tree views of object inspectors, the user can toggle every object node between two states: either it is expanded, and all its fields are shown—unless explicitly hidden—, or it is collapsed, and they are not shown. We indicate the current state of a node by drawing a box around expanded nodes but not around collapsed ones. We assume this to be the most frequent action and trigger it by clicks on nodes. When the user wants to shift their attention to another object entirely, they can select it as the new main object using a pop-up menu.

Internally, the toggling between collapsed and expanded state is handled by updating a set of paths to expanded objects. As an object can be reachable through multiple paths, we need to take care that the resulting behaviour matches the expectations of the user. In particular, when a user collapses a node, it must not stay expanded through a second path. When they expand an object that is reachable in multiple ways, we must decide which path or paths are probably meant.

In the description of the algorithm, we use the following conventions: F is the set of fields. A path $p = f_1 f_2 \cdots f_n \in F^*$ is a sequence of fields. When dealing with paths, we follow the usual conventions for free monoids and use ϵ to denote the empty path and juxtaposition for concatenation. A path can be considered a partial function from objects to values, or, more suitable to our present purpose, from nodes to nodes. A singleton path f maps an object node v to the target of the edge (v, f) that represents f and starts a v: $f(v) = t((v, f))$ if $(v, f) \in E$. Longer paths fp follow edges recursively: $fp(v) = p(f(v))$. The empty path is the identity function $\epsilon(v) = v$.

The visible part of the graph is now defined as follows: given a main object v_0 and a set $\mathcal{X}$ of paths to expanded nodes, we first determine the set X of expanded nodes as the set of nodes reachable from v_0 via the paths in $\mathcal{X}$:[2]

$$X = \{v \mid \exists p \in \mathcal{X} : v = p(v_0)\}$$

The visible nodes are, in a first take, the expanded nodes and their successors. But, it is possible to have a distant node expanded without expanding all intermediate nodes. In order to keep the graph connected, nodes that lie on the paths to expanded nodes must also be visible. The set $\mathcal{V}$ of visible nodes is, thus:

$$\begin{aligned}\mathcal{V} = {} & X \\ & \cup \{v \mid \exists w \in X, e \in E : s(e) = w \wedge t(e) = v\} \\ & \cup \{v \mid \exists pq \in \mathcal{X} : pq(v_0) \in X \wedge v = p(v_0)\}\end{aligned}$$

Finally, we have to decide which edges become visible. Initially, we showed all edges that connect visible nodes, but were not satisfies with the results. The edges between visible nodes include fields of collapsed nodes when their target is already visible. They consume space on the screen and often have adverse effects on the produced layout, even though the user did not request to see them. What is worse, they can not be easily hidden in order to free space.

[2] It is clear, that $p(v)$ can only be defined when the type of v or one of its super types defines first field in p. Thus, the implementation speeds up access to $\mathcal{X}$ by partitioning it according to the type that the first field belongs to. This is not spelled out, here, because we consider it to complicate the description unnecessarily.

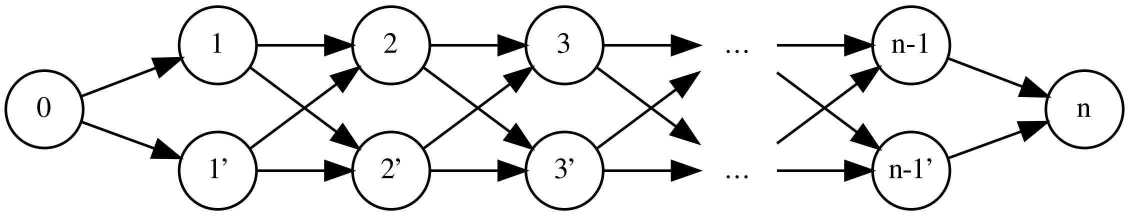


Figure 3.8: Example of a graph in which the number of different shortest paths between two nodes grows exponentially with the number of nodes.

Therefore, edges are only shown when they represent fields of expanded nodes or lie on a path to an expanded node. This concludes the determination of the visible part of the graph.

Expanding or collapsing a node is done by updating the set $\mathcal{X}$ of expanded paths. When a node v is collapsed, all paths p that cause a node to be expanded, i.e. $v = p(v_0)$, are removed from $\mathcal{X}$. For expanding, a path has to be added. Due to cycles, a node can be reachable through many paths, and we have to decide which one, or which subset of them, to use. We do not let the user select a path, because expanding a node is a very common operation which should be simple: the user wants to expand a particular node and gets it expanded without further interaction. The generalisation to other objects is handled automatically by the editor.

A modest generalisation seems to be to pick all shortest paths to the node. This will not work, however: consider the graph shown in figure 3.8. It has $2n$ nodes, but there are 2^{n-1} different shortest paths from node 0 to node n: for the first $n-1$ edges of those paths, one can freely choose a successor from the top or bottom row of nodes. This is, by the way, another reason not to let the user select a path. Although the set of visibly nodes is probably always small enough to keep 2^n in a manageable range, we still want to avoid exponential growth.

This is easily achieved by never adding more than a small constant number of paths to $\mathcal{X}$ at once. The size of $\mathcal{X}$ is then bounded linearily by the number of user actions, which we accept because an editor is usually closed after a session. The only negative consequence of this safety measure is, that a node that the user expects to be expanded is shown collapsed.

We should now discuss a topic which we have passed over so far, namely the members of compound types. We defined paths as sequences of fields, hence they can not identify a member of a compound type. This could be handled by extending the definition to include indices for lists, keys for maps, and so on. We will, however, not do that: paths are used, because we assume that, when the user expands a field, they also want to see it expanded elsewhere. We consider it unlikely that this assumption holds for members of compound types. A use case where one always wants to see, say, the third element of a list is, in our opinion, too uncommon to justify the extension. Members of containers and their successors are, therefore, handled individually. The member node itself is added to a set of expanded nodes, which is kept locally inside the viewer window.

3.3 Graph Layout

There are many solutions to the problem of turning an abstract graph into a node-link diagram, or ‘laying out the graph’, of which Di Battista et al. [DBETT94] give an overview. As we have to display arbitrary user data, we do not consider algorithms which require the graph to have certain properties, like planarity, a limited number of edges per node, or the absence of directed cycles. The algorithms they list for laying out arbitrary, unconstrained graphs are the force directed placement algorithms, also called spring embedders, which simulate the layout as a physical system where a distance-dependant ‘repulsive force’ spreads the nodes evenly and an attractive force pulls connected nodes together. We will now review the common layout algorithms and the goals they pursue, discuss which one can be adapted to meet our additional stability goal, and then present our implementation of the layout algorithm.

3.3.1 Existing Force Directed Placement Algorithms

To state the layout task more formally: given a digraph $(V, E, s, t, \Sigma_V, \Sigma_E, l_V, l_E)$ as defined in the previous chapter and a rectangular frame $C = [0 \ldots \mathit{width}] \times [0 \ldots \mathit{height}]$, find a placement $\vec{x} : V \mapsto C$ that assigns to every node a position on the frame and that is ‘aesthetically-pleasing’, or, to be more explicit, meets a set of goals, which we quote from Fruchterman and Reingold [FR91]:

- Distribute the vertices evenly in the frame.
- Minimize edge crossings.
- Make edge lengths uniform.[3]
- Reflect inherent symmetry.
- Conform to the frame.

Those goals usually can not be met at once. A cycle graph C_n, for example, should be laid out with the nodes on a circle as far as symmetry is concerned, but should curl up in order to achieve a uniform distribution of vertices. Therefore, a compromise between the different goals has to be found. In order to get an objective measure for the ‘overall layout quality’, one proceeds to quantify how well each of the sub-goals are met and specify cost functions that translate the badness according to each criterion into a ‘common currency’ that allows specifying how much, say, pixels standard deviation of the edge length distribution one is willing to trade off for removing one edge crossing.

[3]Though Gansner et al. argue that edge length should depend on the number of adjacent edges, for otherwise the layout becomes too crowded in the vicinity of high degree nodes [GKN04]. This should be seen as a refinement of this goal, not a refutation, for every edge should still have its, now variable, ideal length.

Symmetry and edge crossings are usually not explicitly optimised for: symmetry is hard to quantify[4], and edge crossings would complicate the algorithm, where the results are usually good enough without explicitly optimising for it.

The two criteria that are addressed are then uniform distribution and of nodes and uniform edge length. The first is expressed by a cost function that penalises pairs of unconnected vertices which are located close together, and the second by a function that penalises deviations from a desired edge length[5]. Both are functions of the distance of a pair of nodes, and their gradient tells us whether the distance between the nodes should increase or decrease in order to improve the quality of the layout. This can be likened to a physical system where the penalty function is viewed as potential energy and its gradient is a force, hence the name force directed placement. A layout algorithm can specify the energies and search for a position where the energy is minimal. Alternatively, one can specify the forces and search for an equilibrium, i.e. a position where all forces acting on a vertex cancel each other out. We consider examples of both cases, below.

In the following discussion of the algorithms, we use $v, w \in V$ to denote two nodes of the graph, $\vec{\delta}_{vw} = \vec{x}(w) - \vec{x}(v)$ their distance on the canvas with magnitude $\delta_{vw} = |\vec{\delta}_{vw}|$ and direction $\hat{\delta}_{vw} = \frac{\vec{\delta}_{vw}}{\delta_{vw}}$. Two vertices v and w are connected when there is an edge between them in either direction: we abbreviate this as $v \leftrightarrow w$ iff $\exists e \in E : \{v, w\} = \{s(e), t(e)\}$. We write $\vec{F}(v, \ldots)$ to denote forces exerted *on* node v *by* some other entity that takes the place of the ellipsis.

Eades [Ead84] gives the goal of the optimisation as a pair of force functions, viz. a logarithmic attractive force

$$\vec{F}_A(v, w) = \hat{\delta}_{vw} c_1 \log \frac{\delta_{vw}}{c_2}$$

between connected nodes and inverse square repelling force

$$\vec{F}_R(v, w) = -\hat{\delta}_{vw} \frac{c_3}{\delta_{vw}^2}$$

between unconnected nodes. The constant c_2 specifies the desired distance of adjacent nodes, the ratio of c_1 and c_3 gives the trade-off between uniform edge length and separation of nodes. The absolute values of c_1 and c_3 only scale the forces and do not affect the layout; this can be relevant for the algorithm that searches the minimum.

[4]Hard can be read in two ways, here: it might be hard to define, for, to the best of our knowledge, no one attempted to quantify how well a graph reflects symmetry. But this may be attributed to the fact that a definition would probably involve comparing the placement of isomorphic subgraphs. Graph isomorphism is not known to be solvable in polynomial time [GJ79], and, thus, the quantity would be hard to compute, anyway.

[5]To achieve truly uniform distribution, the desired edge length must equal the distance between close unconnected nodes. Fruchterman and Reingold [FR91] use $C\sqrt{\frac{\text{area}}{\text{number of nodes}}}$ as desired edge length for some experimentally determined constant C. We can, unfortunately, see that the constant should actually depend on the data when we consider graphs that form regular rectangular and triangular grids: the latter is more densely packed, therefore the edges would need to be longer in order to fill the available space. However, the uniform distribution is usually not the first priority.

Fruchterman and Reingold [FR91]reject the logarithmic force function for its computational cost. They instead use the force functions

$$\vec{F}_R(v,w) = -\hat{\delta}_{vw}\frac{k^2}{\delta_{vw}}$$

between all pairs of vertices and additionally

$$\vec{F}_A(v,w) = \hat{\delta}_{vw}\frac{\delta_{vw}}{k}$$

between connected ones. Here, k is the optimum edge length; there is no separate constant to specify the trade-off between the forces, because the relative weight of the forces determines the edge length.[6]

The algorithm of Kamada and Kawai [KK89] uses a different approach and tries to find a layout where the distance of two nodes v and w on the frame is proportional to their shortest path distance d_{vw} in the graph. If unconnected graphs are laid out, some sensible default value has to be supplied for nodes from different components. Their algorithm minimises the energy

$$\sum_{v\neq w}\frac{1}{d_{vw}^2}(\delta_{vw} - Ld_{vw})^2$$

which penalises, for each pair of nodes, the deviation from the desired edge length L. This is equivalent to a force

$$\vec{F}(v,w) = \hat{\delta}_{vw}\frac{1}{d_{vw}}(\delta_{vw} - Ld_{vw})$$

on vertex v by vertex w.

Most recent work is based upon the algorithms of Kamada and Kawai and Fruchterman and Reingold—practitioners will know them from the `neato` and `fdp` layout engines of the software tool Graphviz [GN00]. However, we found during initial experiments, that they determine the placement of nodes too firmly for our purposes. In section 3.3.4, we add a further force to keep fields in a stable position relative to their containing object. This appears to work better if the basic layout leaves some slack. Therefore, we adopt the quickly decaying inverse square repulsive and slowly increasing logarithmic attractive forces from Eades. We do not consider the calculation of the logarithm a problem for the small graphs we are concerned with.

3.3.2 Overlap Removal

The algorithms given so far only cover the problem of placing point nodes. When one wants to show node labels, the nodes may be too close together and the labels will overlap. The overlap

[6]The constant C used by Hu [Hu05] can be seen to simultaneously change the edge length and scale the force

can either be removed in a separate step or by taking the extent of the label into account when calculating the forces during node placement.

The simplest approach to overlap removal is to change the scale of the layout. This is not an option, because we only have a fixed area of screen space available. Other approaches to remove overlaps post hoc distort the layout. As we want to keep the relative positions of certain nodes fixed, we would have to impose additional constraints on the node movements allowed for during overlap removal; this seems to complicate matters unnecessarily.

Therefore, the extent of the node is taken into account during the initial placement. Following one of the approaches explored by Harel and Koren [HK02], the forces are calculated based on length of the line segment connecting the centres of two nodes that is not obstructed by their labels. This will not guarantee that nodes do not overlap—consider a high node and a wide node intersecting at their ends—, but is usually sufficient to ensure that the labels of both nodes are legible. As noted there, considering node extent leads to slower convergence of the search for an equilibrium placement and prevents the layout from overcoming local minima: due to their extent, the nodes can not get past each other any more. We, therefore, also adopt their solution of starting to factor in the extent of nodes only after an initial layout was found.

3.3.3 Conforming to the Frame

In order to be visible, the nodes must be placed inside the available frame. Confining nodes to the frame can distort the placement badly: for example, a chain has to curl up when it is longer than the diagonal. For a clear layout, it is best when the frame is not a limiting factor. Fruchterman and Reingold [FR91] indicate that they often choose the edge length ‘so that the resulting graph is small enough that it never nears the borders’ and scale the graph afterwards. Kamada and Kawai [KK89] suggest using a suitable ideal edge length in order to ensure that the layout fits inside the frame. In our application, with labeled nodes and edges, there is, however, not much room for scaling and changing edge lengths. The distortions caused by the frame are, therefore, accepted.

In the literature, we found essentially two ways to confine the nodes to the frame. Fruchterman and Reingold [FR91] consider the borders of the frame to be solid walls that absorb all forces orthogonal to them and thus prevent nodes from passing through. For a rectangular frame that is aligned with the coordinate system, this amounts to clamping the component of the position vectors to the ranges $0 \ldots width$ and $0 \ldots height$ respectively. The alternative approach uses a long range repelling force between the borders and the nodes. A naïve way to implement this is to place immovable dummy nodes at the circumference. This is, however, generally rejected because a high number of dummy nodes is needed to ensure confinement [FR91, DH96]. Davidson and Harel [DH96] instead add terms to their energy function, which repel a node from each of the four borders separately. In a rectangle aligned with the coordinate system, this is, again, computable component wise.

We adopted the solution of Davidson and Harel. Like in their implementation, the power of the distance in the frame function is the same as for the node-node repulsion. That is, with $x_v = (x, y)$ and basis vectors $e_x = (1, 0), e_y = (0, 1)$, the force of the frame on vertex v is:

$$F_{\mathit{Frame}}(v) = c_3 \left(e_x \frac{1}{x^2} - e_x \frac{1}{(\mathit{width} - x)^2} + e_y \frac{1}{y^2} - e_y \frac{1}{(\mathit{height} - y)^2} \right)$$

3.3.4 Enforcing Stability

We said initially, that graphs representing objects of the same type should look alike. This means that nodes representing objects that are referenced by the same field stay in the same position relative to the node representing the object which stores the reference. Fortunately, the layout produced by force directed placement algorithms can be easily adjusted by adding suitable forces.

A literal reading of the requirement to keep the same field in the same position would suggest using an ideal relative displacement between an object node and the value of the field. A force that causes the layout to maintain this displacement could, for example, be proportional to the difference between ideal and actual displacement. However, we consider this to be too restrictive. Due to inheritance, and the fact that we usually hide default values, the same field occurs in nodes with few and with many neighbours. Nodes, therefore, have to move away from their ideal position, otherwise parts of the layout would become too densely populated. In cases where the ideal displacement can not be maintained, we consider it more tolerable to change the distance than the direction. We, therefore, restrict the directions of edges representing fields and leave it to the forces of the basic algorithm to determine the length.

In order to determine the direction of an edge, we need to add a force that rotates an edge without influencing its length. Borrowing, again, from physics, we consider what happens when the ends of an inelastic bar are pulled in different directions: the components of the forces that are parallel to the bar are transmitted through it and cancel each other out. The components perpendicular to it accelerate its ends into opposite directions and effect a rotation that aligns the bar with the direction of the force. This leads directly to the following implementation: let $\vec{p}(e)$ denote the ideal direction of edge e and c_4 a constant that determines the relative importance of achieving the ideal direction. Then, for every edge e with distance $\vec{\delta}_e = \vec{\delta}_{s(e)t(e)}$ in the graph, add the forces

$$\vec{F}_p(t(e), e) = -\vec{F}_p(s(e), e) = c_4 \left(p(f) - \frac{\vec{\delta}_e(\vec{\delta}_e \vec{p}(e))}{\vec{\delta}_e^2} \right)$$

to the forces acting on the endpoints $s(e)$ and $t(e)$.

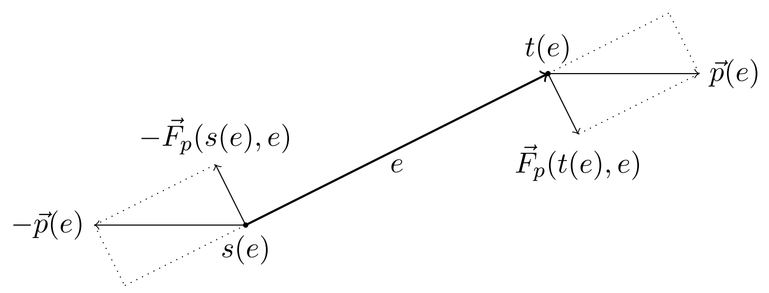


Figure 3.9: Forces $\vec{F}_p$ added in order to prefer layouts where the edge e is parallel to $\vec{p}(e)$ without influencing edge length.

This does not work correctly anymore as soon as the extent of nodes is taken into account. Consider a vertical edge: when this edge is rotated, the centres of the nodes at its ends keep their distance. But the labels of the nodes do not rotate and the unobstructed distance decreases. Therefore, the basic repulsive force starts to push their centres farther apart. This is usually no issue because most nodes are restrained in their position by multiple edges. But when multiple pending nodes are connected to the same node and compete for the same direction, the direction forces on them are not aligned properly and do not cancel each other in an equilibrium. Instead, the pending nodes start to orbit each other. We did not find a solution to this for rectangular labels. However, we see no adverse consequences of this behaviour.

The direction preference $\vec{p}(e)$ of an edge e is meant to keep all edges representing the same field aligned. It therefore depends on the direction preference $\vec{p}(f)$ of the field f which is represented by $e = (o, f)$. Treating all edges that represent a field in the same way does, however, lead to problems with the layout. Two or more edges which represent the same field may point to the same target node. Their source nodes can, then, not be placed in a way that satisfies all goals at the same time. In order to achieve similar directions, the nodes have to be placed close together or the edges have to become longer. In the end none of the goals is satisfied. We found it preferable to relax the direction goal when such a conflict arises. The direction force on an edge $e = (o, f)$ is, therefore, divided by the number of conflicting edges.

This leaves the problem of how to determine the value of $\vec{p}(f)$ for all fields f. In some cases, like a field called `parent` in a type that represents nodes of a tree, the user has a strong opinion what $\vec{p}(f)$ should be, and is able to specify the direction of their choice. But for a stable layout, we need a direction preference for all fields. This, we can not ask the user to provide. Therefore, the directions are determined automatically.

It is not possible to assign the directions randomly. The graph contains cycles, and the directions of the edges in a cycle are mutually dependent: the last edge must return to the beginning. Direction preferences that are not compatible with the cycles in the data can obviously not be met. That we can not find suitable directions by inspecting the type system can be seen by considering binary trees and doubly linked lists. Both have two link fields that refer to another node of the same type. But, in the case of lists, they have to be represented by edges in opposite directions, while the links to the child nodes in a tree should be represented by roughly perpendicular edges.

We therefore have to derive the edge directions from the actual data. Considering the fact we have to deal with large files, we do not inspect the whole file before showing the first graph, but instead iteratively refine the ideal directions every time a graph is rendered: the directions start unconstrained (i.e. $\vec{p}(f) = (0, 0)$) and are updated to reflect the direction that the edges have in the produced layout. A, direction that the layout was able to respect becomes determined more strongly, while the ones that were overridden by the other forces adapt towards a direction that is possible to achieve.

As the direction that a field has in a layout, we use the average of the normalised distances of the edges that represent the field:

$$\vec{a}(f) = \frac{1}{|\{o.f | o.f \in E\}|} \sum_{o.f \in E} \frac{\vec{r}(o.f)}{|\vec{r}(o.f)|}$$

This is a unit vector if all edges are parallel and close to zero if their directions are unrelated. The ideal direction is then updated with exponential smoothing. Thus, the mutual dependencies of field directions that are present in any given layout contribute to the preferences but do not completely override the dependencies encountered before.

$$\vec{p}(f) \leftarrow c\vec{p}(f) + (1 - c)\vec{a}(f)$$

The constant c should be relatively close to 1 because the directions, once found, should remain relatively stable. But then, on the other hand, it takes a long time until strong direction preferences emerge. We implement a fast start behaviour by using $c = \min\{c_5, |p(f)|\}$. When $p(f)$ is weak, it quickly adapts to the directions found in the layout. Once it is strongly determined, it only changes slowly. Fields for which the user has defined a direction do not participate in this iterative process.

An example of this process is shown in figure 3.10. With edge direction unconstrained, a doubly linked tree is laid out as shown in part (a) with nodes evenly distributed but random edge directions. But, given the small size of the graph, the average directions $a(f)$ will be non-zero. In the example, the field rightoperand accidentally never points to the left. This unevenness in the distribution of edge directions gets the determination of the directions started. Because c is initially zero, the preferred directions are set to the average directions in (a). These are: $(-0.18, 0.29)$ for parent, $(0.61, -0.24)$ for rightoperand, and $(-0.18, -0.34)$ for leftoperand. When the same graph is rendered again, the tree structure clearly emerges in sub-figure (b). The bold lines at the node `arithmeticadd#2887` indicate the preferred directions inside a unit circle. The edges do not point into the preferred direction exactly, but this is expected. The tree is doubly linked, therefore every edge is subject to two direction preferences. Also, meeting all direction preferences would place the leftoperand of a rightoperand and the rightoperand of a leftoperand at the same location, which is prevented by the repulsive force. Repeatedly updating the edge directions and placing the nodes leads to drawing (c) after 10 iterations. The visual appearance has not changed greatly, but the edge direction preferences became stronger and can now transfer the achieved layout to other sub-graphs.

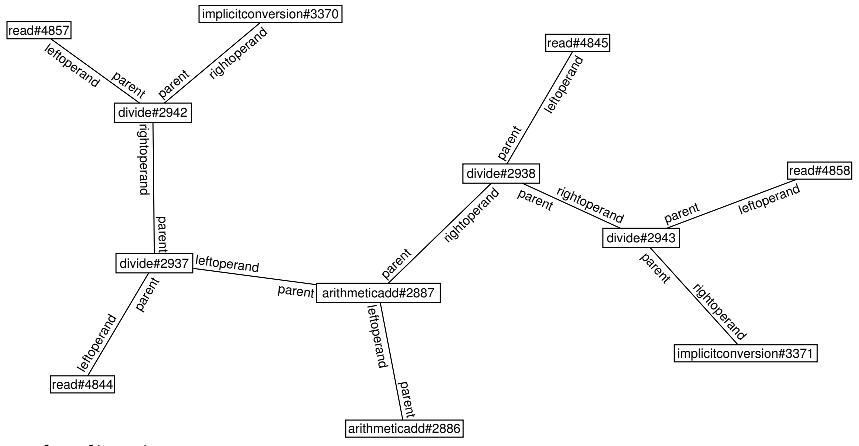
3.3.5 Searching an Optimum

After defining the forces that determine the layout, we have to find a placement where the forces are in an equilibrium. For this we adapt the placement algorithm with adaptive cooling scheme from Hu [Hu05] and include the gradual phase-in of node overlap avoidance from Harel and Koren [HK02]. Hu's algorithm is given in algorithm 1: nodes are repeatedly moved by a fixed step size into the direction indicated by sum of the forces acting upon them. The step size changes between the iterations and is ultimately decreased (in the else branch) by multiplication with a constant factor $0 < t < 1$, which allows the nodes to move closer and closer towards a force equilibrium or energy minimum. However, when the step size is decreased in every iteration, the search is likely to get stuck in a local minimum, which is overcome by not decreasing, or even increasing, the step size while the algorithm makes progress. The progress is measured with an energy function, which is the sum of squared forces on all vertices.[7]

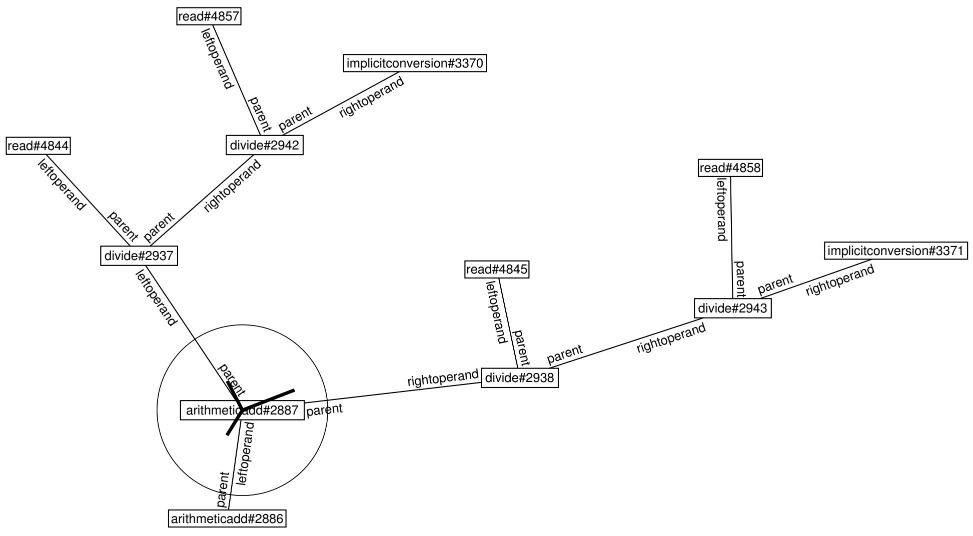
As to when to end the search, we follow Eades [Ead84] and Fruchterman and Reingold [FR91] and simply stop after a fixed number of iterations. This makes it easy to integrate the gradual increase of the importance of overlap avoidance, which is suggested but not explicitly spelled out by Harel and Koren [HK02]. Specifically, an initial placement is found by running k iterations of the search assuming point nodes. During the next l iterations, the size of the nodes is increased step-wise. Finally, m iterations are dedicated to the fine-tuning of the layout with the full size of the nodes taken into account. The node size is increased step by step because a continuous increase is not compatible with Hu's algorithm for determining the step size. When the node size is increased, nodes have to move, which leads to an increase of the total energy. Rising energy usually indicates, that a minimum was overshot, hence the step size is decreased. If the node size is increased continuously, the step size gets smaller and smaller, and no progress is made. Increasing the node size step-wise allows the search algorithm to relax the layout to the new equilibrium. Experimentation shows, that $k = l = m = 50$ is usually sufficient to lead to sub pixel step sizes, thus more iterations would not lead to an improvement.

The calculation of the repulsive forces loops over all pairs of vertices. Quadratic growth is usually undesirable, and there are techniques to calculate such forces in log-lin time. However,

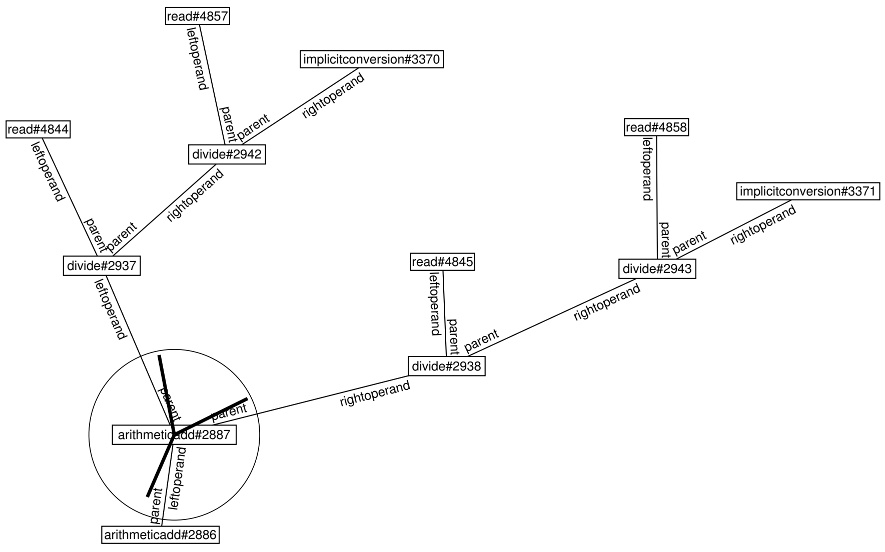
[7]This does not correspond to the physical notion of potential energy, except for the linear force functions in the algorithm of Kamada and Kawai. But it still indicates whether the layout is close to an equilibrium, and one must not take the force metaphor too far.



(a) Arbitrary edge directions.



(b) Using average directions in (a) as preferred directions.



(c) Result after 10 iterations.

Figure 3.10: Determination of directions preferences from data. See text.

Algorithm 1: Force directed placement algorithm with adaptive step length update scheme from Hu [Hu05].

```
Input: $x \in V \to C$ initial placement
Input: $0 < t < 1$ rate of change of step-size
Result: $x$ optimised placement
$step \leftarrow$ initial step length;
$energy \leftarrow \infty$;
$progress \in \mathbb{N} \leftarrow 0$ number of successive steps which improved the layout;
repeat
    $energy' \leftarrow 0$;
    forall the $v \in V$ do
        $\vec{F}(v) = \sum_{v' \not\leftrightarrow v} \vec{F}_R(v, v') + \sum_{v' \leftrightarrow v} \vec{F}_A(v, v') + \sum_e \vec{F}_p(v, e) + \vec{F}_{Frame}(v)$;
        $x(v) \leftarrow x(v) + step * \frac{\vec{F}(v)}{|\vec{F}(v)|}$;
        $energy' \leftarrow energy' + \vec{F}(v)^2$;
    if $energy' < energy$ then Layout has improved
        $progress \leftarrow progress + 1$;
        if $progress = 5$ then
            $progress \leftarrow 0$;
            $stepSize \leftarrow \frac{1}{t} stepSize$;
    else
        $stepSize \leftarrow t\ stepSize$
    $energy \leftarrow energy'$
until $done$;
```

we can only display graphs with a modest number of nodes, otherwise the labels will start to overlap and the graph is not readable anyway. The simple implementation with nested loops needs about 100 ms to layout graphs with 30 nodes on a 2,4 GHz Core i7, which we consider fast enough. With 50 nodes, run-time is around 300 ms, but graphs of this size are usually too cluttered to be of interest. Therefore, we did not investigate the advanced techniques. Besides, it is not clear how they can be extended to include node size.

3.3.6 Minor Topics

The field dependent edge directions help to keep the layouts of nodes of the same type similar. Yet it has to permit nodes to move in order to adapt to the constraints imposed by different reference structures. When the user expands and collapses nodes, those movements are usually a nuisance and have to be avoided. To this end, we clamp all nodes to their position once a graph is displayed. Only newly added nodes will then be subject to the force directed placement algorithm. This clamping pertains only to the individual nodes visible in one window and is not transfered to other objects of the same type.

Finally, the graph display must be prepared to handle changes. Whenever a field of one of the visible nodes is modified, the graph has to be updated. This is handled in the simplest possible way: the set of visible nodes is determined again and a new layout is calculated. Nodes which were visible before stay clamped at their previous position.

3.4 Results

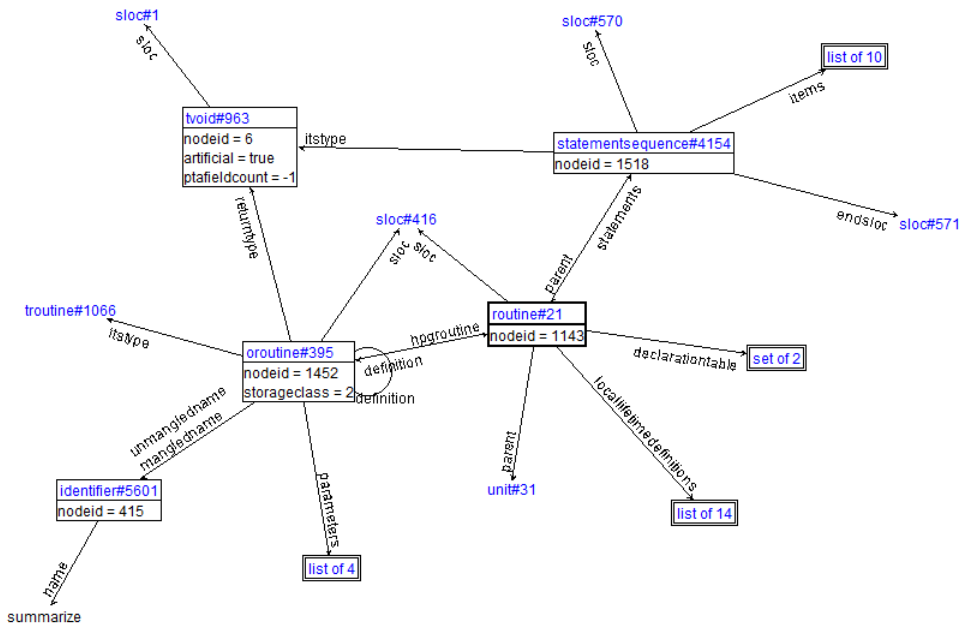
We now present some examples of the layouts that are produced by the algorithm described above. The first example concerns the selection of visible nodes. Figure 3.11 (a) shows some neighbours of routine#21. The expanded node tvoid#963 is reachable over two different paths of length two, viz. definition–returntype and statements–itstype. As both are considered equally likely, both, tpointer#984 and tvoid#963 are expanded in sub-figure (b). This is also a case, where field directions are preserved well.

We, secondly, look at small graphs with a fairly low number of cycles. Such graphs are usually handled well. As reference case, we use the graph around implicitconversion#3111, which is shown in figure 3.12; this graph was chosen almost arbitrarily by visiting objects of random types with all fields expanded until a graph of the desired size was found. Therefore, the edge directions have already emerged: itstype points to the right, and sloc to the left. Parent was set to point upwards initially. Note, that this graph admits a better layout by placing sloc#1 below tpointer#964. But this will not work in the later examples.

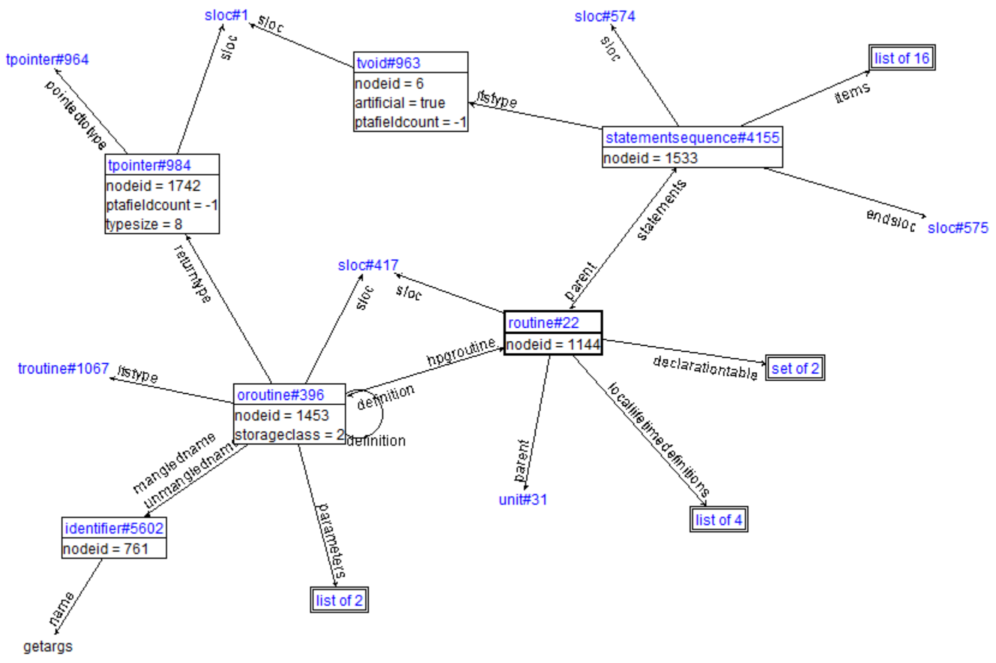
We now examine some objects having the same type. Graphs with an identical structure usually produce identical layouts, such as for example figure 3.13. Without the edge direction preference, the positions of the four pending neighbours of initialise#3683 would be equivalent, and the four fields could swap their positions arbitrarily. The direction force keeps them in place. For graphs like this, the optimisation algorithm produces the desired layout reliably.

When the occurring edges are not too different, the layout is usually only changed locally. Consider the graph around implicitconversion#3114 in 3.14. Unlike in the reference case, the implicitconversion and its parent have the same object as value of their itstype field. The shared type tpointer#964 moves to a compromise position, but the rest of the layout is not affected.

Now, consider unaryminus#3083 in figure 3.15. Its parent is distinctly more to the right than the parents of the previous examples. This is partially caused by the fact that the backward edge is rightoperand instead of source, as it was in the previous cases. Thus, the layout must find a compromise between showing a good unaryminus and showing a good unequal.



(a)



(b)

Figure 3.11: Expanding node tvoid#963 in (a) translates to expanding the returntype of the definition and the itstype of the statements. In (b), this expands two different nodes and the graph is adjusted to accommodate both.

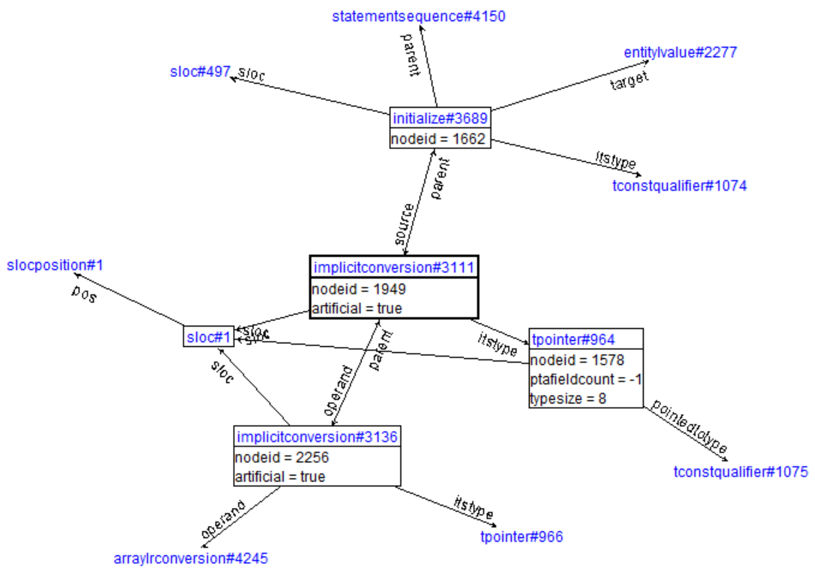


Figure 3.12: Reference case for the exampled concerning the preservation of the direction of fields: a graph chosen almost arbitrarily.

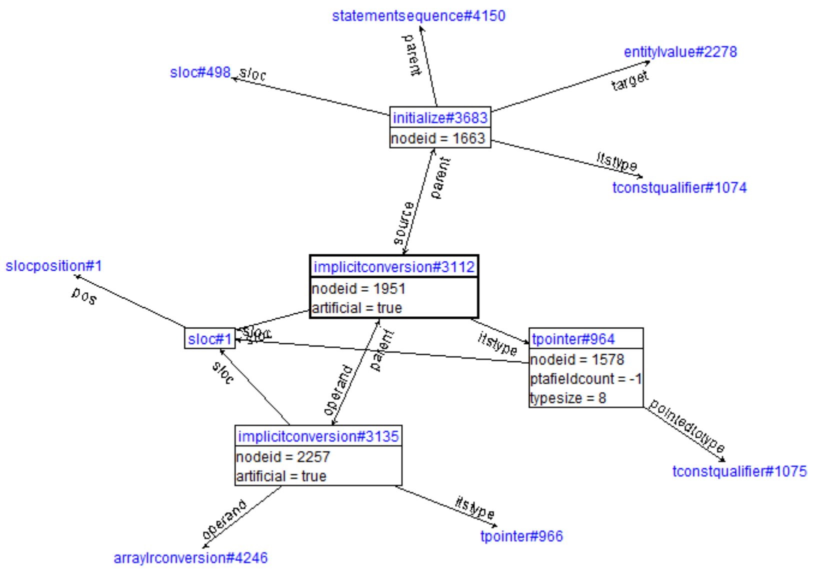


Figure 3.13: A graph with the same structure as the one in figure 3.12. The nodes are placed almost identically.

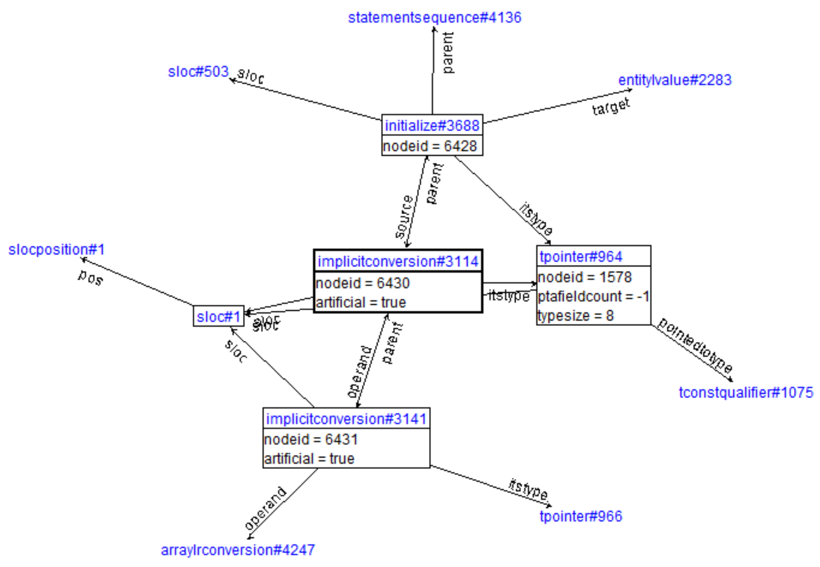


Figure 3.14: A graph with a cycle which was not present in figure 3.12. The position of the node tpointer#964 is a compromise between the requirements of the itstype edges. The rest of the layout is not affected.

However, the examination of further examples showed, that the node was also displaced by sloc#1, for which no good position was found. It would preferably be placed closer to tcint#960. We tried to improve this by implementing using different ideal edge lengths as suggested by Gansner et al. [GKN04]. However, as we need to keep the shortest edges long enough to accommodate a label, the resulting layout becomes too sparse and even fewer objects can be shown simultaneously. There can also not be solved with a better direction for sloc edges because they are so common. The placement of macroinvocation#274 is a non-optimal equilibrium. When this graph was laid out repeatedly, it was usually placed above sloc#976.

Finally, we turn to the problematic cases, which are exemplified in figure 3.16. Recall the example in figure 3.10, where the tree structure of the graph was recognised and neatly represented. However, only the links fields of the tree were considered there. When we add the remaining fields to the picture, the layout deteriorates rapidly. For one thing, many nodes are stuck in a local optimum. For example, consider read#4825, the leftoperand of divide#2926. The only positive thing about its placement is that it is in a good position for a leftoperand. The likelihood of a bad placement decreases with a larger frame, but still a good layout is not found reliably for a graph of this size. The gradual phase-in of the node size improves matters but does not overcome the problem completely. In order to deal with graphs of this size, a more elaborate search procedure has to be employed.

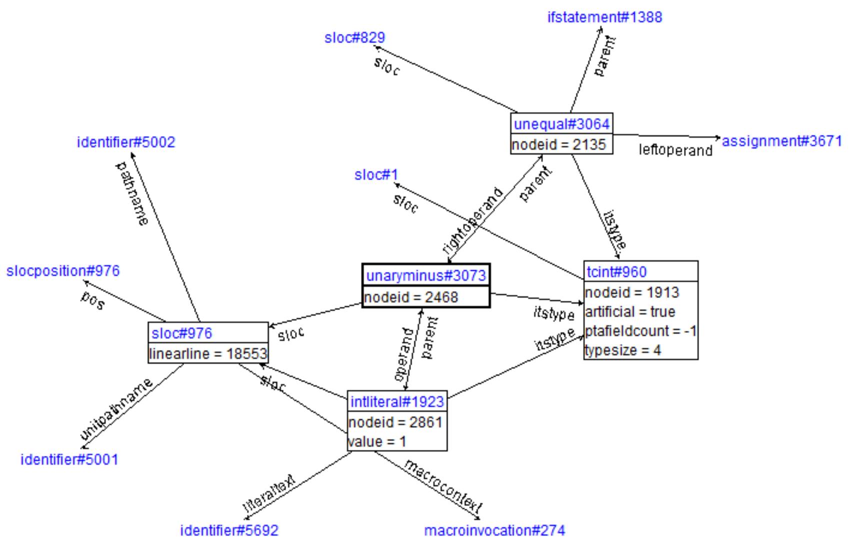


Figure 3.15: The node unequal#3064 is displaced from the desired position of a parent by the rightoperand link and the new position for sloc#1.

The next issue is the node tcint#957, which is the itstype of every expanded node. If we would we align all itstype edges with the preferred direction, tcint#957 would be pushed to one side and distort the remaining graph badly. Hence, the direction force is scaled down in cases like this one, and the node is placed somewhere near the centre. When the algorithm finds good layout of this graph, the tree structure is well represented. It is, however, hardly recognisable due to the crossing itstype edges. It seems clear, that there is no placement of such a highly connected node that prevents this problem. A solution would probably involve a more sophisticated handling of edges, which is an issue we did not investigate.

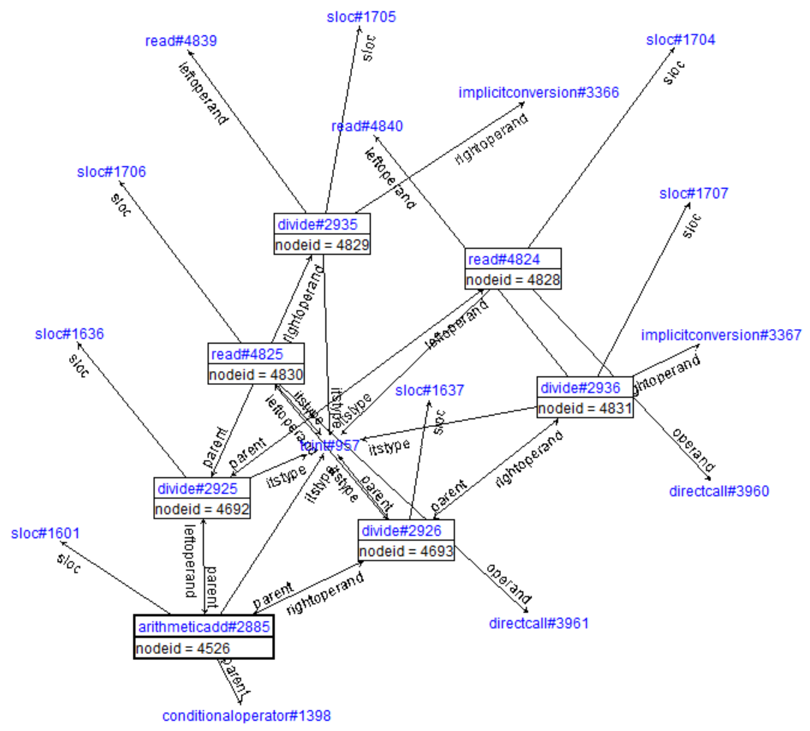


Figure 3.16: A low quality layout of a highly connected graph in a small frame.

4 Viewing the Type System

The type system of the serialised data is an important part of the contents of a SKilL file. Its modification is not in scope of the present work, but it still has to be displayed. Besides being of interest on its own right, the user needs to inspect the type system when they want to create a new object. Also, objects can only be found by enumerating the instances of a type, which has to be selected.

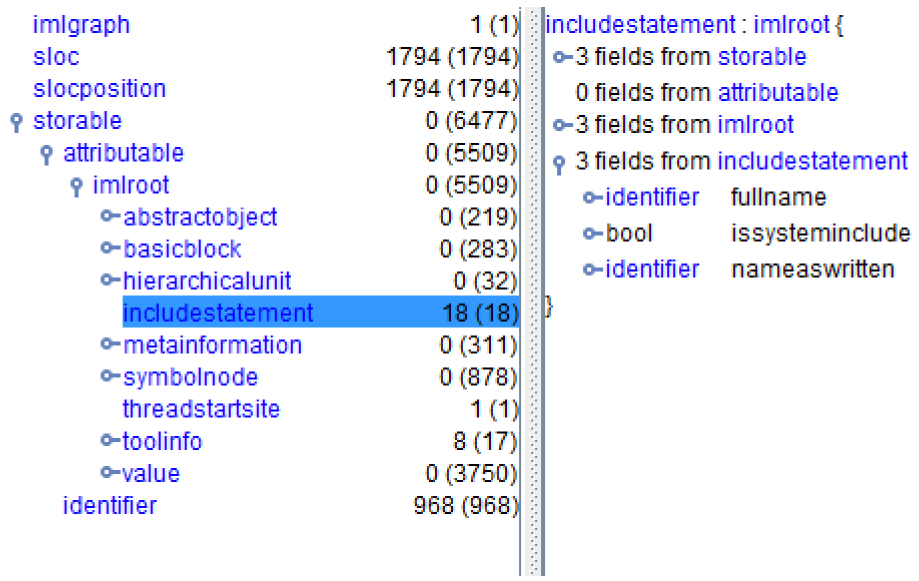
The preceding chapter suggest displaying a class diagram. The sub-type relation would be represented by inheritance edges, fields by associations. However, initial experiments showed that the layout algorithm in chapter 3 is not well suited for this task. While fields of the same type often refer to different objects which can be distributed across the frame, the associations in the type system refer to the same type. Together with sub-type edges, the resulting graph becomes too highly connected and results in visual clutter. We, then, did not pursue a graph representation in the final implementation.

Therefore, the type system is visualised using simple standard approaches. The sub-type hierarchy is displayed in a tree view, and associations with field types become hyper links. As users can be assumed to be familiar with SKilL specification files, type definitions are shown in specification syntax. This is also the only place where constant fields are shown. An example of this representation is given in figure 4.1. The selected type, includestatement, defines three fields—fullname, issysteminclude, and nameaswritten—and inherits from imlroot. Its position in the sub-type hierarchy is shown in the tree on the left side: it has no sub-types and eight siblings. The figures in the middle give the number of instances, both direct and including sub-types.

4.1 Field Preferences

In chapter 3, we introduced some user preferences which apply to fields, viz.

- whether a field is hidden (section 3.2),
- whether a field is represented by an edge in the graph or inside the label of the containing object (section 3.1), and,
- whether the edges representing it should point into a fixed direction (section 3.3.4).



```
imlgraph                         1 (1)   includestatement : imlroot {
sloc                         1794 (1794)   3 fields from storable
slocposition                 1794 (1794)   0 fields from attributable
storable                        0 (6477)   3 fields from imlroot
  attributable                  0 (5509)   3 fields from includestatement
    imlroot                     0 (5509)     identifier   fullname
      abstractobject             0 (219)     bool         issysteminclude
      basicblock                 0 (283)     identifier   nameaswritten
      hierarchicalunit            0 (32)
      includestatement           18 (18) }
      metainformation            0 (311)
      symbolnode                 0 (878)
      threadstartsite              1 (1)
      toolinfo                    8 (17)
      value                     0 (3750)
  identifier                   968 (968)
```

Figure 4.1: Representation of the sub-type hierarchy (left) and type definitions (right).

These settings have nothing to do with the SKilL file itself. Still, there must be a way to edit them for every single field. As every field is already included in the visualisation of the type system, it is convenient to locate the settings there. The visualisation of the type system, thus, serves a second purpose as editor for field preferences.

A user may spend some time to set up the field preferences. They therefore are retained between runs of the program. Of course, identical settings should be used in all SKilL files that adhere to the same specification. It would, however, be too restrictive to use a given set of settings only for files which have an identical type system. After all, SKilL allows every program that processes a file to extend the type system with new fields and types. We, therefore, implemented a liberal approach and simply use identical preferences whenever two fields have the same name and belong to a type with the same name.

5 Editing Objects

The graph view described in chapter 3 is not well suited for modifying objects. The goal, there, was to provide a summary of the relevant information. Therefore, fields were hidden in order to allow following longer chains of references. For editing objects, all fields have to be available and large collections have to be handled. Therefore, objects are displayed a second time in way better suited for editing. This second user interface is described in section 5.1. In section 5.2 we note some issues that arise with the creation and deletion of objects.

5.1 Modifying Fields

This chapter describes how the SKilL fields are translated systematically into elements of a user interface that display the current value of a field and allow the user to modify it. We try to take care, that every possible value of a field can be displayed and also entered by the user, and note where this is not achieved.

The editor for a skill object is then composed from the editors for the fields and labels which identify them. We use a simple layout which places labels above input fields. This keeps the resulting user interface narrow and permits showing it next to the graph viewer. In order to keep the user interface clean and simple, extended functions are only available through pop-up menus.

As the same field can be visible in multiple locations, input fields notify each other about modifications and update their content accordingly.

5.1.1 Primitive Type Fields

Editing booleans and integers poses no problems. Booleans are represented as switches which the user can toggle. Integers are rendered as decimal numbers in text fields and modified by typing the desired new value. Examples of those basic input elements are shown in figure 5.1.

Floating-point numbers are also edited via their string representation as semi-logarithmic decimal figures. How floating-point numbers are represented as strings is completely left to the standard Java libraries [Jav16], which laudably handle all cases one could ask for correctly.

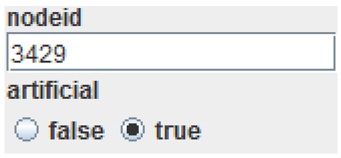


Figure 5.1: Editor elements for primitive type fields: switches and text input.

The displayed string always includes enough significant figures to permit the recovery of the original binary number. Also, the special values ‘infinity’, ‘negative infinity’, and ‘not a number’ can be entered as the strings `Infinity`, `-Infinity`, and `NaN`. There is still a possibility of data loss because different bit patterns are interpreted as the value not-a-number and indiscriminately displayed `NaN`. We do not consider this to be a problem, because storing user data in the mantissa of a not-a-number value is clearly an abuse of floating-point numbers which we do not have to support.

Editing strings, which already are text, should pose no problems at the first sight, but this is not so. First, though strings exhibit a value semantic, they are implemented as references internally and can be null. SKilL treats null string and empty strings, therefore the editor has to reflect the distinction. However, printing null values as `null` will not do either, because this also is a valid string. Also, while strings superficially resemble text, they can contain all kinds of non-printable control characters which can not be displayed and are hard-to-impossible to input.[1] We solve both issues at once by displaying strings as quoted string literals with control characters represented by backslash escape sequences. The supported escape sequences are those defined by the Scala language [Oa04], which can be processed by existing functions in the Scala runtime library. This also frees the string `(null)` for null references.

There are some issues with the treatment of strings. The implemented solution to ensure that nulls and control characters are treated correctly makes it awkward to work with normal strings. It would also be desirable to display strings which contain multiple lines with their line breaks intact. We also took no precautions to prevent excessively large strings from crashing the editor. Finding a general solution for the handling of string was, however, not in the scope of this work.

5.1.2 Reference Type Fields

References are represented by text fields showing the type and SKilL ID of the object they refer to or literal `(null)`. Non-null reference fields can be expanded to show the values of the fields of the referenced object. Unlike the view in chapter 3, no fields are filtered out, here, and

[1] Our experience is, that after one has found a way to type most of them, like Ctrl-letter or Windows Alt-sequences, the backspace control character stubbornly back-deletes instead of being inserted.

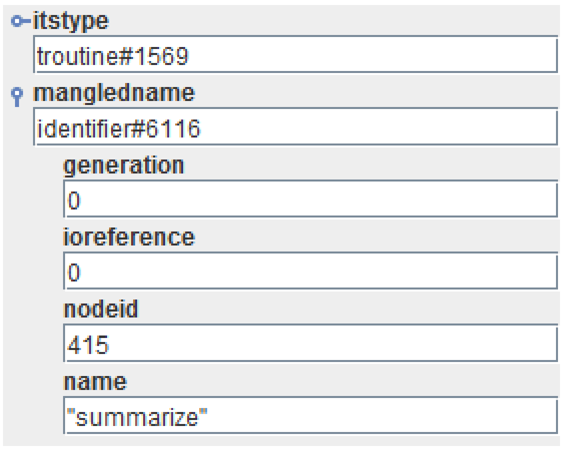

itstype
troutine#1569
mangledname
identifier#6116
generation
0
ioreference
0
nodeid
415
name
"summarize"

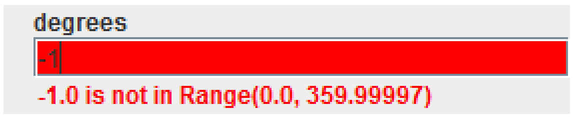
Figure 5.2: Editor elements for reference fields. The field mangledname is expanded and shows the fields of the object it refers to.

nothing is expanded automatically. Figure 5.2 shows how reference fields are displayed in the edit view.

The most direct way to modify a reference field is to enter a type name and ID manually. This is, however, not sufficient, because the ID is a meaningless number and has to be looked up first. The user can, therefore, also browse a list of existing objects and select one of them. Of course, the considerations from chapter 3 about displaying objects apply again, and we want to have the full set of features mentioned there available. Also, the user might be able to locate the object they want to use, or at least restrict the set of candidates, by a search query. Therefore, the graph viewer can operate in an object selection mode. In this mode, the currently viewed object can be accepted by pressing an ‘OK’ button and is passed to a function provided by the action that started the selection process. In the present case of selecting a new value for a field, this updates the field and returns the editor to the object in which the selection process was started.

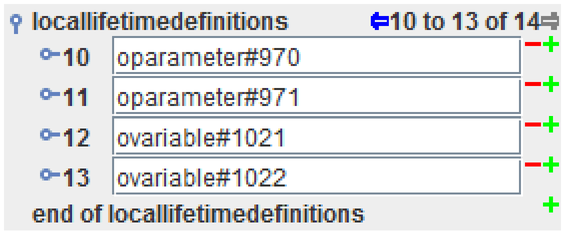
5.1.3 Restrictions

Field restrictions are checked as part of the validation of user inputs. Values that do not pass all restrictions stay in the user interface and are not written to the SKilL data structures. Text inputs indicate violated restrictions—as well as other validation failures, like ‘is not a number’—with red background colour and an error message is shown below the input field. Figure 5.3 gives an example of this. For field restrictions, the SKilL binding already provides us with error messages why a value violated a restriction, which are shown directly.



-1.0 is not in Range(0.0, 359.99997)

Figure 5.3: An editor element which violates a SKilL restriction.



localliftimedefinitions
10 oparameter#970
11 oparameter#971
12 ovariable#1021
13 ovariable#1022
end of locallifetimedefinitions

Figure 5.4: Editor elements for a list. The second page of the list locallifetimedefinitions is visible.

5.1.4 Compound Types

Of the compound types, lists and arrays are most easily dealt with. Their members have a primitive or reference type, and we have already covered how we handle values of those types. The editors for members of a list are, therefore, extensions of the editors for ground type fields. The only difference is, that they access a specific index in a list field instead of the field itself. The editor for a complete list combines editors for the members and adds labels. Lists and variable length arrays also support the deletion and insertion of elements. Deletion poses no problem, insertion is easily implemented by inserting a default-initialised element at the indicated position.

Displaying a long list completely would lead to excessive scrolling and would make it hard to find fields that come between two long collections. Therefore, all collections are split into groups of a few—per default 10—entries and shown one group at a time. To access entries that are not in the first group, the user can page to other groups. An example of a list editor is shown in figure 5.4.

Sets are handled similarly. However, we must keep in mind that they can not contain the same element twice. The input elements for set members add, therefore, a restriction that prevents entering a value that is already a member of the set. We also can not insert default-initialised elements here, because the default value could already be part of the set. The value of a new element is, therefore, queried with a dialogue. In the case of reference types, it would be desirable to be able to select an existing object or create a new one. This is, however, not implemented.

Maps are, again, the hard case. Like in chapter 3, they are treated as containers that are indexed with a key tuple. The desired consequence of this is, that this makes it possible to speak of the size of the map as a whole. Consequently, it is possible to show the map page by page and keep the size which it occupies in the user interface small. Treating maps as chained functions would lead to showing a page of maps, all of which can have many entries themselves. We did not notice that this misses empty functions, thus, a better solution has to be found.

They are, thus, displayed like lists where the values of the keys take the place of the index. Keys can not be modified, but reference keys admit the inspection of their fields. For adding a new entry, values for all keys are queried successively. Like with sets, an object selection function is missing.

5.2 Object Deletion and Creation

Some peculiarities have to be taken care of when it comes to the deletion and creation of objects. We have to take precautions for the proper operation of the undo/redo function across deletions and creations, and address the issue, that newly created objects do not have a SKilL ID.

An issue, that is not addressed is the restrictions which can be violated when an object is created or deleted. Those violations will be detected by the binding when a file is saved, but this does not help the user to find the offending fields in order to fix them. Deletions can violate non-null restrictions anywhere in the file, therefore this is an issue. The editor should therefore detect violated restrictions and provide quick navigation to the objects that contain them. This was, however, not implemented.

5.2.1 Undoing Deletions

When the user undoes a deletion, the object must be re-created. But then, older edits become available for undo, some of which might be modifications of the fields of the undeleted object. It is therefore important, that the original object, which the other edits in the undo-redo-chain refer to, reappears when a deletion is undone. The same problem arises when a newly created object is deleted via undo and later re-created via redo. Unfortunately, the Scala SKilL binding does not export functions that would restore a deleted object to its previous, not yet deleted, state. Therefore, we keep a set of objects marked for deletion and defer the actual deletion to the moment immediately before the changes are written back to disk, at which point we simply discard the undo/redo chain. So all cases of object deletion—the user explicitly deletes an object, a deletion is re-done, or a creation is un-done—just add the object to the set of objects that are deleted before saving. This can be safely undone by removing it from the set.

As we do not actually delete the objects, we should take care, that virtually deleted objects are not shown to the user. All references to an object are removed before it is deleted, therefore it is not reachable from other objects anymore. The search function removes deleted objects from the result. Thus, one can not navigate to a deleted object.

It can, however, still be visible in another window of the editor and found through the back-navigation buffer. We did, however, not complete an implementation that addresses this.

5.2.2 IDs for New Objects

SKilL IDs are derived from the object's position in the file and are assigned to newly created objects upon serialisation. A new object has therefore no ID, or, to be precise, all new objects share the dummy ID -1. As the ID is valuable for telling objects of the same type apart, we provide surrogate IDs for newly created objects. To keep things simple and the surrogated IDs recognisable, all newly created objects are assigned consecutive descending IDs starting with -1. Lookup tables are used to find the surrogate ID of objects and objects by their ID. The lookups only have to be performed if the actual ID of the object is -1, otherwise the functions of the SKilL binding are used.

6 Search

As discussed before in chapter 3, the meaning of an object can not be understood from the values of it fields alone. Rather, the fields of the objects it refers to have also to be taken into consideration. Likewise, when the user searches an object, or a set of objects, they need to be able to put constraints on referenced objects in order to sufficiently narrow down the results set.

This very same problem is already addressed by the SPARQL query language [PS08], which is used to query information from semantic web data repositories. The simplest SPARQL query is the triple pattern $?x\ r\ c$, where $?x$ is a variable, r is a relation in the queried data set, and c is a literal or the name of an object. This query will retrieve all objects that are in r-relation with c. The basic pattern can be combined to form more sophisticated queries like $?x\ r\ ?y\ .\ ?y\ s\ c$,[1] which returns all pairs of objects $(?x, ?y)$ where $?x$ is r-related to $?y$ and $?y$ is s-related to c. This neatly translates to SKilL files when we treat fields as relations and say that $o\ r\ x$ iff $o.r = x$.

The implementation of the search function was not out main objective. It is, however, necessary to provide a way to list all objects of a type as well as the members of a large container. Therefore, a search function, albeit not very elaborate, is still included because it is a convenient place for handling those lists as special cases.

6.1 The Query Language

We implemented a query language that resembles a highly simplified subset of SPARQL. In fact, it is confined to triples and joins. We now specify the syntax and semantics of the query language in lock step.

We state the semantics by means of a partial assignment function g that maps variables to values of a universe $\mathcal{U}$. The universe contains the values of the primitive types and the set of all SKilL objects. A valuation function $V_g(\cdot)$ is then defined, which specifies whether a query is satisfied under a given assignment in a SKilL file. The search result is then the set of all assignments that satisfy the query and are defined exactly for the variables that occur it.

[1]One may, in the tradition of Peano and Russel, read the dot as *and*, here, but it is actually inherited from the RDF triple notation where it terminates the statement of a fact.

6.1.1 Lexical Tokens

The query language uses the following symbols and reserved words:

$$\texttt{.} \quad \texttt{type} \quad \texttt{directType} \tag{6.1}$$

Further lexical tokens are:

- *Identifiers* (Ident), which follow the usual C rules, i.e. their first character is a Latin letter, underscore, or universal character name. All further characters also admit numbers. Keeping with the SKilL specification [Fel17], we also allow all Unicode characters above \u007f.

  For cases where a reserved word hides an identifier, identifiers can be stropped with single straight quotation marks (apostrophes), thus: `'type'`. The use of universal character names is not possible in this case.
- *Integer number literals* (Int), i.e. decimal numbers with an optional sign.
- *Floating-point number literals* (Float), floating-point numbers, optionally in exponential notation.
- *String literals* (String), arbitrary text enclosed in quotation marks with the usual processing of backslash escape sequences.
- *Object names* (ObjId), which consist of an identifier part which names the base pool or the storage pool the object belongs to, a pound sign, and the SKilL-ID of the object as integer literal, e.g. `unit#5` or `'type'#7`.
- *Variables* (Var), which consist of a dollar sign or a question mark followed by an identifier without intervening white space. `?x` and `$x` can be used interchangeably.

6.1.2 Terms

The terms of the queries are the literals and variables:

$$\mathit{Term} ::= \mathit{Var} \mid \mathit{Int} \mid \mathit{Float} \mid \mathit{String} \mid \text{true} \mid \text{false} \mid \mathit{ObjId}$$

A term t has a value $[\![t]\!]_g$. The value of a literal term is just the value of the literal itself, thus, for example, $[\![\texttt{1}]\!]_g = 1$, $[\![\texttt{true}]\!]_g = \top$, and so on. The value of an object name is the object in the skill file: $[\![\texttt{t\#1}]\!]_g$ is the object of user type t with SKilL ID 1. The value of a variable v is the determined by the assignment function: $[\![v]\!]_g = g(v)$.

6.1.3 Basic Triple Pattern

The basic query is one of the triples:

$$\begin{aligned} \mathit{Triple} ::= &\ \mathit{Term}\ \texttt{Ident}\ \mathit{Term} \\ |&\ \mathit{Term}\ \texttt{Ident.Ident}\ \mathit{Term} \\ |&\ \dots \end{aligned}$$

Informally, a triple $o\ f\ v$ means, that the value of field f in object o is v. Thus, $V_g(o\ f\ v) = 1$ if $[\![o]\!]_g$ is a SKilL object which has a field with name f and $[\![o]\!]_g.f = [\![v]\!]_g$. Note the sufficient condition; there are other ways to satisfy a triple which involve compound types. These are explained later. However, a triple can be never satisfied when the left term is not an object.

The second variant, $o\ t.f\ v$, is used to disambiguate between multiple fields with the same name: here, o must have a field with name f that was defined in a type with name t itself, not one of its super types. The basic form, $o\ f\ v$, uses all fields with a given name, i.e. it depends on the type of o which one is used. This may change dynamically during the execution of the query because the objects o can have different types.

6.1.4 Pseudo Triple Patterns

Unlike SPARQL, we do not treat types as first-class values. Therefore, there are two special cases for restricting the types of objects:

$$\begin{aligned} \mathit{Triple} ::= &\ \mathit{Term}\ \texttt{type}\ \mathit{Ident} \\ |&\ \mathit{Term}\ \texttt{directType}\ \mathit{Ident} \\ |&\ \dots \end{aligned}$$

A query o `type` t is satisfied if o is an object of type t, i.e. $V_g(o\ \texttt{type}\ t) = 1$ iff $[\![o]\!]_g$ is a SKilL object of type t. With `directType`, it must also be an instance of t directly, not of a sub-type of it.

6.1.5 Collections

The simplest query one may run against a set is the enumeration of its elements. This query uses the basic triple pattern. It depends on the type of the collection how the elements in it are retrieved. For sets, the reading is clear: $V_g(o\ f\ v) = 1$ if $[\![o]\!]_g$ is a SKilL object which has a set field with name f and $[\![v]\!]_g \in [\![o]\!]_g.f$. The elements of arrays and lists are just returned disregarding their index: $V_g(o\ f\ v) = 1$ if $[\![o]\!]_g$ is a SKilL object which has a list or array field with name f and $\exists i : [\![o]\!]_g.f_i = [\![v]\!]_g$. Maps, in line with their treatment in previous chapters, are regarded to be collections of values indexed by a key tuple and return their rightmost elements:

$V_g(o\ f\ v) = 1$ if $[\![o]\!]_g$ is a SKilL object which has a map field with type $\tau_1 \to \cdots \to \tau_n$ and name f and $\exists x_1 : \tau_1, \ldots x_{n-1} : \tau_{n-1} : [\![o]\!]_g.f(x_1)\cdots(x_{n-1}) = [\![v]\!]_g$.

For lists, and especially for maps, it would be desirable to be able to retrieve the indices and keys. We suggest a syntax along the lines of $o\ f(k_1, k_2, \ldots)\ v$, but did not finish an implementation.

6.1.6 Join

A join is a sequence of triples separated by dots:

$$\begin{aligned} \mathit{JoinFactor} &::= \mathit{Triple} \\ \mathit{Join} &::= \mathit{JoinFactor}\ (.\ \mathit{JoinFactor})^*\ [.] \end{aligned}$$

In a join, all factors have to be satisfied, therefore $V_g(f_1\ .\ f_2\ .\ \ldots\ .\ f_n) = 1$ iff $V_g(f_i) = 1$ for every $1 \leq i \leq n$.

6.1.7 Special Cases

We add two special queries: a single identifier is taken to be an abbreviation for a type query, which we assume to be common. A single object literal finds the identified object. This does not correspond to any normal query and is added in order to allow direct navigation to an object of which the ID is known.

$$\begin{aligned} \mathit{Query} &::= \mathit{Join} \\ &\quad |\mathit{Ident} \\ &\quad |\mathit{ObjId} \end{aligned}$$

So, for an identifier i, object literal o, and a new variable $?v$: $V_g(t) = V_g(?v\ \texttt{type}\ t)$ and $V_g(o) = 1$ iff $g(v) = o$.

6.2 Query Execution

A query can easily produce a huge set of results: ?a `type t`. ?b `type t` would return all pairs of objects of type t. We consider such queries too expensive and reject their execution. Search being not our main priority, we did not investigate how and which queries can be executed efficiently in general. The only composite queries we want to handle are the ones that filter objects based on the value of a field in a child object. For example, in order to allow looking up all objects a with $a.definition.mangledname.name$ = "main", execution of the query `?a definition ?b. ?b mangledname ?c. ?c name "main"` is supported.

The joins are implemented by a simple procedure that can handle this special case efficiently. The procedure would be lengthy to spell out in detail, because a lot of cases have to be considered. As it is rather simpleminded, we do not consider it noteworthy and confine the description to one example.

First, the search expressed by one of the factors is executed. This may involve a scan over a large part of the SKilL file. The factor `?a definition ?b`, for example, requires the enumeration of all instances of types which have a field 'definition'. This yields an initial list of results which include an assignment $(?a \mapsto a, ?b \mapsto b)$ iff $a.\mathit{definition} = b$. The result of this initial search is then filtered or extended by each of the other factors in turn. This must be possible without enumerating all instances of a type a second time. Therefore, factors of the form $?v\ f\ x$ are only allowed, if the variable $?v$ is already assigned in the result obtained so far. New values can only be obtained by reading the fields of objects that are already known. In the example, `?b mangledname ?c` is processed next. Every assignment $(?a \mapsto a, ?b \mapsto b)$ from the initial set is extended to $(?a \mapsto a, ?b \mapsto b, ?c \mapsto c)$ with $b.\mathit{mangledname} = c$. Now ?c is bound and `?c name "main"` can be processed, which restricts the results to the instances $(?a \mapsto a, ?b \mapsto b, ?c \mapsto c)$ with $c.\mathit{name}$ = "main". That is $a.definition.mangledname.name$ = "main" as intended.

Thus, one factor contributes a potentially huge list of candidate results. The remaining factors extend or filter this list entry by entry. The list can only be lengthened when enumerating elements of containers. The order in which the factors have to be processed to admit execution by this procedure is found by an exhaustive search. If no such order is found, the query is rejected.

After the execution of a query, the search results are tabulated in textal form. When a row of the result is selected, the first object in row is shown in the graphical editor.

7 Conclusion

The functions presented in the preceding chapters permit the inspection and modification of the contents of binary SKilL files in a graphical user interface.

A plain representation was chosen for the type system that is stored in the binary file. The sub-type hierarchy is visualised in a hierarchical outline. The definition of the type of a field can be looked up through a hyper link. Modification of the type system is outside the scope of this work. Consequently, no new binary files can be created.

The focus of this thesis is the visual representation of the objects and their mutual references in a node-link diagram. Fields of objects can be freely shown and hidden in the output. The choice which fields are shown is reused when another object of the same type is encountered later by associating it with the sequence of references through which an object was found. For the layout of the node-link diagram, classical solutions for force directed graph layout were reviewed, and solutions for different sub-problems were combined. In order to produce comprehensible layouts, the algorithm is extended to constrain the mutual position of objects depending on the references between them. The presented solution produces reasonable results as long as the number of visible objects is small enough and the edges do not form too many cycles. The object graph can be navigated by following references and by search.

With a few exceptions, fields of objects can be modified in accordance with the serialised type system. Deletion and creation of objects is supported and all modifications can be undone. However, no suitable solution was found for nested maps, and the implemented representation does not cover maps with empty sub-maps. The implementation is further not prepared to handle huge string fields. While most modifications are supported, there is room for improving the usability. Notable issues are the lack of the possibility to select existing objects in all places where this would be desirable and the poor handling of multi line strings.

Objects can be located by a search function which supports a limited set of join queries in order to filter objects based on indirectly referenced fields. The solution is based on established technology but incomplete. It is mainly included in order to provide a few essential functions like the enumeration of all objects of a type. For practical purposes, a more powerful solution is desirable. The inclusion of numerical inequalities and negative queries could follow the established model. How to satisfyingly handle maps needs to be investigated.

Whether the presented solution meets the needs of users has to be seen in practice. The unconstrained node-link diagrams used in the present work appear to require a lot of space for a clear layout. Thus, the number of objects which can be visible simultaneously is severely limited. To investigate whether this can be addressed with a better placement algorithm, or whether a different approach, which permits the information to be packed more densely, for example based on hierarchical outlines, would be preferable, is left for future work.

Glossary

API Application programming interface, the set of subroutines and data types through which an application program uses the services of a software library. 9

base type A type which has no super type. A root of the sub-type hierarchy. 10

built-in type One of the basic data types available in SKilL. A primitive type or an annotation. 11

compound type A data type which is composed of other types. The SKilL, compound types are formed by instantiating the type schemata list-of-type, set-of-type, &c. with ground types. 11, 24, 54

field type A type that can be used as the type of the field of a user type. A ground type or a compound type. 10

per-type A user type that, directly or indirectly, is extended by another user type is a super type of the latter. The inverse relation is sub-type. 10

primitive type One of the integer, floating point number, boolean, and string data types. 11, 23, 51

reference type A type, the values of which are references to SKilL objects. A user type reference or an annotation. 11, 22, 52

Scala A multi paradigm object oriented functional programming language based on Java. The name is from 'scalable language'. 7, 9

SKilL Serialisation killer language, a platform independent file format for storing complex data structures. 9

SKilL ID A tuple of a type name and a consecutive number which uniquely identifies a SKilL object within a file. 10, 58

SKilL object A record of user data stored in a SKilL file. Has a user type. 9, 22

sub-type A user type that, directly or indirectly, extends another user type is a sub-type of the latter. The inverse relation is super type. 10

user type A type consisting of named fields which have field types and optionally extending another user type. 10

Bibliography

[DBETT94] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. *Comput. Geom. Theory Appl.*, 4(5):235–282, 1994. doi:10.1016/0925-7721(94)00014-X. URL `http://dx.doi.org/10.1016/0925-7721(94)00014-X`.

[DH96] R. Davidson, D. Harel. Drawing Graphs Nicely Using Simulated Annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996. doi:10.1145/234535.234538. URL `http://doi.acm.org/10.1145/234535.234538`.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[Fel17] T. Felden. The SKilL Language V1.0. Technischer Bericht Informatik 2017/01, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau, 2017. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2017-01&engl=`.

[FR91] T. M. J. Fruchterman, E. M. Reingold. Graph Drawing by Force-directed Placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991. doi:10.1002/spe.4380211102. URL `http://dx.doi.org/10.1002/spe.4380211102`.

[GJ79] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GKN04] E. R. Gansner, Y. Koren, S. C. North. Graph Drawing by Stress Majorization. In J. Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pp. 239–250. Springer, 2004. URL `http://dblp.uni-trier.de/db/conf/gd/gd2004.html#GansnerKN04`.

[GN00] E. R. Gansner, S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[HK02] D. Harel, Y. Koren. Drawing Graphs with Non-uniform Vertices. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, pp. 157–166. ACM, New York, NY, USA, 2002. doi:10.1145/1556262.1556288. URL http://doi.acm.org/10.1145/1556262.1556288.

[Hu05] Y. F. Hu. Efficient, High Quality Force-Directed Graph Drawing. *The Mathematica Journal*, 10:37–71, 2005. URL http://www.mathematica-journal.com/issue/v10i1/contents/graph_draw/graph_draw.pdf.

[Jav16] Oracle. *Java Platform Standard Edition 8 API Specification*, 2016. URL http://docs.oracle.com/javase/8/docs/api/.

[KK89] T. Kamada, S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989. doi:10.1016/0020-0190(89)90102-6. URL http://dx.doi.org/10.1016/0020-0190(89)90102-6.

[Oa04] M. Odersky, al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[PS08] E. Prud'hommeaux, A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008. URL http://www.w3.org/TR/rdf-sparql-query/. http://www.w3.org/TR/rdf-sparql-query/.

Declaration